第二周周报

Linux rootkit 组员：柴宇龙，霍南风，林坤，王春萍。第二周工作如下：

1. 柴宇龙

1.1 2.4 和 2.6 内核版本的模块机制的差异

1.1.1 模块编译

从 2.4 到 2.6，外部可装载内核模块的编译、连接过程以及 Makefile 的书写都发生了改变。

2.4 内核中，模块的编译只需内核源码头文件；需要在包含 linux/modules.h 之前定义 MODULE；编译、连接后生成的内核模块后缀为.o。

2.6 内核中，模块的编译需要配置过的内核源码；编译、连接后生成的内核模块后缀为.ko；编译过程首先会到内核源码目录下，读取顶层的 Makefile 文件，然后再返回模块源码所在目录。

2.4 内核模块的 Makefile 模板

```
2. #Makefile2.4
3. KVER=$(shell uname -r)
4. KDIR=/lib/modules/$(KVER)/build
5. OBJS=mymodule.o
6. CFLAGS=-D__KERNEL__ -I$(KDIR)/include -DMODULE
7.     -D__KERNEL_SYSCALLS__ -DEXPORT_SYMTAB
8.   -O2 -fomit-frame-pointer  -Wall  -DMODVERSIONS
9.          -include $(KDIR)/include/linux/modversions.h
10.     all: $(OBJS)
11.     mymodule.o: file1.o file2.o
12.             ld -r -o $@ $^
13.     clean:
14.             rm -f *.o
```

在 2.4 内核下，内核模块的 Makefile 与普通用户程序的 Makefile 在结构和语法上都相同，但是必须在 CFLAGS 中定义-D__KERNEL__-DMODULE，指定内核头文件目录-I$(KDIR)/include。 有一点需注意，之所以在 CFLAGS 中定义变量，而不是在模块源码文件中定义，一方面这些预定义变量可以被模块中所有源码文件可见，另一方面等价于将这些预定义变量定义在源码文件的起始位置。在模块编译中，对于这些全局的预定义变量，一般在 CFLAGS 中定义。

2.6 内核模块的 Makefile 模板

```
15.     # Makefile2.6
16.     ifneq ($(KERNELRELEASE),)
17.     #kbuild syntax. dependency relationshsip of files and
   target modules are listed here.
18.     mymodule-objs := file1.o file2.o
19.     obj-m := mymodule.o
20.     else
21.     PWD  := $(shell pwd)
```

```
22.     KVER ?= $(shell uname -r)
23.     KDIR := /lib/modules/$(KVER)/build
24.     all:
25.             $(MAKE) -C $(KDIR) M=$(PWD)
26.     clean:
27.     rm -rf .*.cmd *.o *.mod.c *.ko .tmp_versions
28.     endif
```

KERNELRELEASE 是在内核源码的顶层 Makefile 中定义的一个变量，在第一次读取执行此 Makefile 时，KERNELRELEASE 没有被定义， 所以 make 将读取执行 else 之后的内容。如果 make 的目标是 clean，直接执行 clean 操作，然后结束。当 make 的目标为 all 时，-C $(KDIR) 指明跳转到内核源码目录下读取那里的 Makefile；M=$(PWD) 表明然后返回到当前目录继续读入、执行当前的 Makefile。当从内核源码目录返回时，KERNELRELEASE 已被定义，kbuild 也被启动去解析 kbuild 语法的语句，make 将继续读取 else 之前的内容。else 之前的内容为 kbuild 语法的语句, 指明模块源码中各文件的依赖关系，以及要生成的目标模块名。mymodule-objs := file1.o file2.o 表示 mymoudule.o 由 file1.o 与 file2.o 连接生成。obj-m := mymodule.o 表示编译连接后将生成 mymodule.o 模块。

补充一点，"$(MAKE) -C $(KDIR) M=$(PWD)"与"$(MAKE) -C $(KDIR) SUBDIRS =$(PWD)"的作用是等效的，后者是较老的使用方法。推荐使用 M 而不是 SUBDIRS，前者更明确。

通过以上比较可以看到，从 Makefile 编写来看，在 2.6 内核下，内核模块编译不必定义复杂的 CFLAGS，而且模块中各文件依赖关系的表示简洁清晰。

1.1.2 模块的初始化与退出

在 2.6 内核中，内核模块必须调用宏 module_init 与 module_exit() 去注册初始化与退出函数。在 2.4 内核中，如果初始化函数命名为 init_module()、退出函数命名为 cleanup_module()，可以不必使用 module_init 与 module_exit 宏。推荐使用 module_init 与 module_exit 宏，使代码在 2.4 与 2.6 内核中都能工作。

适用于 2.4 与 2.6 内核的模块的初始化与退出模板

```
#include <linux/module.h>  /* Needed by all modules */
#include <linux/init.h>    /* Needed for init&exit macros */
static int mod_init_func(void)
{
/*code here*/
return 0;
}
static void mod_exit_func(void)
{
/*code here*/
}
module_init(mod_init_func);
module_exit(mod_exit_func);
```

需要注意的是初始化与退出函数必须在宏 module_init 和 module_exit 使用前定义，否则会出现编译错误。

1.1.3 模块使用计数

模块在被使用时，是不允许被卸载的。2.4 内核中，模块自身通过 MOD_INC_USE_COUNT、MOD_DEC_USE_COUNT 宏来管理自己被使用的计数。2.6 内核提供了更健壮、灵活的模块计数管理接口 try_module_get(&module)及 module_put(&module)取代 2.4 中的模块使用计数管理宏；模块的使用计数不必由自身管理，而且在管理模块使用计数时考虑到 SMP 与 PREEMPT 机制的影响。

int try_module_get(struct module *module)：用于增加模块使用计数；若返回为 0，表示调用失败，希望使用的模块没有被加载或正在被卸载中。

void module_put(struct module *module)：减少模块使用计数。

try_module_get 与 module_put 的引入与使用与 2.6 内核下的设备模型密切相关。模块是用来管理硬件设备的，2.6 内核为不同类型的设备定义了 struct module *owner 域，用来指向管理此设备的模块。如字符设备的定义：

```
struct cdev {
        struct kobject kobj;
        struct module *owner;
        struct file_operations *ops;
        struct list_head list;
        dev_t dev;
        unsigned int count;
};
```

从设备使用的角度出发，当需要打开、开始使用某个设备时，使用 try_module_get(dev->owner)去增加管理此设备的 owner 模块的使用计数；当关闭、不再使用此设备时，使用 module_put(dev->owner)减少对管理此设备的 owner 模块的使用计数。这样，当设备在使用时，管理此设备的模块就不能被卸载；只有设备不再使用时模块才能被卸载。

2.6 内核下，对于为具体设备写驱动的开发人员而言，基本无需使用 try_module_get 与 module_put，因为此时开发人员所写的驱动通常为支持某具体设备的 owner 模块，对此设备 owner 模块的计数管理由内核里更底层的代码如总线驱动或是此类设备共用的核心模块来实现，从而简化了设备驱动开发。

1.1.4 模块输出的内核符号

2.4 内核下，缺省情况时模块中的非静态全局变量及函数在模块加载后会输出到内核空间。

2.6 内核下，缺省情况时模块中的非静态全局变量及函数在模块加载后不会输出到内核空间，需要显式调用宏 EXPORT_SYMBOL 才能输出。所以在 2.6 内核的模块下，EXPORT_NO_SYMBOLS 宏的调用没有意义，是空操作。在同时支持 2.4 与 2.6 内核的设备驱动中，可以通过以下代码段来输出模块的内核符号

同时支持 2.4 与 2.6 的输出内核符号代码段

```
#include <linux/module.h>
#ifndef LINUX26
```

```
EXPORT_NO_SYMBOLS;
#endif
EXPORT_SYMBOL(var);
EXPORT_SYMBOL(func);
```

需要注意的是如需在 2.4 内核下使用 EXPORT_SYMBOL，必须在 CFLAGS 中定义 EXPORT_SYMTAB，否则编译将会失败。

从良好的代码风格角度出发，模块中不需要输出到内核空间且不需为模块中其它文件所用的全局变量及函数最好显式申明为 static 类型，需要输出的内核符号以模块名为前缀。

模块加载后，2.4 内核下可通过 /proc/ksyms、 2.6 内核下可通过/proc/kallsyms 查看模块输出的内核符号

1.1.5 模块的命令行输入参数

在装载内核模块时，用户可以向模块传递一些参数，如`modprobe modname var=value`，否则，var 将使用模块内定义的缺省值。

2.4 内核下，linux/module.h 中定义有宏 MODULE_PARM(var,type) 用于向模块传递命令行参数。var 为接受参数值的变量名，type 为采取如下格式的字符串[min[-max]]{b,h,i,l,s}。min 及 max 用于表示当参数为数组类型时，允许输入的数组元素的个数范围；b：byte；h：short；i：int；l：long；s：string。

2.6 内核下，宏 MODULE_PARM(var,type)不再被支持。在头文件 linux/moduleparam.h 里定义了如下宏：

```
module_param(name, type, perm)
module_param_array(name, type, nump, perm)
```

type 类型可以是 byte、short,、ushort、 int、 uint、long、ulong、charp, bool or invbool, 不再采用 2.4 内核中的字符串形式，而且在模块编译时会将此处申明的 type 与变量定义的类型进行比较，判断是否一致。

perm 表示此参数在 sysfs 文件系统中所对应的文件节点的属性。2.6 内核使用 sysfs 文件系统，这是一个建立在内存中比 proc 更强大的文件系统。sysfs 文件系统可以动态、实时，有组织层次地反应当前系统中的硬件、驱动等状态。当 perm 为 0 时，表示此参数不存在 sysfs 文件系统下对应的文件节点。 模块被加载后，在/sys/module/ 目录下将出现以此模块名命名的目录。如果此模块存在 perm 不为 0 的命令行参数，在此模块的目录下将出现 parameters 目录，包含一系列以参数名命名的文件节点，这些文件的权限值等于 perm，文件的内容为参数的值。

nump 为保存输入的数组元素个数的变量的指针。当不需保存实际输入的数组元素个数时，可以设为 NULL。从 2.6.0 至 2.6.10 版本，须将变量名赋给 nump；从 2.6.10 版本开始，须将变量的引用赋给 nump，这更易为开发人员理解。加载模块时，使用逗号分隔输入的数组元素。

适用于 2.4 与 2.6 内核的模块输入参数模板

```
#include <linux/module.h>
#ifdef LINUX26
#include <linux/moduleparam.h>
#endif
```

```
int debug = 0;
char *mode = "800x600";
int tuner[4] = {1, 1, 1, 1};
#ifdef LINUX26
int tuner_c = 1;
#endif
#ifdef LINUX26
MODULE_PARM(debug, "i");
MODULE_PARM(mode, "s");
MODULE_PARM(tuner,"1-4i");
#else
module_param(debug, int, 0644);
module_param(mode, charp, 0644);
#if LINUX_VERSION_CODE >= KERNEL_VERSION(2, 6, 10)
module_param_array(tuner, int, &tuner_c, 0644);
#else
module_param_array(tuner, int, tuner_c, 0644);
#endif
#endif
```

模块编译生成后，加载模块时可以输入：`modprobe my_module mode=1024x768 debug=1 tuner=22,33`。

在 linux/moduleparam.h 还定义有：

```
module_param_array_named(name, array, type, nump, perm)
module_param_call(name, set, get, arg, perm)
module_param_named(name, value, type, perm)
```

可以参阅 linux/moduleparam.h 查看这些宏的详细描述，有一点需注意，在 2.6 内核里，module_param 这一系列宏使用的都是小写名字。

1.2 sys_call_table 的获取

linux-2.6 以前，可以直接导出 sys_call_table，利用 sys_call_table[_NR_open]=(void*)func_ptr 改变系统 api 为自定义的 api，但从 2.6 之后，sys_call_table 便不能导出，所以必须用另一种方法找到 sys_call_table:

a.通过指令如SIDT得到中断描述符表寄存器IDTR。

b.从中断描述符表寄存器IDTR得到中断描述符表IDT的地址。

c.从中断描述符表IDT中得到system_call的地址。

d.由于在system_call中会调用system_call_table，所以可以遍历其指令得到system_call_table的地址。

e.直接将system_call_table中的系统调用重定向到相应的rootkit提供的系统调用。 最后修改系统调用表中相应系统调用函数为其他模块提供的修改版本的系统调用函数，实现系统调用的重定向。

2 王春萍

2.1 基础知识补习（先天知识不足，要花很多时间后天补齐，还是基础知识，好忧桑……）

2.1.1 Linux 下的终端控制及系统调用

Linux 用一个中断向量（128 或者 0x80）来实现系统调用，所有的系统调用都通过唯一的入口 system_call 来进入内核，当用户动态进程执行一条 int 0x80 汇编指令时，CPU 就切换到内核态，并开始执行 system_call 函数，system_call 函数再通过系统调用表 sys_call_table 来取得相应系统调用的地址进行执行。系统调用表 sys_call_table 中存放所有系统调用函数的地址，每个地址可以用系统调用号来进行索引，例如 sys_call_table[NR_fork]索引到的就是系统调用 sys_fork（）的地址。

所有的中断描述符存放在一片连续的地址空间中，这个连续的地址空间称作中断描述符表（IDT），其起始地址存放在中断描述符表寄存器（IDTR）中。

通过上面的说明可以得出通过 IDTR 寄存器来找到 system_call 函数地址的方法：根据 IDTR 寄存器找到中断描述符表，中断描述符表的第 0x80 项即是 system_call 函数的地址。

2.1.2 Linux 的 LKM（可装载内核模块）技术

在模块编程中必须存在初始化函数及清除函数，一般情况下，这两个函数默认为 init_module()以及 clearup_module()，初始化函数在模块被插入系统时调用，在其中可以进行一些函数及符号的注册工作，清除函数则在模块移除系统时进行调用，一些恢复工作通常在该函数中完成。

2.1.3 Linux 下的内存映像

/dev/kmem 是一个字符设备，是计算机主存的映像，通过它可以测试甚至修改系统，当内核不导出 sys_call_table 地址（Linux2.4.18 版以前的内核导出系统调用表（sys_call_table）的地址）或者不允许插入模块时可以通过该映像修改系统调用，从而实现隐藏文件、进程或者模块的目的。

2.2 文件隐藏（不得不说相关的源码好晦涩难懂，遇上春节，心定不下来，就更读不懂了，再次忧桑……⊙__⊙b……亲爱的自己，加油！！！）

getdents64()用 Linux dirent64 数据结构存放其信息。

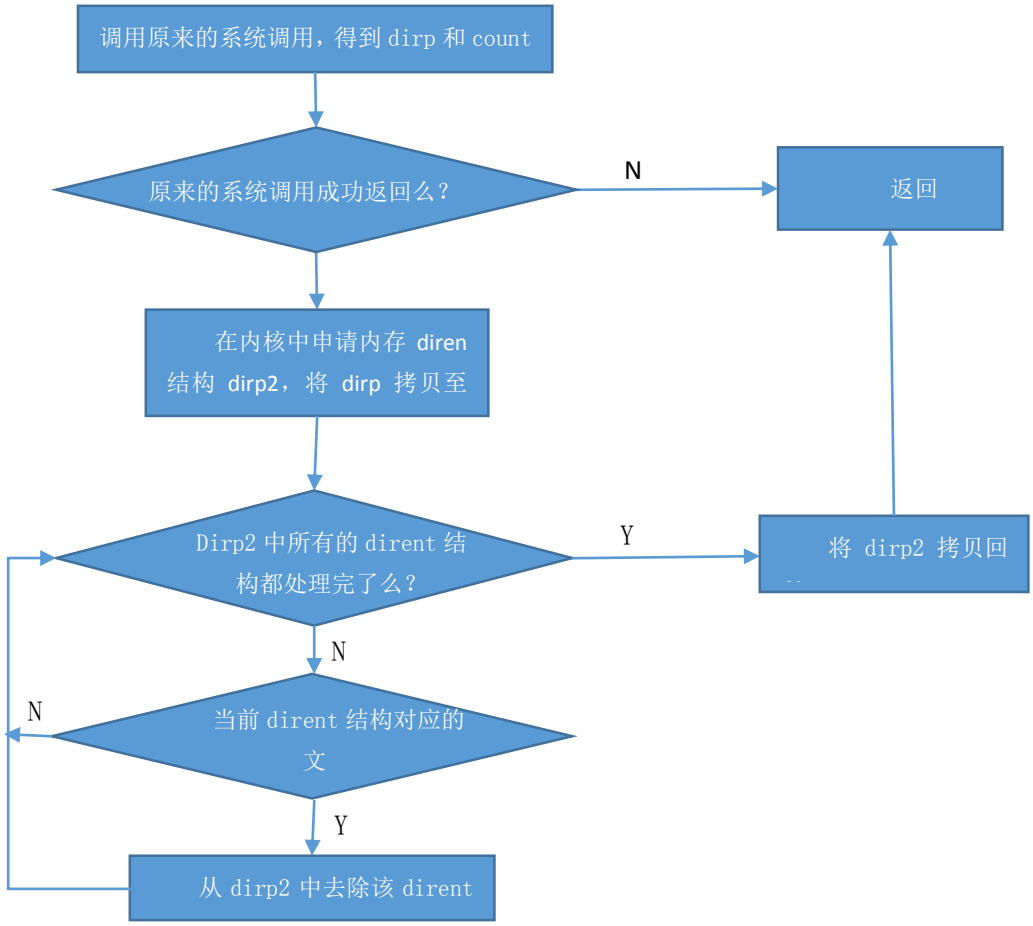
2.2.1 Linux dirent64 数据结构的定义

```
struct dirent64 {
  __u64       d_ino;       //文件或目录对应的 inode 号
   __s64       d_off;              //偏移距离
   unsigned short d_reclen;    //结构体大小
```

```
    unsigned char    d_type;
    char        d_name[256];      //文件或目录的文件名
};
```

2.2.2 实现文件隐藏步骤

就是先调用原来的系统调用，然后从得到的 dirent 结构中去除与特定文件名相关的文件信息，从而应用程序从该系统调用返回后将看不到该文件的存在。一些较新的版本中是通过 sys_getdents64 来查询文件信息的。



（1） 调用原来的 sys_getdents64()并将所得到的文件和目录信息读入用户传入的参数 dirp 所指向的缓存中，同时得到 count 值；

（2） 在内核中申请内存 dirent 结构 temp 做为临时处理区域；

（3） 把 dirp 所指向的内容拷贝到 temp 所指向的临时处理区域中；

（4） 判断并处理 temp 中与所要隐藏的文件和目录名称相匹配的内容；

（5） 将处理后的 temp 所指向的 dirent 结构内容拷贝回 dirp 所指向的结构中；

（6） 返回给原来的调用程序。

3 霍南风

隐藏及需要调用的 API!

1. 隐藏文件。像这些命令如"ls", "du" 使用 sys_getdents() 来获得目录信息。 所以 LKM 程序必须过滤这些输出来达到隐藏文件的目的。
2. 隐藏进程。在 Linux 的实现中,进程的信息被映射到/proc 文件系统去了。我们的工作仍旧是捕获 sys_getdents()调用在进程链表中标记为不可见。通常的手法是设置任务的信号标志位为一些未用的信号量，比如 31 就是一个例子。
3. 隐藏网络连接。 和隐藏进程相似， 在这个例子中我们是这去隐藏一些包括/proc/net/tcp 和/proc/net/udp 的文件。所以我们改变 sys_read()。 无论何时读包含匹配字符串的这两个文件的时候， 系统调用将不会声明在使用它。
4. 重定向可执行文件。 有时候， 入侵者可能会需要替换系统的二进制文件， 像"login", 但不想改变原文件。他可以截获 sys_execve()。因此， 无论何时系统尝试去执行"login" 程序的时候， 它都会被重定向到入侵者给定的其他程序。
5. 隐藏 sniffer。这儿我们指隐藏网络接口。在这里我们要替换的是 sys_ioctl()。
6. 隐藏 LKM 本身。一个优秀的 LKM 程序必须很好地隐藏它自己。系统里的 LKM 是用单向链表连接起来的， 为了隐藏 LKM 本身我们必须把它从链表中移走以至于 lsmod 这样的命令不能把它显示出来。
7. 隐藏符号表。通常的 LKM 中的函数将会被导出以至于其他模块可以使用它。因为我们是入侵者， 所以隐藏这些符号是必须的。幸运的是， 有一个宏可以供我们使用："EXPORT_NO_SYMBOLS"。 把这个宏放在 LKM 的最后可以防止任何符号的输出。

擦除日志需要擦除的日志

（1） logcheck

logchcek 可以自动地检查日志文件，定期检查日志文件以发现违反安全规则以及异常的活动。它先把正常的日志信息剔除掉，把一些有问题的日志保留下来，然后把这些信息 email 给系统管理员。logcheck 用 logtail 程序记住上次已经读过的日志文件的位置，然后从这个位置开始处理新的日志信息。logcheck 主要由下面几个主要的文件：logcheck.sh 可执行的脚本文件，记录 logcheck 检查那些日志文件等，我们可以把它加入 crontab 中定时运行。logcheck.hacking 是 logcheck 检查的模式文件。和下面的文件一起，按从上到下的顺序执行。这个文件表明了入侵活动的模式。logcheck.violations 这个文件表示有问题，违背常理的活动的模式。优先级小于上面的那个模式文件。logcheck.violations.ignore 这个文件和上面的 logcheck.violations 的优先是相对的，是我们所不关心的问题的模式文件。logcheck.ignore 这是检查的最后一个模式文件。如果没有和前三个模式文件匹配，也没有匹配这个模式文件的话，则输出到报告中。Logtail 记录日志文件信息。Logcheck 首次运行时读入相关的日志文件的所有内容，Logtail 会在日志文件的目录下为每个关心的日志文件建立一个 logfile.offset 的偏移量文件，以便于下次检查时从这个偏移量开始检查。Logcheck 执行时，将未被忽略的内容通过邮件的形式发送给 logcheck.sh 中 系统管理员指定的用户。

（2） logrotate

一般 Linux 发行版中都自带这个工具。它可以自动使日志循环，删除保存最久的日志，它的配置文件是 /etc/logrotate.conf，我们可以在这个文件中设置日志的循

环周期、日志的备份数目以及如何备份日志等等。在/etc /logrotate.d 目录下，包括一些工具的日志循环设置文件，如 syslog 等，在这些文件中指定了如何根据 /etc /logrotate.conf 做日志循环，也可以在这里面添加其他的文件以循环其他服务的日志。

（3） swatch

swatch 是一个实时的日志监控工具，我们可以设置感兴趣的事件。Swatch 有两种运行方式：一种可以在检查日志完毕退出，另一种可以连续监视日志中的新信息。Swatch 提供了许多通知方式，包括 email、振铃、终端输出、多种颜色等等。安装前，必须确保系统支持 perl。swatch 软件的重点是配置文件 swatchmessage，这个文本文件告诉 swatch 需要监视什么日志，需要寻找什么触发器，和当触发时所要执行的动作。当 swatch 发现与 swatchmessage 中定义的触发器正则表达式相符时，它将执行在 swatchrc 中定义的通知程序。

涉及的 linux API 详解：

1. int sys_getdents(unsigned int fd, struct dirent *dirp,unsigned int count)

其中 fd 为指向目录文件的文件描述符，该函数根据 fd 所指向的目录文件读取相应 dirent 结构，并放入 dirp 中，其中 count 为 dirp 中返回的数据量，正确时该函数返回值为填充到 dirp 的字节数。

附：dirent 结构：

```
struct dirent {
    long d_ino; /* inode number 索引节点号 */
    off_t d_off; /* offset to this dirent 在目录文件中的偏移 */
    unsigned short d_reclen; /* length of this d_name 文件名长 */
    unsigned char d_type; /* the type of d_name 文件类型 */
    char d_name [NAME_MAX+1]; /* file name (null-terminated) 文件名，最长 256 字符 */
}
```

linux 中的 off_t 类型默认是 32 位的 long int

2. sys_read()

```
asmlinkage ssize_t sys_read(unsigned int fd, char __user * buf, size_t count)
{
 struct file *file;
 ssize_t ret = -EBADF;
 int fput_needed;

 file = fget_light(fd, &fput_needed);
```

```
 if (file) {
 loff_t pos = file_pos_read(file);
 ret = vfs_read(file, buf, count, &pos);
 file_pos_write(file, pos);
 fput_light(file, fput_needed);
 }

 return ret;
}
```

fget_light() ：根据 fd 指定的索引，从当前进程描述符中取出相应的 file 对象

如果没找到指定的 file 对象，则返回错误

如果找到了指定的 file 对象：

调用 file_pos_read() 函数取出此次读写文件的当前位置。

调用 vfs_read() 执行文件读取操作，而这个函数最终调用 file->f_op.read() 指向的函数，代码如下：

```
if (file->f_op->read)

ret = file->f_op->read(file, buf, count, pos);
```

调用 file_pos_write() 更新文件的当前读写位置。

调用 fput_light() 更新文件的引用计数。

最后返回读取数据的字节数。

附：size_t 是标准 C 库中定义的，应为 unsigned int。定义为 typedef int ssize_t。

ssize_t:这个数据类型用来表示可以被执行读写操作的数据块的大小.它和 size_t 类似,但必需是 signed.意即：它表示的是 sign size_t 类型的。

3. sys_ioctl(fd,cmd,arg);

最后一个是传递额外数据的。

4 林坤

对 socket 的基本学习：

网络中进程之间如何通信？网络层的“ip 地址”可以唯一标识网络中的主机，而传输层的套接字，即“协议+端口”可以唯一标识主机中的应用程序（进程）。socket 起源于 Unix，而 Unix/Linux 基本哲学之一就是“一切皆文件”，都可以用“打开 open -> 读写 write/read -> 关闭 close”模式来操作。

socket 的基本操作：

1. socket()函数

int socket(int domain, int type, int protocol);

socket 函数对应于普通文件的打开操作。普通文件的打开操作返回一个文件描述字，而 socket()用于创建一个 socket 描述符（socket descriptor），它唯一标识一个 socket。这

个 socket 描述字跟文件描述字一样，后续的操作都有用到它，把它作为参数，通过它来进行一些读写操作。创建 socket 的时候，也可以指定不同的参数创建不同的 socket 描述符，socket 函数的三个参数分别为：domain，type，protocol。

2.bind()函数

bind()函数把一个地址族中的特定地址赋给 socket。例如对应 AF_INET、AF_INET6 就是把一个 ipv4 或 ipv6 地址和端口号组合赋给 socket。通常服务器在启动的时候都会绑定一个众所周知的地址（如 ip 地址+端口号），用于提供服务，客户就可以通过它来接连服务器；而客户端就不用指定，有系统自动分配一个端口号和自身的 ip 地址组合。这就是为什么通常服务器端在 listen 之前会调用 bind()，而客户端就不会调用，而是在 connect()时由系统随机生成一个。

而此处的进程号就会用到我们所学的网络的一些知识：

- **熟知端口** 范围从 0～1 023 的端口由 ICANN 指派和控制。这些叫做熟知端口。
- **注册端口** 范围从 1 024～49 151 的端口，IANA 不指派也不控制。它们只能在 IANA 注册以防止重复。
- **动态端口** 范围从 49 152～65 535 的端口既不用指派也不用注册。它们可以由任何进程来使用。这些可用作临时或私用端口号。最初的建议是客户使用的短暂端口号应当在这个范围之内选择。但是，大多数系统并没有遵循这个推荐。

熟知端口号小于 1024。

如果绑定的端口号是暂时的，最后从 49152~65535 直接选取。

3.listen()、connect()函数

如果作为一个服务器，在调用 socket()、bind()之后就会调用 listen()来监听这个 socket，如果客户端这时调用 connect()发出连接请求，服务器端就会接收到这个请求。

```
int listen(int sockfd, int backlog);
int connect(int sockfd, const struct sockaddr *addr, socklen_t addrlen);
```

listen 函数的第一个参数即为要监听的 socket 描述字，第二个参数为相应 socket 可以排队的最大连接个数。socket()函数创建的 socket 默认是一个主动类型的，listen 函数将 socket 变为被动类型的，等待客户的连接请求。

connect 函数的第一个参数即为客户端的 socket 描述字，第二参数为服务器的 socket 地址，第三个参数为 socket 地址的长度。客户端通过调用 connect 函数来建立与 TCP 服务器的连接。

4.accept()函数

TCP 服务器端依次调用 socket()、bind()、listen()之后，就会监听指定的 socket 地址了。TCP 客户端依次调用 socket()、connect()之后就想 TCP 服务器发送了一个连接请求。TCP 服务器监听到这个请求之后，就会调用 accept()函数取接收请求，这样连接就建立好了。之后就可以开始网络 I/O 操作了，即类同于普通文件的读写 I/O 操作。

5.read()、write()等函数

read 函数是负责从 fd 中读取内容.当读成功时，read 返回实际所读的字节数，如果返回的值是 0 表示已经读到文件的结束了，小于 0 表示出现了错误。如果错误为 EINTR 说明读

是由中断引起的，如果是 ECONNREST 表示网络连接出了问题。

write 函数将 buf 中的 nbytes 字节内容写入文件描述符 fd.成功时返回写的字节数。失败时返回-1，并设置 errno 变量。 在网络程序中，当我们向套接字文件描述符写时有俩种可能。1)write 的返回值大于 0，表示写了部分或者是全部的数据。2)返回的值小于 0，此时出现了 错误。我们要根据错误类型来处理。

6.close()函数

在服务器与客户端建立连接之后，会进行一些读写操作，完成了读写操作就要关闭相应的 socket 描述字，好比操作完打开的文件要调用 fclose 关闭打开的文件。

接下来的工作：

1.熟悉隐藏进程模块

2.熟悉隐藏连接模块