

Linux netfilter源码分析

内容基本上来自两篇文章:

《Netfilter源码分析》——(独孤九贱)

《Linux Netfilter实现机制和扩展技术》——(杨沙洲 国防科技大学计算机学院)

一、IP报文的接收到hook函数的调用

1.1 ip_input.c ip_rcv() 函数

以接收到的报文为例, 类似的还有ip_forward(ip_forward.c)和ip_output(ip_output.c)

```
int ip_rcv(struct sk_buff *skb, struct net_device *dev, struct packet_type
*pt, struct net_device *orig_dev)
{
    struct iphdr *iph;    //定义一个ip报文的数据报头
    u32 len;
    if (skb->pkt_type == PACKET_OTHERHOST)
        goto drop;    //数据包不是发给我们的
    IP_INC_STATS_BH(IPSTATS_MIB_INRECEIVES); //收到数据包统计量加1
    if ((skb = skb_share_check(skb, GFP_ATOMIC)) == NULL)
    {
        /* 如果数据报是共享的, 则复制一个出来, 此时复制而出的已经和socket脱离了关系
        */
        IP_INC_STATS_BH(IPSTATS_MIB_INDISCARDS);
        goto out;
    }
    if (!pskb_may_pull(skb, sizeof(struct iphdr)))
        goto inhdr_error;    //对数据报的头长度进行检查,

    iph = skb->nh.iph;    //取得数据报的头部位置
    if (iph->ihl < 5 || iph->version != 4)    //版本号或者头长度不对,
        goto inhdr_error;    //头长度是以4字节为单位的, 所以5表示的是20字节
    if (!pskb_may_pull(skb, iph->ihl*4))
        goto inhdr_error;

    if (unlikely(ip_fast_csum((u8 *)iph, iph->ihl)))
        goto inhdr_error;    //检查报文的检验和字段
    len = ntohs(iph->tot_len);
    if (skb->len < len || len < (iph->ihl*4))
        goto inhdr_error;    //整个报文长度不可能比报头长度小
    if (pskb_trim_rsum(skb, len))
    {    //对数据报进行裁减, 这样可以分片发送过来的数据报不会有重复数据
        IP_INC_STATS_BH(IPSTATS_MIB_INDISCARDS);
        goto drop;
    }

    return NF_HOOK(PF_INET, NF_IP_PRE_ROUTING, skb, dev, NULL,
        ip_rcv_finish);    //通过回调函数调用ip_rcv_finish

inhdr_error:
    IP_INC_STATS_BH(IPSTATS_MIB_INHDRERRORS);
drop:
    kfree_skb(skb);    //丢掉数据报
out:
    return NET_RX_DROP;
}
```

1.2 include/linux/netfilter.h NF_HOOK宏

```

#ifdef CONFIG_NETFILTER_DEBUG
#define NF_HOOK(pf, hook, skb, indev, outdev, okfn)
\
    nf_hook_slow((pf), (hook), (skb), (indev), (outdev), (okfn), INT_MIN)
#define NF_HOOK_THRESH nf_hook_slow
#else
#define NF_HOOK(pf, hook, skb, indev, outdev, okfn)
\
    (list_empty(&nf_hooks[(pf)][(hook)]))
\
    ? (okfn)(skb)
    : nf_hook_slow((pf), (hook), (skb), (indev), (outdev), (okfn), INT_MIN)
#define NF_HOOK_THRESH(pf, hook, skb, indev, outdev, okfn, thresh)
\
    (list_empty(&nf_hooks[(pf)][(hook)]))
\
    ? (okfn)(skb)
    : nf_hook_slow((pf), (hook), (skb), (indev), (outdev), (okfn), (thresh)))
#endif

```

/* 如果nf_hooks[PF_INET][NF_IP_FORWARD]所指向的链表为空（即该钩子上没有挂处理函数），则直接调用okfn；否则，则调用net/core/netfilter.c::nf_hook_slow()转入Netfilter的处理。 */

1.3 net/core/netfilter.c nf_kook_slow() 函数

```

int nf_hook_slow(int pf, unsigned int hook, struct sk_buff **pskb,
                 struct net_device *indev,
                 struct net_device *outdev,
                 int (*okfn)(struct sk_buff *),
                 int hook_thresh)
{
    struct list_head *elem;
    unsigned int verdict;
    int ret = 0;
    rcu_read_lock();
    /*取得对应的链表首部*/
    elem = &nf_hooks[pf][hook];
next_hook:
    /*调用对应的钩子函数*/
    verdict = nf_iterate(&nf_hooks[pf][hook], pskb, hook, indev,
                        outdev, &elem, okfn, hook_thresh);
    /*判断返回值，做相应的处理*/
    if (verdict == NF_ACCEPT || verdict == NF_STOP) {
        ret = 1; /*前面提到过，返回1，则表示装继续调用okfn函数指针*/
        goto unlock;
    } else if (verdict == NF_DROP) {
        kfree_skb(*pskb); /*删除数据包，需要释放skb*/
        ret = -EPERM;
    } else if (verdict == NF_QUEUE) {
        NFDEBUG("nf_hook: Verdict = QUEUE.\n");
        if (!nf_queue(*pskb, elem, pf, hook, indev, outdev, okfn))
            goto next_hook;
    }
unlock:
    rcu_read_unlock();
}

```

```

        return ret;
    }

```

1.4 net/core/netfilter.c nf_iterate() 函数

```

static unsigned int nf_iterate(struct list_head *head,
                               struct sk_buff **skb,
                               int hook,
                               const struct net_device *indev,
                               const struct net_device *outdev,
                               struct list_head **i,
                               int (*okfn)(struct sk_buff *),
                               int hook_thresh)
{
    /*
     * The caller must not block between calls to this
     * function because of risk of continuing from deleted element.
     */

```

/* 依次调用指定hook点下的所有nf_hook_ops->(*hook)函数，这些nf_hook_ops里有filter表注册的，有mangle表注册的，等等。

list_for_each_continue_rcu函数是一个for循环的宏，当调用结点中的hook函数后，根据返回值进行相应处理。如果hook函数的返回值是NF_QUEUE, NF_STOLEN, NF_DROP时，函数返回该值；如果返回值是NF_REPEAT时，则跳到前一个结点继续处理；如果是其他值，由下一个结点继续处理。如果整条链表处理完毕，返回值不是上面四个值，则返回NF_ACCEPT。*/

```

list_for_each_continue_rcu(*i, head)
{
    struct nf_hook_ops *elem = (struct nf_hook_ops *)*i;
    if (hook_thresh > elem->priority)
        continue;
    switch (elem->hook(hook, skb, indev, outdev, okfn))
    {
        case NF_QUEUE:
            return NF_QUEUE;

        case NF_STOLEN:
            return NF_STOLEN;

        case NF_DROP:
            return NF_DROP;

        case NF_REPEAT:
            *i = (*i)->prev;
            break;
    }
}
return NF_ACCEPT;
}

```

二、ipt_table数据结构和表的初始化

2.1 include/linux/netfilter_ipv4/ip_tables.h struct ipt_table 表结构

```

struct ipt_table
{

```

```

struct list_head list;      /* 表链 */
char name[IPT_TABLE_MAXNAMELEN]; /* 表名，如“filter”、“nat”等，为了满足自动
模块加载的设计，包含该表的模块应命名为iptables_’name’.o */
struct ipt_replace *table;  /* 表模子，初始为initial_table.repl */
unsigned int valid_hooks;   /* 位向量，标示本表所影响的HOOK */
rwlock_t lock;             /* 读写锁，初始为打开状态 */
struct ipt_table_info *private; /* iptable的数据区，见下 */
struct module *me;         /* 是否在模块中定义 */
};

```

2.2 struct ipt_table_info是实际描述表的数据结构 iptables.c

```

struct ipt_table_info
{
    unsigned int size;      /* 表大小 */
    unsigned int number;    /* 表中的规则数 */
    unsigned int initial_entries; /* 初始的规则数，用于模块计数 */
    unsigned int hook_entry[NF_IP_NUMHOOKS]; /* 记录所影响的HOOK的规则入口相对于
于下面的entries变量的偏移量 */
    unsigned int underflow[NF_IP_NUMHOOKS]; /* 与hook_entry相对应的规则表上限
偏移量，当无规则录入时，相应的
hook_entry和underflow均为0 */
    char entries[0] ____cacheline_aligned; /* 规则表入口 */
};

```

2.3 include/linux/netfilter_ipv4 规则用struct ipt_entry结构表示，包含匹配用的IP头部分、一个Target和0个或多个Match。由于Match数不定，所以一条规则实际的占用空间是可变的。结构定义如下

```

struct ipt_entry
{
    struct ipt_ip ip;      /* 所要匹配的报文的IP头信息 */
    unsigned int nfcache;  /* 位向量，标示本规则关心报文的什么部分，暂未使用 */
    u_int16_t target_offset; /* target区的偏移,通常target区位于match区之后,而
match区则在ipt_entry的末尾;初始化为sizeof(struct
ipt_entry)，即假定没有match */
    u_int16_t next_offset; /* 下一条规则相对于本规则的偏移，也即本规则所用空间的
总和，初始化为sizeof(struct ipt_entry)
+sizeof(struct ipt_target)，即没有match */
    unsigned int comefrom; /* 规则返回点，标记调用本规则的HOOK号，可用于检查
规则的有效性 */
    struct ipt_counters counters; /* 记录该规则处理过的报文数和报文总字节数 */
    unsigned char elems[0];      /* target或者是match的起始位置 */
};

```

4 iptables的初始化init(void)，以filter表为例 iptable_filter.c

```

static int __init init(void)
{
    int ret;
    if (forward < 0 || forward > NF_MAX_VERDICT) {
        printk("iptables forward must be 0 or 1\n");
        return -EINVAL;
    }
}

```

```

/* Entry 1 is the FORWARD hook */
initial_table.entries[1].target.verdict = -forward - 1;

/* Register table */
ret = ipt_register_table(&packet_filter);    //注册filter表
if (ret < 0)
    return ret;

/* Register hooks */
ret = nf_register_hook(&ipt_ops[0]);        //注册三个HOOK
if (ret < 0)
    goto cleanup_table;

ret = nf_register_hook(&ipt_ops[1]);
if (ret < 0)
    goto cleanup_hook0;

ret = nf_register_hook(&ipt_ops[2]);
if (ret < 0)
    goto cleanup_hook1;

return ret;

cleanup_hook1:
    nf_unregister_hook(&ipt_ops[1]);
cleanup_hook0:
    nf_unregister_hook(&ipt_ops[0]);
cleanup_table:
    ipt_unregister_table(&packet_filter);

return ret;
}

/* ipt_register_table函数的参数packet_filter包含了待注册表的各个参数 */
static struct ipt_table packet_filter = {
    .name      = "filter",
    .table      = &initial_table.repl,
    .valid_hooks = FILTER_VALID_HOOKS,
    .lock       = RW_LOCK_UNLOCKED,
    .me        = THIS_MODULE
};

```

/* 上面的&initial_table.repl是一个ipt_replace结构，也就是ipt_table->*table的初始值。

下面是ipt_replace结构的定义，它和ipt_table_info很相似，基本上就是用来初始化ipt_table中的ipt_table_info *private的，这个结构不同于ipt_table_info之处在于，它还要保存表的旧的规则信息 */

```

struct ipt_replace
{
    char name[IPT_TABLE_MAXNAMELEN];    /* 表名 */
    unsigned int valid_hooks;            /* 影响的hook */
    unsigned int num_entries;            /* entry数 */
    unsigned int size;                   /* entry的总大小 */
    unsigned int hook_entry[NF_IP_NUMHOOKS]; /* 规则入口的偏移值 */
    unsigned int underflow[NF_IP_NUMHOOKS]; /* 规则的最大偏移值 */
    unsigned int num_counters;           /* 规则数 */
    struct ipt_counters __user *counters;
};

```

```

    struct ipt_entry entries[0];                                /* 规则入口 */
};

/* 下面是initial_table.repl的初始化 */
static struct
{
    struct ipt_replace repl;
    struct ipt_standard entries[3];
    struct ipt_error term;
} initial_table __initdata
= { { "filter", FILTER_VALID_HOOKS, 4,
    sizeof(struct ipt_standard) * 3 + sizeof(struct ipt_error),
    { [NF_IP_LOCAL_IN] = 0,
[NF_IP_FORWARD] = sizeof(struct ipt_standard),
[NF_IP_LOCAL_OUT] = sizeof(struct ipt_standard) * 2 },
    { [NF_IP_LOCAL_IN] = 0,
[NF_IP_FORWARD] = sizeof(struct ipt_standard),
[NF_IP_LOCAL_OUT] = sizeof(struct ipt_standard) * 2 },
    0, NULL, { } },
    {
        /* LOCAL_IN */
        { { { { 0 }, { 0 }, { 0 }, { 0 }, "", "", { 0 }, { 0 }, 0, 0, 0 },
0,
sizeof(struct ipt_entry),
sizeof(struct ipt_standard),
0, { 0, 0 }, { } },
        { { { { IPT_ALIGN(sizeof(struct ipt_standard_target)), "" } }, {
} },
        -NF_ACCEPT - 1 } },
        /* FORWARD */
        { { { { 0 }, { 0 }, { 0 }, { 0 }, "", "", { 0 }, { 0 }, 0, 0, 0 },
0,
sizeof(struct ipt_entry),
sizeof(struct ipt_standard),
0, { 0, 0 }, { } },
        { { { { IPT_ALIGN(sizeof(struct ipt_standard_target)), "" } }, {
} },
        -NF_ACCEPT - 1 } },
        /* LOCAL_OUT */
        { { { { 0 }, { 0 }, { 0 }, { 0 }, "", "", { 0 }, { 0 }, 0, 0, 0 },
0,
sizeof(struct ipt_entry),
sizeof(struct ipt_standard),
0, { 0, 0 }, { } },
        { { { { IPT_ALIGN(sizeof(struct ipt_standard_target)), "" } }, {
} },
        -NF_ACCEPT - 1 } }
    },
    /* ERROR */
    { { { { 0 }, { 0 }, { 0 }, { 0 }, "", "", { 0 }, { 0 }, 0, 0, 0 },
0,
sizeof(struct ipt_entry),
sizeof(struct ipt_error),
0, { 0, 0 }, { } },
        { { { { IPT_ALIGN(sizeof(struct ipt_error_target)), IPT_ERROR_TARGET
} },
        { } },
        "ERROR"
    }
};

```

```

    }
}
};

```

三、ipt_table表的注册

init（）函数初始化时调用了ipt_register_table函数进行表的注册

3.1 ip_tables.c 表的注册 ipt_register_table

```

int ipt_register_table(struct ipt_table *table)
{
    int ret;
    struct ipt_table_info *newinfo;
    static struct ipt_table_info bootstrap
        = { 0, 0, 0, { 0 }, { 0 }, { } };
    /*宏MOD_INC_USE_COUNT用于模块计数器累加，主要是为了防止模块异常删除，对应的宏MOD_DEC_USE_COUNT就是累减了*/
    MOD_INC_USE_COUNT;

    /*为每个CPU分配规则空间*/
    newinfo = vmalloc(sizeof(struct ipt_table_info)
        + SMP_ALIGN(table->table->size) * smp_num_cpus);
    if (!newinfo) {
        ret = -ENOMEM;
        MOD_DEC_USE_COUNT;
        return ret;
    }

    /*将规则项拷贝到新表项的第一个cpu空间里面*/
    memcpy(newinfo->entries, table->table->entries, table->table->size);

    /*translate_table函数将newinfo表示的table的各个规则进行边界检查，然后对于newinfo所指的ipt_table_info结构中的hook_entries和underflows赋予正确的值，最后将表项向其他cpu拷贝*/
    ret = translate_table(table->name, table->valid_hooks,
        newinfo, table->table->size,
        table->table->num_entries,
        table->table->hook_entry,
        table->table->underflow);
    if (ret != 0) {
        vfree(newinfo);
        MOD_DEC_USE_COUNT;
        return ret;
    }

    ret = down_interruptible(&ipt_mutex);
    if (ret != 0) {
        vfree(newinfo);
        MOD_DEC_USE_COUNT;
        return ret;
    }

    /* 如果注册的table已经存在，释放空间 并且递减模块计数 */
    /* Don't autoload: we'd eat our tail... */
    if (list_named_find(&ipt_tables, table->name)) {

```

```

        ret = -EEXIST;
        goto free_unlock;
    }

    /* 替换table项. */
    /* Simplifies replace_table code. */
    table->private = &bootstrap;
    if (!replace_table(table, 0, newinfo, &ret))
        goto free_unlock;

    duprintf("table->private->number = %u\n", table->private->number);

    /* 保存初始规则计数器 */
    /* save number of initial entries */
    table->private->initial_entries = table->private->number;

    table->lock = RW_LOCK_UNLOCKED;
    /*将表添加进链表*/
    list_prepend(&ipt_tables, table);

unlock:
    up(&ipt_mutex);
    return ret;

free_unlock:
    vfree(newinfo);
    MOD_DEC_USE_COUNT;
    goto unlock;
}

```

3.2 ip_tables.c translate_table()函数

```

/* 函数:translate_table()
* 参数:
*   name:表名称;
*   valid_hooks: 当前表所影响的hook
*   newinfo: 包含当前表的所有信息的结构
*   size: 表的大小
*   number: 表中的规则数
*   hook_entries: 记录所影响的HOOK的规则入口相对于下面的entries变量的偏移量
*   underflows: 与hook_entry相对应的规则表上限偏移量
* 作用:
*   translate_table函数将newinfo表示的table的各个规则进行边界检查, 然后对于
newinfo所指的ipt_table_info结构中的hook_entries和underflows赋予正确的值, 最后
将表项向其他cpu拷贝
* 返回值:
*   int ret==0表示成功返回
*/

```

```

static int
translate_table(const char *name,
               unsigned int valid_hooks,
               struct ipt_table_info *newinfo,
               unsigned int size,

```



```

        unsigned int number,
        const unsigned int *hook_entries,
        const unsigned int *underflows)
{
    unsigned int i;
    int ret;

    newinfo->size = size;
    newinfo->number = number;

    /* 初始化所有Hooks为不可能的值. */
    for (i = 0; i < NF_IP_NUMHOOKS; i++) {
        newinfo->hook_entry[i] = 0xFFFFFFFF;
        newinfo->underflow[i] = 0xFFFFFFFF;
    }

    duprintf("translate_table: size %u\n", newinfo->size);
    i = 0;
    /* 遍历所有规则，检查所有偏量，检查的工作都是由IPT_ENTRY_ITERATE这个
    宏来完成，并且它的最后一个参数i，返回表的所有规则数. */
    ret = IPT_ENTRY_ITERATE(newinfo->entries, newinfo->size,
        check_entry_size_and_hooks,
        newinfo,
        newinfo->entries,
        newinfo->entries + size,
        hook_entries, underflows, &i);

    if (ret != 0)
        return ret;

    /*实际计算得到的规则数与指定的不符*/
    if (i != number) {
        duprintf("translate_table: %u not %u entries\n",
            i, number);
        return -EINVAL;
    }

    /* 因为函数一开始将HOOK的偏移地址全部初始成了不可能的值，而在上一个宏的
    遍历中设置了hook_entries和underflows的值，这里对它们进行检查 */
    for (i = 0; i < NF_IP_NUMHOOKS; i++) {
        /* 只检查当前表所影响的hook */
        if (!(valid_hooks & (1 << i)))
            continue;
        if (newinfo->hook_entry[i] == 0xFFFFFFFF) {
            duprintf("Invalid hook entry %u %u\n",
                i, hook_entries[i]);
            return -EINVAL;
        }
        if (newinfo->underflow[i] == 0xFFFFFFFF) {
            duprintf("Invalid underflow %u %u\n",
                i, underflows[i]);
            return -EINVAL;
        }
    }
}

```

```

/*确保新的table中不存在规则环*/
if (!mark_source_chains(newinfo, valid_hooks))
    return -ELOOP;

/* 对tables中的规则项进行完整性检查，保证每一个规则项在形式上是合法的*/
i = 0;
ret = IPT_ENTRY_ITERATE(newinfo->entries, newinfo->size,
                        check_entry, name, size, &i);

/*检查失败，释放空间，返回*/
if (ret != 0) {
    IPT_ENTRY_ITERATE(newinfo->entries, newinfo->size,
                    cleanup_entry, &i);
    return ret;
}

/* 为每个CPU复制一个完整的table项*/
for (i = 1; i < smp_num_cpus; i++) {
    memcpy(newinfo->entries + SMP_ALIGN(newinfo->size)*i,
        newinfo->entries,
        SMP_ALIGN(newinfo->size));
}

return ret;
}

```

3.3 IPT_ENTRY_ITERATE宏 ip_tables.h

用来遍历每一个规则，然后调用其第三个参数（函数指针）进行处理，前两个参数分别表示规则的起始位置和规则总大小，后面的参数则视情况而定。

```

#define IPT_ENTRY_ITERATE(entries, size, fn, args...)
\
({
\
    unsigned int __i;
\
    int __ret = 0;
\
    struct ipt_entry *__entry;
\
    for (__i = 0; __i < (size); __i += __entry->next_offset) { \
        __entry = (void *) (entries) + __i;
\
        __ret = fn(__entry, ## args);
\
        if (__ret != 0)
            break;
\
    }
\
    __ret;
\
})

```

```

/* translate_table中出现了三次，分别是 */
IPT_ENTRY_ITERATE(newinfo->entries, newinfo->size,
                  check_entry_size_and_hooks,
                  newinfo,
                  newinfo->entries,
                  newinfo->entries + size,
                  hook_entries, underflows, &i);

```

```

IPT_ENTRY_ITERATE(newinfo->entries, newinfo->size,
                  check_entry, name, size, &i);
IPT_ENTRY_ITERATE(newinfo->entries, newinfo->size,
                  cleanup_entry, &i);

```

即是在遍历到每条entry时分别调用

check_entry_size_and_hooks, check_entry, cleanup_entry,三个函数
check_entry有大用处，后面解释

3.4 list_named_find() 函数 listhelp.h

在注册函数中，调用

list_named_find(&ipt_tables, table->name)

来检查当前表是否已被注册过了。可见，第一个参数为链表首部，第二个参数为当前表名。

其原型如下：

```

#define list_named_find(head, name) \
LIST_FIND(head, __list_cmp_name, void *, name)

#define LIST_FIND(head, cmpfn, type, args...) \
({ \
    const struct list_head *__i = (head); \
    ASSERT_READ_LOCK(head); \
    do { \
        __i = __i->next; \
        if (__i == (head)) { \
            __i = NULL; \
            break; \
        } \
    } while (!cmpfn((const type)__i, ## args)); \
    (type)__i; \
})

```

前面提过，表是一个双向链表，在宏当中，以while进行循环，以__i = __i->next;进行遍历，然后调用比较函数进行比较，传递过来的比较函数是__list_cmp_name。

比较函数很简单：

```

static inline int __list_cmp_name(const void *i, const char *name)
{
    return strcmp(name, i+sizeof(struct list_head)) == 0;
}

```

3.5 replace_table() 函数 ip_tables.c

表中以struct ipt_table_info *private;表示实际数据区。但是在初始化赋值的时候，被设为NULL，而表的初始变量都以模版的形式，放在struct ipt_replace *table;中。

注册函数一开始，就声明了：struct ipt_table_info *newinfo;然后对其分配了空间，将模块中的初值拷贝了进来。所以replace_table要做的工作，主要就是把newinfo中的值传递给table结构中的private成员。

```

replace_table(struct ipt_table *table,
              unsigned int num_counters,
              struct ipt_table_info *newinfo,
              int *error)

```

```

{
    struct ipt_table_info *oldinfo;

    write_lock_bh(&table->lock);

    if (num_counters != table->private->number) {
        duprintf("num_counters != table->private->number (%u/%u)\n",
            num_counters, table->private->number);
        /* ipt_register_table函数中, replace_table函数之前有一句 table->private =
        &bootstrap;将private初始化为bootstrap, 即{ 0, 0, 0, {0}, {0}, {} */
        write_unlock_bh(&table->lock);
        *error = -EAGAIN;
        return NULL;
    }
    oldinfo = table->private;
    table->private = newinfo;
    newinfo->initial_entries = oldinfo->initial_entries;
    write_unlock_bh(&table->lock);

    return oldinfo;
}

```

3.6 list_prepend() 函数 listhelp.h

当所有的初始化工作结束, 就调用**list_prepend**来构建链表了。

```

static inline void
list_prepend(struct list_head *head, void *new)
{
    ASSERT_WRITE_LOCK(head);          /*设置写互斥*/
    list_add(new, head);              /*将当前表节点添加进链表*/
}

```

list_add就是一个构建双向链表的过程:

```

static __inline__ void list_add(struct list_head *new, struct list_head *head)
{
    __list_add(new, head, head->next);
}

```

```

static __inline__ void __list_add(struct list_head * new,
    struct list_head * prev,
    struct list_head * next)
{
    next->prev = new;
    new->next = next;
    new->prev = prev;
    prev->next = new;
}

```

四、nf_hook_ops 钩子的注册

在filter表的初始化函数static int __init init(void)中除了有一个nf_register_hook函数注册一个tables外, 还由nf_register_hook函数注册了3个hook

4.1 **nf_hook_ops**数据结构 **netfilter.h**

```
struct nf_hook_ops
{
    struct list_head list;           //链表成员
    /* User fills in from here down. */
    nf_hookfn *hook;                //钩子函数指针
    struct module *owner;
    int pf;                          //协议簇，对于ipv4而言，是PF_INET
    int hooknum;                     //hook类型
    /* Hooks are ordered in ascending priority. */
    int priority;                    //优先级
};
```

list成员用于维护Netfilter hook的列表。

hook成员是一个指向**nf_hookfn**类型的函数的指针，该函数是这个**hook**被调用时执行的函数。**nf_hookfn**同样在**linux/netfilter.h**中定义。

pf这个成员用于指定协议族。有效的协议族在**linux/socket.h**中列出，但对于IPv4我们使用协议族**PF_INET**。

hooknum这个成员用于指定安装的这个函数对应的具体的**hook**类型：

NF_IP_PRE_ROUTING	在完整性校验之后，选路确定之前
NF_IP_LOCAL_IN	在选路确定之后，且数据包的目的是本地主机
NF_IP_FORWARD	目的地是其它主机地数据包
NF_IP_LOCAL_OUT	来自本机进程的数据包在其离开本地主机的过程中
NF_IP_POST_ROUTING	在数据包离开本地主机“上线”之前

C:\Documents and Settings\Administrator\桌面\Netfilter\image006

再看看它的初始化，仍以filter表为例

```
static struct nf_hook_ops ipt_ops[]
= { { { NULL, NULL }, ipt_hook, PF_INET, NF_IP_LOCAL_IN,
NF_IP_PRI_FILTER },
    { { NULL, NULL }, ipt_hook, PF_INET, NF_IP_FORWARD,
NF_IP_PRI_FILTER },
    { { NULL, NULL }, ipt_local_out_hook, PF_INET, NF_IP_LOCAL_OUT,
NF_IP_PRI_FILTER }
};
```

4.2 **int nf_register_hook**函数 **netfilter.c**

注册实际上就是在一个**nf_hook_ops**链表中再插入一个**nf_hook_ops**结构

```
int nf_register_hook(struct nf_hook_ops *reg)
{
    struct list_head *i;

    spin_lock_bh(&nf_hook_lock);
    list_for_each(i, &nf_hooks[reg->pf][reg->hooknum]) {
```

```

        if (reg->priority < ((struct nf_hook_ops *)i)->priority)
            break;
    }
    list_add_rcu(&reg->list, i->prev);
    spin_unlock_bh(&nf_hook_lock);

    synchronize_net();
    return 0;
}

```

list_for_each 函数遍历当前待注册的钩子的协议pf及Hook类型所对应的链表，其首地址是&nf_hooks[reg->pf][reg->hooknum]，如果当前待注册钩子的优先级小于匹配的节点的优先级，则找到了待插入的位置，也就是说，按优先级的升序排列。

list_add_rcu把当前节点插入到查找到适合的的位置，这样，完成后，所有pf协议下的hooknum类型的钩子，都被注册到&nf_hooks[reg->pf][reg->hooknum]为首的链表当中了。

4.3 ipt_hook钩子函数 iptable_raw.c

注册nf_hook_ops，也就向内核注册了一个钩子函数，这些函数有ipt_hook，ipt_local_hook，ipt_route_hook，ipt_local_out_hook等。

前面在nf_iterate()里调用的钩子函数就是它了

下面是ipt_hook函数的定义：

```

static unsigned int
ipt_hook(unsigned int hook,          /* hook点 */
        struct sk_buff **pskb,
        const struct net_device *in,
        const struct net_device *out,
        int (*okfn)(struct sk_buff *)) /* 默认处理函数 */
{

```

/* 参数&packet_filter是由注册该nf_hook_ops的表（filter）决定的，也有可能是&packet_raw */

```

    return ipt_do_table(pskb, hook, in, out, &packet_filter, NULL);
}

```

实际上是直接调用ipt_do_table(ip_tables.c)函数

接下来就是根据table里面的entry来处理数据包了

一个table就是一组防火墙规则的集合

而一个entry就是一条规则，每个entry由一系列的matches和一个target组成

一旦数据包匹配了该某个entry的所有matches，就用target来处理它

Match又分为两部份，一部份为一些基本的元素，如来源/目的地址，进/出网口，协议等，对应了struct ipt_ip，我们常常将其称为标准的match，另一部份match则以插件的形式存在，是动态可选择，也允许第三方开发的，常常称为扩展的match，如字符串匹配，p2p匹配等。同样，规则的target也是可扩展的。这样，一条规则占用的空间，可以分为：struct ipt_ip+n*match+n*target，（n表示了其个数，这里的match指的是可扩展的match部份）。

五、 ipt_do_table()函数，数据包的过滤

5.1 ipt_entry 相关结构 ip_tables.h

ipt_entry结构前面有过了，再看一遍

```

    struct ipt_entry
    {
        struct ipt_ip ip;          /* 所要匹配的报文的IP头信息 */
        unsigned int nfcache;      /* 位向量，标示本规则关心报文的什么部分，暂未使用 */
        u_int16_t target_offset; /* target区的偏移，通常target区位于match区之后，而match区则在ipt_entry的末尾；初始化为sizeof(struct ipt_entry)，即假定没有match */
        u_int16_t next_offset; /* 下一条规则相对于本规则的偏移，也即本规则所用空间的总和，初始化为sizeof(struct ipt_entry)+sizeof(struct ipt_target)，即没有match */
        unsigned int comefrom;     /* 位向量，标记调用本规则的HOOK号，可用于检查规则的有效性 */
        struct ipt_counters counters; /* 记录该规则处理过的报文数和报文总字节数 */
        unsigned char elems[0];    /* target或者是match的起始位置 */
    }

```

ipt_ip结构 ip_tables.h

```

struct ipt_ip {
    struct in_addr src, dst;          /* 来源/目的地址 */
    struct in_addr smask, dmask;      /* 来源/目的地址的掩码 */
    char iniface[IFNAMSIZ], outiface[IFNAMSIZ]; /* 输入输出网络接口 */
    unsigned char iniface_mask[IFNAMSIZ], outiface_mask[IFNAMSIZ];
    u_int16_t proto; /* 协议，0 = ANY */
    u_int8_t flags; /* 标志字段 */
    u_int8_t invflags; /* 取反标志 */
};

```

5.2 ipt_do_table函数 ip_tables.c

```

unsigned int
ipt_do_table(struct sk_buff **pskb,
             unsigned int hook,
             const struct net_device *in,
             const struct net_device *out,
             struct ipt_table *table,
             void *userdata)
{
    static const char nulldevname[IFNAMSIZ] = \
        __attribute__((aligned(sizeof(long))));
    u_int16_t offset;
    struct iphdr *ip;
    u_int16_t datalen;
    int hotdrop = 0;
    /* Initializing verdict to NF_DROP keeps gcc happy. */
    unsigned int verdict = NF_DROP;
    const char *indev, *outdev;
    void *table_base;
    struct ipt_entry *e, *back;

    /* Initialization */
    ip = (*pskb)->nh.iph; /* 获取IP头 */
    datalen = (*pskb)->len - ip->ihl * 4; /* 指向数据区 */
    indev = in ? in->name : nulldevname; /* 取得输入设备名 */
    outdev = out ? out->name : nulldevname; /* 取得输出设备名 */
    offset = ntohs(ip->frag_off) & IP_OFFSET; /* 设置分片包的偏移 */

```

```

        read_lock_bh(&table->lock);        /*设置互斥锁*/
        IP_NF_ASSERT(table->valid_hooks & (1 << hook));
/*检验HOOK, debug用的*/
/*获取当前表的当前CPU的规则入口*/
        table_base = (void *)table->private->entries
            + TABLE_OFFSET(table->private, smp_processor_id());
/*获得当前表的当前Hook的规则的开始偏移量*/
        e = get_entry(table_base, table->private->hook_entry[hook]);

/*获得当前表的当前Hook的规则的上限偏移量*/
/* For return from builtin chain */
        back = get_entry(table_base, table->private->underflow[hook]);
/* do ..... while (! hotdrop)
   进行规则的匹配 */
        do {
            IP_NF_ASSERT(e);
            IP_NF_ASSERT(back);
            (*pskb)->nfcache |= e->nfcache;

/*
   匹配IP包, 成功则继续匹配下去, 否则跳到下一个规则
   ip_packet_match匹配标准match, 也就是ip报文中的一些基本的元素, 如来源/目的
   的地址, 进/出网口, 协议等, 因为要匹配的内容是固定的, 所以具体的函数实现也是固定
   的。
   而IPT_MATCH_ITERATE (应该猜到实际是调用第二个参数do_match函数) 匹配扩展
   的match, 如字符串匹配, p2p匹配等, 因为要匹配的内容不确定, 所以函数的实现也是不
   一样的, 所以do_match的实现就和具体的match模块有关了。
   这里的&e->ip就是上面的ipt_ip结构
*/
            if (ip_packet_match(ip, indev, outdev, &e->ip, offset)) {
                struct ipt_entry_target *t;

                if (IPT_MATCH_ITERATE(e, do_match,
                    *pskb, in, out,
                    offset, &hotdrop) != 0)
                    goto no_match;        /*不匹配则跳到 no_match, 往下一个规则*/

/* 匹配则继续执行 */
/* 这个宏用来分别处理字节计数器和分组计数器这两个计数器 */
                ADD_COUNTER(e->counters, ntohs(ip->tot_len), 1);

/*获取规则的target的偏移地址*/
                t = ipt_get_target(e);
                IP_NF_ASSERT(t->u.kernel.target);

/* 下面开始匹配target */
/* Standard target? */
                if (!t->u.kernel.target->target) {
                    int v;

                    v = ((struct ipt_standard_target *)t)->verdict;
                    if (v < 0) {
                        /* Pop from stack? */
                        if (v != IPT_RETURN) {
                            verdict = (unsigned)(-v) - 1;
                            break;
                        }
                    }
                    e = back;

```



```

        back = get_entry(table_base,
                        back->comefrom);
        continue;
    }
    if (table_base + v
        != (void *)e + e->next_offset) {
        /* Save old back ptr in next entry */
        struct ipt_entry *next
            = (void *)e + e->next_offset;
        next->comefrom
            = (void *)back - table_base;
        /* set back pointer to next entry */
        back = next;
    }

    e = get_entry(table_base, v);
} else {
    verdict = t->u.kernel.target->target(pskb,
                                         in, out,
                                         hook,
                                         t->data,
                                         userdata);

    /* Target might have changed stuff. */
    ip = (*pskb)->nh.iph;
    datalen = (*pskb)->len - ip->ihl * 4;

    if (verdict == IPT_CONTINUE)
        e = (void *)e + e->next_offset;
    else
        /* Verdict */
        break;
}
} else {

    no_match:
        e = (void *)e + e->next_offset; /* 匹配失败，跳到下一个规则 */
    }
} while (!hotdrop);

read_unlock_bh(&table->lock);

#ifdef DEBUG_ALLOW_ALL
    return NF_ACCEPT;
#else
    if (hotdrop)
        return NF_DROP;
    else return verdict;
#endif
}

```

5.3 标准的match ip_packet_match函数 ip_tables.c

```

static inline int
ip_packet_match(const struct iphdr *ip,
               const char *indev,
               const char *outdev,

```

```

        const struct ipt_ip *ipinfo,
        int isfrag)
    {
        size_t i;
        unsigned long ret;

        /*定义一个宏，当bool和invflg的是一真一假的情况时，返回真。注意这里使用两个
        “！”的目的是使得这样计算后的值域只取0和1两个值*/
        #define FWINV(bool, invflg) ((bool) ^ !(ipinfo->invflags & invflg))

        /*处理源和目标ip地址，这个if语句的意义是：到达分组的源ip地址经过掩码处理后
        与规则中的ip不匹配并且规则中没有包含对ip地址的取反，或者规则中包含了对匹配地址
        的取反，但到达分组的源ip与规则中的ip地址匹配，if的第一部分返回真，同样道理处理
        到达分组的目的ip地址。这两部分任意部分为真时，源或者目标地址不匹配。*/
        if (FWINV((ip->saddr&ipinfo->smsk.s_addr) != ipinfo->src.s_addr,
            IPT_INV_SRCIP)
            || FWINV((ip->daddr&ipinfo->dmsk.s_addr) != ipinfo->dst.s_addr,
            IPT_INV_DSTIP)) {
            dprintf("Source or dest mismatch.\n");

            dprintf("SRC: %u.%u.%u.%u. Mask: %u.%u.%u.%u. Target: %u.%u.%u.%u.%
s\n",
                NIPQUAD(ip->saddr),
                NIPQUAD(ipinfo->smsk.s_addr),
                NIPQUAD(ipinfo->src.s_addr),
                ipinfo->invflags & IPT_INV_SRCIP ? " (INV)" : "");
            dprintf("DST: %u.%u.%u.%u Mask: %u.%u.%u.%u Target: %u.%u.%u.%u.%
s\n",
                NIPQUAD(ip->daddr),
                NIPQUAD(ipinfo->dmsk.s_addr),
                NIPQUAD(ipinfo->dst.s_addr),
                ipinfo->invflags & IPT_INV_DSTIP ? " (INV)" : "");
            return 0;
        }

        /*接着处理输入和输出的接口，for语句处理接口是否与规则中的接口匹配，不匹配
        时，ret返回非零，离开for语句后，处理接口的取反问题：当接口不匹配并且接口不取
        反，或者接口匹配，但是接口取反，说明接口不匹配。*/
        /* Look for ifname matches; this should unroll nicely. */

        /*输入接口*/
        for (i = 0, ret = 0; i < IFNAMSIZ/sizeof(unsigned long); i++) {
            ret |= (((const unsigned long *)indev)[i]
                ^ ((const unsigned long *)ipinfo->iniface)[i])
                & ((const unsigned long *)ipinfo->iniface_mask)[i];
        }

        if (FWINV(ret != 0, IPT_INV_VIA_IN)) {
            dprintf("VIA in mismatch (%s vs %s).%s\n",
                indev, ipinfo->iniface,
                ipinfo->invflags&IPT_INV_VIA_IN ? " (INV)": "");
            return 0;
        }

        /*输出接口*/
        for (i = 0, ret = 0; i < IFNAMSIZ/sizeof(unsigned long); i++) {
            ret |= (((const unsigned long *)outdev)[i]
                ^ ((const unsigned long *)ipinfo->outiface)[i])

```

```

        & ((const unsigned long *)ipinfo->outiface_mask)[i];
    }

    if (FWINV(ret != 0, IPT_INV_VIA_OUT)) {
        dprintf("VIA out mismatch (%s vs %s).%s\n",
            outdev, ipinfo->outiface,
            ipinfo->invflags&IPT_INV_VIA_OUT ? " (INV)":"");
        return 0;
    }

/* 检查协议是否匹配 */
/* Check specific protocol */
    if (ipinfo->proto
        && FWINV(ip->protocol != ipinfo->proto, IPT_INV_PROTO)) {
        dprintf("Packet protocol %hi does not match %hi.%s\n",
            ip->protocol, ipinfo->proto,
            ipinfo->invflags&IPT_INV_PROTO ? " (INV)":"");
        return 0;
    }

/*处理分片包的匹配情况*/
/* If we have a fragment rule but the packet is not a fragment
 * then we return zero */
    if (FWINV((ipinfo->flags&IPT_F_FRAG) && !isfrag, IPT_INV_FRAG)) {
        dprintf("Fragment rule but not fragment.%s\n",
            ipinfo->invflags & IPT_INV_FRAG ? " (INV)" : "");
        return 0;
    }

    return 1;        /* 以上所有都匹配则返回1 */
}

```

六、扩展的match

6.1 do_match函数 ip_tables.c

do_match通过IPT_MATCH_ITERATE宏来调用，
IPT_MATCH_ITERATE是在ipt_do_table函数中调用的宏
IPT_MATCH_ITERATE(e, do_match,

*pskb, in, out,
offset, &hotdrop)

定义如下：

```

#define IPT_MATCH_ITERATE(e, fn, args...) \
({ \
    unsigned int __i; \
    int __ret = 0; \
    struct ipt_entry_match *__match; \
    \
    for (__i = sizeof(struct ipt_entry); \
        __i < (e)->target_offset; \
        __i += __match->u.match_size) { \
        __match = (void *) (e) + __i; \
        \
        __ret = fn(__match, ## args); \
        if (__ret != 0) \
            break; \
    } \
})

```

```

        __ret;
    })
\

```

下面就是do_match函数:

```

static inline
int do_match(struct ipt_entry_match *m,
             const struct sk_buff *skb,
             const struct net_device *in,
             const struct net_device *out,
             int offset,
             const void *hdr,
             u_int16_t datalen,
             int *hotdrop)
{
    /* Stop iteration if it doesn't match */
    if (!m->u.kernel.match->match(skb, in, out, m->data,
                                   offset, hdr, datalen, hotdrop))
        return 1;
    else
        return 0;
}

```

实际上就是调用了m->u.kernel.match->match, 这个东西应该就是调用后面解释这里还出现了一个ipt_entry_match结构, 它用来把match的内核态与用户态关连起来

6.2 ipt_xxx.c文件

我们在编译内核的netfilter选项时, 有ah、esp、length……等一大堆的匹配选项, 他们既可以是模块的形式注册, 又可以是直接编译进内核, 所以, 他们应该是以单独的文件形式, 以:

```

module_init(init);
module_exit(cleanup);

```

这样形式存在的, 我们在源码目录下边, 可以看到Ipt_ah.c、Ipt_esp.c、Ipt_length.c等许多文件, 这些就是我们所要关心的了, 另一方面, 基本的TCP/UDP 的端口匹配, ICMP类型匹配不在此之列, 所以, 应该有初始化的地方, 我们注意到Ip_tables.c的init中, 有如下语句:

```

/* Noone else will be downing sem now, so we won't sleep */
down(&ipt_mutex);
list_append(&ipt_target, &ipt_standard_target);
list_append(&ipt_target, &ipt_error_target);
list_append(&ipt_match, &tcp_matchstruct);
list_append(&ipt_match, &udp_matchstruct);
list_append(&ipt_match, &icmp_matchstruct);
up(&ipt_mutex);

```

可以看到, 这里注册了standard_target、error_target两个target和tcp_matchstruct等三个match。这两个地方, 就是涉及到match在内核中的注册了, 以Ipt_*.c为例, 它们都是以下结构:

```

#include XXX

MODULE_AUTHOR ( )
MODULE_DESCRIPTION ( )
MODULE_LICENSE ( )

static int match ( )      /* ipt_match中的匹配函数 */
{
}

static int checkentry ( ) /* 检查entry有效性 */
{
}

```

```

}

static struct ipt_match XXX_match = { { NULL, NULL }, "XXX", &match,
                                       &checkentry, NULL, THIS_MODULE };

static int __init init(void)
{
    return ipt_register_match(&XXX_match);
}

static void __exit fini(void)
{
    ipt_unregister_match(&XXX_match);
}

module_init(init);
module_exit(fini);

```

其中，init函数调用ipt_register_match对一个struct ipt_match结构的XXX_match进行注册，另外，有两个函数match和checkentry。

6.3 ipt_match，内核中的match结构 ip_tables.h

```

struct ipt_match
{
    struct list_head list;          /* 可见ipt_match也由一个链表来维护 */
};

const char name[IPT_FUNCTION_MAXNAMELEN]; /* match名称 */

/* 匹配函数，最重要的部分，返回非0表示匹配成功，如果返回0且hotdrop设为
1，则表示该报文应当立刻丢弃。 */
/* Arguments changed since 2.4, as this must now handle
non-linear skbs, using skb_copy_bits and
skb_ip_make_writable. */
int (*match)(const struct sk_buff *skb,
             const struct net_device *in,
             const struct net_device *out,
             const void *matchinfo,
             int offset,
             int *hotdrop);

/* 在使用本Match的规则注入表中之前调用，进行有效性检查，如果返回0，规则
就不会加入iptables中。 */
int (*checkentry)(const char *tablename,
                 const struct ipt_ip *ip,
                 void *matchinfo,
                 unsigned int matchinfo_size,
                 unsigned int hook_mask);

/* 删除包含本match的entry时调用，与checkentry配合可用于动态内存分配和释
放 */
void (*destroy)(void *matchinfo, unsigned int matchinfo_size);

/* 是否为模块 */
struct module *me;
};

```

有了对这个结构的认识，就可以很容易地理解init函数了。我们也可以猜测，ipt_register_match的作用可能就是建立一个双向链表的过程，到时候要用某个match的某种功能，调用其成员函数即可。

当然，对于分析filter的实现，每个match/target的匹配函数才是我们关心的重点，但是这里为了不中断分析系统框架，就不再一一分析每个match的match函数

```
6.4 iptables_match, 用户态的match结构    ip_tables.h
struct iptables_match
{
    /* Match链，初始为NULL */
    struct iptables_match *next;

    /* Match名，和核心模块加载类似，作为动态链接库存在的Iptables
Extension的命名规则为libipt_'name'.so */
    ipt_chainlabel name;

    /*版本信息，一般设为NETFILTER_VERSION */
    const char *version;

    /* Match数据的大小，必须用IPT_ALIGN()宏指定对界*/
    size_t size;

    /*由于内核可能修改某些域，因此size可能与确切的用户数据不同，这时就应该
把不会被改变的数据放在数据区的前面部分，而这里就应该填写被改变的数据区大小；一
般来说，这个值和size相同*/
    size_t userspace_size;

    /*当iptables要求显示当前match的信息时（比如iptables-m ip_ext -h），就会
调用这个函数，输出在iptables程序的通用信息之后. */
    void (*help)(void);

    /*初始化，在parse之前调用. */
    void (*init)(struct ipt_entry_match *m, unsigned int *nfcache);

    /*扫描并接收本match的命令行参数，正确接收时返回非0，flags用于保存状态信
息*/
    int (*parse)(int c, char **argv, int invert, unsigned int *flags,
                  const struct ipt_entry *entry,
                  unsigned int *nfcache,
                  struct ipt_entry_match **match);

    /* 前面提到过这个函数，当命令行参数全部处理完毕以后调用，如果不正确，应
该退出 (exit_error()) */
    void (*final_check)(unsigned int flags);

    /*当查询当前表中的规则时，显示使用了当前match的规则*/
    void (*print)(const struct ipt_ip *ip,
                  const struct ipt_entry_match *match, int numeric);

    /*按照parse允许的格式将本match的命令行参数输出到标准输出，用于
iptables-save命令. */
    void (*save)(const struct ipt_ip *ip,
                  const struct ipt_entry_match *match);

    /* NULL结尾的参数列表，struct option与getopt(3)使用的结构相同*/
    const struct option *extra_opts;
```

```

        /* Ignore these men behind the curtain: */
        unsigned int option_offset;
        struct ipt_entry_match *m;
        unsigned int mflags;
        unsigned int used;
#ifdef NO_SHARED_LIBS
        unsigned int loaded; /* simulate loading so options are merged properly */
#endif
};

```

6.5 ipt_entry_match结构 ip_tables.h

ipt_entry_match将内核态与用户态关联起来，按我的理解，内核和用户在注册和维护match时使用的是各自的match结构ipt_match和iptables_match，但在具体应用到某个规则时则需要统一成ipt_entry_match结构。

前面说过，match区存储在ipt_entry的末尾，target在最后，结合ipt_entry_match的定义，可以知道一条具体的规则中存储的数据结构不是：

ipt_entry + ipt_match1 + ipt_match2 + ipt_match3 + ... + target

而是：

ipt_entry + ipt_entry_match1 + ipt_entry_match2 + ipt_entry_match3 + ... + target

```

struct ipt_entry_match
{
    union {
        struct {
            u_int16_t match_size;

            /* 用户态 */
            char name[IPT_FUNCTION_MAXNAMELEN];
        } user;
        struct {
            u_int16_t match_size;

            /* 内核态 */
            struct ipt_match *match;
        } kernel;

        /* 总长度 */
        u_int16_t match_size;
    } u;

    unsigned char data[0];
};

```

里面定义了两个数据结构，user和kernel，很明显，是分别为iptables_match和ipt_match准备的

前面在do_match函数中出现的m->u.kernel.match->match()函数，也就是调用ipt_match里的match函数了，接下来要关心的就是如何将ipt_entry_match与ipt_match关联起来。换句话说，注册时还是ipt_match结构的match是何时变成ipt_entry_match结构的？

还记得注册table时调用的translate_table()函数吗

IPT_ENTRY_ITERATE宏出现三次，分别调用了

check_entry_size_and_hooks, check_entry, cleanup_entry, 三个函数

check_entry_size_and_hooks用来做一些边界检查，检查数据结构的长度之类的，略过

cleanup_entry，很明显，释放空间用的

下面看看check_entry

6.6 check_entry和check_match函数 ip_tables.c

顾名思义，对entry结构进行检查

```
check_entry(struct ipt_entry *e, const char *name, unsigned int size,
            unsigned int *i)
```

```
{
    struct ipt_entry_target *t;
    struct ipt_target *target;
    int ret;
    unsigned int j;

    /* 检查flag和invflag ... */
    if (!ip_checkentry(&e->ip)) {
        duprintf("ip_tables: ip check failed %p %s.\n", e, name);
        return -EINVAL;
    }
}
```

/* 先别看后面，这里是重点，之前遍历时用了IPT_ENTRY_ITERATE宏，这里又出现了用来遍历match的IPT_MATCH_ITERATE宏，两个很像。

另外IPT_MATCH_ITERATE宏前面看到过一次，在调用钩子函数时的ipt_do_table()函数里出现过，那里是用来遍历match并调用do_match()函数的。怎么样，思路又回到开头扩展的match那里了吧，那里是调用阶段，而这里正好是之前的初始化阶段。应该说这里才是IPT_MATCH_ITERATE和ipt_entry_match的第一次出现。

遍历该entry里的所有match，并对每一个match调用检查函数check_match() */

```
    j = 0;
    ret = IPT_MATCH_ITERATE(e, check_match, name, &e->ip, e->comefrom, &j);
    if (ret != 0)
        goto cleanup_matches;

    /* 下面是关于target的部分 */
    t = ipt_get_target(e);
    target = ipt_find_target_lock(t->u.user.name, &ret, &ipt_mutex);
    if (!target) {
        duprintf("check_entry: `%s' not found\n", t->u.user.name);
        goto cleanup_matches;
    }
    if (!try_module_get(target->me)) {
        up(&ipt_mutex);
        ret = -ENOENT;
        goto cleanup_matches;
    }
    t->u.kernel.target = target;
    up(&ipt_mutex);

    if (t->u.kernel.target == &ipt_standard_target) {
        if (!standard_check(t, size)) {
            ret = -EINVAL;
            goto cleanup_matches;
        }
    } else if (t->u.kernel.target->checkentry
        && !t->u.kernel.target->checkentry(name, e, t->data,
            t->u.target_size
            - sizeof(*t),
            e->comefrom)) {
        module_put(t->u.kernel.target->me);
        duprintf("ip_tables: check failed for `%s'.\n",
            t->u.kernel.target->name);
        ret = -EINVAL;
    }
}
```



```

        goto cleanup_matches;
    }

    (*i)++;
    return 0;

cleanup_matches:
    IPT_MATCH_ITERATE(e, cleanup_match, &j);
    return ret;
}

```

再看一下IPT_MATCH_ITERATE宏的定义：

```

#define IPT_MATCH_ITERATE(e, fn, args...) \
({ \
    unsigned int __i; \
    int __ret = 0; \
    struct ipt_entry_match *__match; \
    \
    for (__i = sizeof(struct ipt_entry); \
        __i < (e)->target_offset; \
        __i += __match->u.match_size) { \
        __match = (void *) (e) + __i; \
        \
        __ret = fn(__match, ## args); \
        if (__ret != 0) \
            break; \
    } \
    __ret; \
})

```

可以看到，在这个宏里，ipt_entry_match结构出现了，就是说，到这里为止，entry结构中的match结构已经由ipt_match替换成了ipt_entry_match，当然这只是形式上，因为具体结构还是有区别，所以还要对新的ipt_entry_match做一些初始化，也就是把ipt_match里的实际内容关联过来

check_match()对match结构进行检查：

```

static inline int
check_match(struct ipt_entry_match *m,
            const char *name,
            const struct ipt_ip *ip,
            unsigned int hookmask,
            unsigned int *i)
{
    int ret;
    struct ipt_match *match;

```

/*根据规则中Match的名称，在已注册好的ipt_match双向链表中查找对应结点

可能有一点疑问就是为什么用m->u.user.name作为名字来查找一个ipt_match，在定义ipt_entry_match的时候应该只是把它的指针指向了ipt_match的开头位置，并没有对里面的name变量赋值吧。

我猜想是这两个结构里第一个变量分别是一个list_head结构体和一个u_int16_t，它们都应该是一个（还是两个？）地址变量，所以占用同样的空间，那么两个作为结构里第二个参数的字符串name[IPT_FUNCTION_MAXNAMELEN] 就刚好重合了 */

```

    match = find_match_lock(m->u.user.name, &ret, &ipt_mutex);
    if (!match) {
        duprintf("check_match: '%s' not found\n", m->u.user.name);
        return ret;
    }

```

```

    }

    if (!try_module_get(match->me)) {
        up(&ipt_mutex);
        return -ENOENT;
    }

    /* 再回到开头的do_match()函数，这下全部联系起来了把 */
    m->u.kernel.match = match;
    up(&ipt_mutex);

    /* 调用match里的checkentry做一些检查 */
    if (m->u.kernel.match->checkentry
        && !m->u.kernel.match->checkentry(name, ip, m->data,
                                           m->u.match_size - sizeof(*m),
                                           hookmask)) {
        module_put(m->u.kernel.match->me);
        duprintf("ip_tables: check failed for `%s'.\n",
                 m->u.kernel.match->name);
        return -EINVAL;
    }

    (*i)++;
    return 0;
}

```

还有一点，这里并没有讲到具体的match的实现，包括每个match是如何放进entry里，entry又是如何放进table里的。也就是说，分析了半天，实际上我们的table里的entry部分根本就是空的，不过也对，内核在初始化netfilter时只是注册了3个表（filter，nat，mangle），而里面的规则本来就是空的。至于具体的entry和match是如何加入进来的，就是netfilter在用户空间的配置工具iptables的任务了。

七、target 匹配

7.1 ipt_target和ipt_entry_target结构 ip_tables.h

ipt_target和ipt_match结构类似：

```

struct ipt_target
{
    struct list_head list;

    const char name[IPT_FUNCTION_MAXNAMELEN];

```

/* 在使用本Match的规则注入表中之前调用，进行有效性检查，如果返回0，规则就不会加入iptables中 */

```

    int (*checkentry)(const char *tablename,
                      const struct ipt_entry *e,
                      void *targinfo,
                      unsigned int targinfo_size,
                      unsigned int hook_mask);

```

/* 在包含本Target的规则从表中删除时调用，与checkentry配合可用于动态内存分配和释放 */

```

    void (*destroy)(void *targinfo, unsigned int targinfo_size);

```

/* target的模块函数，如果还需要继续处理则返回IPT_CONTINUE（-1），否则返回NF_ACCEPT、NF_DROP等值，它的调用者根据它的返回值来判断如何处理它处理过的报文*/

```

    unsigned int (*target)(struct sk_buff **pskb,
                          const struct net_device *in,

```

```

        const struct net_device *out,
        unsigned int hooknum,
        const void *targinfo,
        void *userdata);

/* 表示当前Target是否为模块 (NULL为否) */
    struct module *me;
};
ipt_entry_target和ipt_entry_match也几乎一模一样:
struct ipt_entry_target
{
    union {
        struct {
            u_int16_t target_size;
            char name[IPT_FUNCTION_MAXNAMELEN];
        } user;

        struct {
            u_int16_t target_size;
            struct ipt_target *target;
        } kernel;

        u_int16_t target_size;
    } u;

    unsigned char data[0];
};

```

看上去target和match好像没有区别,但当然,一个是条件,一个是动作,接着往下看是不是真的一样

之前有两个地方出现了ipt_target,一次是在ipt_do_table()函数里,当匹配到match后开始匹配target,另一次是在check_entry()里,检查完match后开始检查target先看前一个

7.2 ipt_standard_target结构 ip_tables.h

再看一次ipt_do_table这个函数,前面匹配match的部分略过,从匹配match成功的地方开始:

```

ipt_do_table( )
{
    ..... /* 略去 */
    if (ip_packet_match(ip, indev, outdev, &e->ip, offset)) {
        struct ipt_entry_target *t;

        if (IPT_MATCH_ITERATE(e, do_match,
                                *pskb, in, out,
                                offset, &hotdrop) != 0)
            goto no_match;
        /* 这里开始说明匹配match成功了,开始匹配target */

        ADD_COUNTER(e->counters, ntohs(ip->tot_len), 1);
    }
}

```

/* ipt_get_target获取当前target, t是一个ipt_entry_target结构,这个函数就是简单的返回e->target_offset

每个entry只有一个target,所以不需要像match一样遍历,直接指针指过去了*/

```

    t = ipt_get_target(e);
    IP_NF_ASSERT(t->u.kernel.target);

```

/* 这里都还是和扩展的match的匹配很像，但是下面一句有句注释：Standard target? 判断当前target是否标准的target? 而判断的条件是u.kernel.target->target，就是ipt_target结构里的target函数是否为空，而下面还出现了ipt_standard_target结构和verdict变量，好吧，先停下，看看ipt_standard_target结构再说 */

```

        if (!t->u.kernel.target->target) {
            int v;

            v = ((struct ipt_standard_target *)t)->verdict;
            if (v < 0) {
                ..... /* 略去 */
            }
        }

```

ipt_standard_target的定义：

```

struct ipt_standard_target
{
    struct ipt_entry_target target;
    int verdict;
};

```

也就比ipt_entry_target多了一个verdict（判断），请看前面的nf_hook_slow（）函数，里面也有verdict变量，用来保存hook函数的返回值，常见的有这些

```

#define NF_DROP 0
#define NF_ACCEPT 1
#define NF_STOLEN 2
#define NF_QUEUE 3
#define NF_REPEAT 4
#define RETURN IPT_RETURN
#define IPT_RETURN (-NF_MAX_VERDICT - 1)
#define NF_MAX_VERDICT NF_REPEAT

```

我们知道chain（链）是某个检查点上检查的规则集合。除了默认的chain外，用户还可以创建新的chain。在iptables中，同一个chain里的规则是连续存放的。默认的chain的最后一条规则的target是chain的policy。用户创建的chain的最后一条规则的target的调用返回值是NF_RETURN，遍历过程将返回原来的chain。规则中的target也可以指定跳转到某个用户创建的chain上，这时它的target是ipt_standard_target，并且这个target的verdict值大于0。如果在用户创建的chain上没有找到匹配的规则，遍历过程将返回到原来chain的下一条规则上。

事实上，target也是分标准的和扩展的，但前面说了，毕竟一个是条件，一个是动作，target的标准和扩展的关系和match还是不太一样的，不能一概而论，而且在标准的target里还可以根据verdict的值再划分为内建的动作或者跳转到自定义链

简单的说，标准target就是内核内建的一些处理动作或其延伸

扩展的当然就是完全由用户定义的处理动作

再看if (!t->u.kernel.target->target)就明白了，如果target函数是空的，就是标准target，因为它不需要用户再提供target函数了，而反之是就是扩展的target，那么再看ipt_do_table()吧，还是只看一部分，否则眼花。

```

        if (!t->u.kernel.target->target) {
            /* 如果target为空，是标准target */
            int v;
            v = ((struct ipt_standard_target *)t)->verdict;
            if (v < 0) {
                /*v小于0，动作是默认内建的动作，也可能是自定义链已经结束而返回return标志*/
                if (v != IPT_RETURN) { /*如果不是Return，则是内建的动作*/
                    verdict = (unsigned)(-v) - 1;
                    break;
                }
            }
        }

```

```

        e = back;
        /* e和back分别是当前表的当前Hook的规则的开始偏移量和上限偏移量，即entry的头
        和尾，e=back */

        back = get_entry(table_base, back->comefrom);
        continue;
    }
    /* v大于等于0，处理用户自定义链，如果当前链后还有规则，而要跳到自定义链去执
    行，那么需要保存一个back点，以指示程序在匹配完自定义链后，应当继续匹配的规则位
    置，自然地， back点应该为当前规则的下一条规则（如果存在的话）
    至于为什么下一条规则的地址是table_base+v，就要去看具体的规则是如何添加的了
    */
    if (table_base + v != (void *)e + e->next_offset) {
        /* 如果还有规则 */
        /* Save old back ptr in next entry */
        struct ipt_entry *next = (void *)e + e->next_offset;
        next->comefrom = (void *)back - table_base;
        /* set back pointer to next entry */
        back = next;
    }

    e = get_entry(table_base, v);
} else {
    /* 如果是扩展的target，则调用target函数，返回值给verdict */
    verdict = t->u.kernel.target->target(pskb,
        in, out,
        hook,
        t->data,
        userdata);

    /*Target函数有可能已经改变了stuff，所以这里重新定位指针*/
    ip = (*pskb)->nh.iph;
    datalen = (*pskb)->len - ip->ihl * 4;
    /*如果返回的动作是继续检查下一条规则，则设置当前规则为下一条规则，
    继续循环，否则，就跳出循环，因为在ipt_do_table函数末尾有return verdict;表明，则
    将target函数决定的返回值返回给调用函数nf_iterate，由它来根据verdict决定数据包的
    命运*/
    if (verdict == IPT_CONTINUE)
        e = (void *)e + e->next_offset;
    else
        /* Verdict */
        break;
}

```