一、概述

1. Netfilter/IPTables 框架简介

Netfilter/IPTables 是继 2.0.x 的 IPfwadm、2.2.x 的 IPchains 之后，新一代的 Linux 防火墙机制。Netfilter 采用模块化设计，具有良好的可扩充性。其重要工具模块 IPTables 连接到 Netfilter 的架构中，并允许使用者对数据报进行过滤、地址转换、处理等操作。

Netfilter 提供了一个框架，将对网络代码的直接干涉降到最低，并允许用规定的接口将其他包处理代码以模块的形式添加到内核中，具有极强的灵活性。

2. 主要源代码文件

- Linux 内核版本：2.4.21
- Netfilter 主文件：net/core/netfilter.c

  Netfilter 主头文件：include/linux/netfilter.h
- IPv4 相关：

  c 文件：net/ipv4/netfilter/*.c

  头文件：include/linux/netfilter_ipv4.h

  include/linux/netfilter_ipv4/*.h
- IPv4 协议栈主体的部分 c 文件，特别是与数据报传送过程有关的部分：

  ip_input.c，ip_forward.c，ip_output.c，ip_fragment.c 等

二、Netfilter/IPTables-IPv4 总体架构

Netfilter 主要通过表、链实现规则，可以这么说，Netfilter 是表的容器，表是链的容器，链是规则的容器，最终形成对数据报处理规则的实现。

详细地说，Netfilter/IPTables 的体系结构可以分为三个大部分：

1. Netfilter 的 HOOK 机制

Netfilter 的通用框架不依赖于具体的协议，而是为每种网络协议定义一套 HOOK 函数。这些 HOOK 函数在数据报经过协议栈的几个关键点时被调用，在这几个点中，协议栈将数据报及 HOOK 函数标

号作为参数，传递给 Netfilter 框架。

对于它在网络堆栈中增加的这些 HOOK，内核的任何模块可以对每种协议的一个或多个 HOOK 进行注册，实现挂接。这样当某个数据报被传递给 Netfilter 框架时，内核能检测到是否有任何模块对该协议和 HOOK 函数进行了注册。若注册了，则调用该模块的注册时使用的回调函数，这样这些模块就有机会检查、修改、丢弃该数据报及指示 Netfilter 将该数据报传入用户空间的队列。

这样，HOOK 提供了一种方便的机制：在数据报通过 Linux 内核的不同位置上截获和操作处理数据报。

2．IPTables 基础模块

IPTables 基础模块实现了三个表来筛选各种数据报，具体地讲，Linux2.4 内核提供的这三种数据报的处理功能是相互间独立的模块，都基于 Netfilter 的 HOOK 函数和各种表、链实现。这三个表包括：filter 表，nat 表以及 mangle 表。

3．具体功能模块

1．数据报过滤模块

2．连接跟踪模块（Conntrack）

3．网络地址转换模块（NAT）

4．数据报修改模块（mangle）

5．其它高级功能模块

于是，Netfilter/IPTables 总体架构如图

http://blog.chinaunix.net/photo/24896_061206192251.jpg 所示

三、HOOK 的实现

1．Netfilter-IPv4 中的 HOOK

Netfilter 模块需要使用 HOOK 来启用函数的动态钩接，它在 IPv4 中定义了五个 HOOK（位于文件 include/linux/netfilter_ipv4.h，Line 39），分别对应 0-4 的 hooknum

简单地说，数据报经过各个 HOOK 的流程如下：

数据报从进入系统，进行 IP 校验以后，首先经过第一个 HOOK 函数 NF_IP_PRE_ROUTING 进行处理；然后就进入路由代码，其决定该数据报是需要转发还是发给本机的；若该数据报是发被本机的，则该数据经过 HOOK 函数 NF_IP_LOCAL_IN 处理以后然后传递给上层协议；若该数据报应该被转发则它被 NF_IP_FORWARD 处理；经过转发的数据报经过最后一个 HOOK 函数 NF_IP_POST_ROUTING 处理以后，

再传输到网络上。本地产生的数据经过 HOOK 函数 NF_IP_LOCAL_OUT 处理后，进行路由选择处理，然后经过 NF_IP_POST_ROUTING 处理后发送出去。

总之，这五个 HOOK 所组成的 Netfilter-IPv4 数据报筛选体系如图 http://blog.chinaunix.net/photo/24896_061206192311.jpg：（注：下面所说 Netfilter/IPTables 均基于 IPv4，不再赘述）

详细地说，各个 HOOK 及其在 IP 数据报传递中的具体位置如图 http://blog.chinaunix.net/photo/24896_061206192340.jpg

- *NF_IP_PRE_ROUTING (0)*

  数据报在进入路由代码被处理之前，数据报在 IP 数据报接收函数 ip_rcv()（位于 net/ipv4/ip_input.c，Line379）的最后，也就是在传入的数据报被处理之前经过这个 HOOK。在 ip_rcv()中挂接这个 HOOK 之前，进行的是一些与类型、长度、版本有关的检查。

  经过这个 HOOK 处理之后，数据报进入 ip_rcv_finish()（位于 net/ipv4/ip_input.c，Line306），进行查路由表的工作，并判断该数据报是发给本地机器还是进行转发。

  在这个 HOOK 上主要是对数据报作报头检测处理，以捕获异常情况。

  涉及功能（优先级顺序）：Conntrack(-200)、mangle(-150)、DNAT(-100)

- *NF_IP_LOCAL_IN (1)*

  目的地为本地主机的数据报在 IP 数据报本地投递函数 ip_local_deliver()（位于 net/ipv4/ip_input.c，Line290）的最后经过这个 HOOK。

  经过这个 HOOK 处理之后，数据报进入 ip_local_deliver_finish()（位于 net/ipv4/ip_input.c，Line219）

  这样，IPTables 模块就可以利用这个 HOOK 对应的 INPUT 规则链表来对数据报进行规则匹配的筛选了。防火墙一般建立在这个 HOOK 上。

  涉及功能：mangle(-150)、filter(0)、SNAT(100)、Conntrack(INT_MAX-1)

- *NF_IP_FORWARD (2)*

  目的地非本地主机的数据报，包括被 NAT 修改过地址的数据报，都要在 IP 数据报转发函数 ip_forward()（位于 net/ipv4/ip_forward.c，Line73）的最后经过这个 HOOK。

  经过这个 HOOK 处理之后，数据报进入 ip_forward_finish()（位于 net/ipv4/ip_forward.c，Line44）

  另外，在 net/ipv4/ipmr.c 中的 ipmr_queue_xmit()函数（Line1119）最后也会经过这

个 HOOK。（ipmr 为多播相关，估计是在需要通过路由转发多播数据时的处理）

这样，IPTables 模块就可以利用这个 HOOK 对应的 FORWARD 规则链表来对数据报进行规则匹配的筛选了。

涉及功能：mangle(-150)、filter(0)

- *NF_IP_LOCAL_OUT (3)*

本地主机发出的数据报在 IP 数据报构建/发送函数 ip_queue_xmit()（位于 net/ipv4/ip_output.c，Line339）、以及 ip_build_and_send_pkt()（位于 net/ipv4/ip_output.c，Line122）的最后经过这个 HOOK。（在数据报处理中，前者最为常用，后者用于那些不传输有效数据的 SYN/ACK 包）

经过这个 HOOK 处理后，数据报进入 ip_queue_xmit2()（位于 net/ipv4/ip_output.c，Line281）

另外，在 ip_build_xmit_slow()（位于 net/ipv4/ip_output.c，Line429）和 ip_build_xmit()（位于 net/ipv4/ip_output.c，Line638）中用于进行错误检测；在 igmp_send_report()（位于 net/ipv4/igmp.c，Line195）的最后也经过了这个 HOOK，进行多播时相关的处理。

这样，IPTables 模块就可以利用这个 HOOK 对应的 OUTPUT 规则链表来对数据报进行规则匹配的筛选了。

涉及功能：Conntrack(-200)、mangle(-150)、DNAT(-100)、filter(0)

- *NF_IP_POST_ROUTING (4)*

所有数据报，包括源地址为本地主机和非本地主机的，在通过网络设备离开本地主机之前，在 IP 数据报发送函数 ip_finish_output()（位于 net/ipv4/ip_output.c，Line184）的最后经过这个 HOOK。

经过这个 HOOK 处理后，数据报进入 ip_finish_output2()（位于 net/ipv4/ip_output.c，Line160）另外，在函数 ip_mc_output()（位于 net/ipv4/ip_output.c，Line195）中在克隆新的网络缓存 skb 时，也经过了这个 HOOK 进行处理。

涉及功能：mangle(-150)、SNAT(100)、Conntrack(INT_MAX)

其中，入口为 net_rx_action()（位于 net/core/dev.c，Line1602），作用是将数据报一个个地从 CPU 的输入队列中拿出，然后传递给协议处理例程。

出口为 dev_queue_xmit()（位于 net/core/dev.c，Line1035），这个函数被高层协议的

实例使用，以数据结构 struct sk_buff *skb 的形式在网络设备上发送数据报。

2．HOOK 的调用

HOOK 的调用是通过宏 NF_HOOK 实现的，其定义位于 include/linux/Netfilter.h，Line122：

```
#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)))
```

这里先调用 list_empty 函数检查 HOOK 点存储数组 nf_hooks 是否为空，为空则表示没有 HOOK 注册，则直接调用 okfn 继续处理。如果不为空，则转入 nf_hook_slow()函数。

nf_hook_slow()函数（位于 net/core/netfilter.c，Line449）的工作主要是读 nf_hook 数组遍历所有的 nf_hook_ops 结构，并调用 nf_hookfn()处理各个数据报。

即 HOOK 的调用过程如图 http://blog.chinaunix.net/photo/24896_061206192356.jpg 所示

下面说明一下 NF_HOOK 的各个参数：

- pf：协议族标识，相关的有效协议族列表位于 include/linux/socket.h，Line 178。对于 IPv4，应该使用协议族 PF_INET；
- hook：HOOK 标识，即前面所说 5 个 HOOK 对应的 hooknum；
- skb：是含有需要被处理包的 sk_buuff 数据结构的指针。sk_buff 是 Linux 网络缓存，指那些 linux 内核处理 IP 分组报文的缓存，即套接字缓冲区。

  网卡收到 IP 分组报文后，将它们放入 sk_buff，然后再传送给网络堆栈，网络堆栈几乎一直要用到 sk_buff。其定义在 include/linux/skbuff.h，Line 129，下面列出我认为对分析有意义的部分成员：
  - `struct sock *sk;`：指向创建分组报文的 socket；
  - `struct timeval stamp;`：分组报文到达系统的时间；
  - 下面是三个 union，存放的是各层中各种协议的报文头指针：
    - h 对应传输层的报头
    - nh 对应网络层的报头
    - mac 对应 MAC 层的报头
  - `unsigned int len;`：套接字缓存所代表的报文长度，即从`unsigned char

*data;`的位置算起的当前有效报文长度。

- `unsigned char pkt_type,`：表示报文的类型，具体类型定义在 include/linux/if_packet.h，Line24：

```
#define PACKET_HOST 0 // 发送到本机的报文
#define PACKET_BROADCAST 1 // 广播报文
#define PACKET_MULTICAST 2 // 多播报文
#define PACKET_OTHERHOST 3 // 表示目的地非本机但被本机 接收的报文
#define PACKET_OUTGOING 4 // 离开本机的报文
/* These ones are invisible by user level */
#define PACKET_LOOPBACK 5 // 本机发给自己的报文
#define PACKET_FASTROUTE 6 // 快速路由报文
```

- indev：输入设备，收到数据报的网络设备的 net_device 数据结构指针，即数据报到达的接口。
  - 用于 NF_IP_PRE_ROUTING 和 NF_IP_LOCAL_IN 两个 HOOK
- outdev：输出设备，数据报离开本地所要使用的网络设备的 net_device 数据结构指针。
  - 用于 NF_IP_LOCAL_OUT 和 NF_IP_POST_ROUTING 两个 HOOK
  - 注意：在通常情况下，在一次 HOOK 调用中，indev 和 outdev 中只有一个参数会被使用
- okfn：下一步要处理的函数。即如果有 HOOK 函数，则处理完所有的 HOOK 函数，且所有向该 HOOK 注册过的筛选函数都返回 NF_ACCEPT 时，调用这个函数继续处理；如果没有注册任何 HOOK，则直接调用此函数。其 5 个参数将由宏 NF_HOOK 传入。

3．HOOK 点的实现

对应于各个不同协议的不同 HOOK 点是由一个二维数组 nf_hooks 存储的（位于 net/core/netfilter.c，Line 47），具体的 HOOK 点则由数据结构 nf_hook_ops（位于 include/linux/netfilter.h，Line 44）实现。如图 http://blog.chinaunix.net/photo/24896_061206192528.jpg 所示：

其中，nf_hook_ops 成员中：

- `int priority;` priority 值越小，优先级越高，相关优先级在 include/linux/netfilter_ipv4.h，Line52 中枚举定义：

```
enum NF_IP_hook_priorities {
NF_IP_PRI_FIRST = INT_MIN,
NF_IP_PRI_CONNTRACK= -200,
NF_IP_PRI_MANGLE = -150,
NF_IP_PRI_NAT_DST = -100,
NF_IP_PRI_FILTER = 0,
NF_IP_PRI_NAT_SRC = 100,
NF_IP_PRI_LAST = INT_MAX,
};
```

- `nf_hookfn *hook;` 为处理函数的指针，其函数指针类型定义位于 include/linux/netfilter.h，Line38，为：

```
typedef unsigned int nf_hookfn(unsigned int hooknum,
struct sk_buff **skb,
const struct net_device *in,
const struct net_device *out,
int (*okfn)(struct sk_buff *));
```

这是 nf_hook_ops 中最关键的成员，其五个参数分别对应前面所解释的 NF_HOOK 中弟 2 到 6 个参数

调用 HOOK 的包筛选函数必须返回特定的值，这些值以宏的形式定义于头文件 include/linux/netfilter.h 中（Line15），分别为：

- NF_DROP(0)：丢弃此数据报，禁止包继续传递，不进入此后的处理流程；
- NF_ACCEPT(1)：接收此数据报，允许包继续传递，直至传递到链表最后，而进入 okfn 函数；
    - 以上两个返回值最为常见
- NF_STOLEN(2)：数据报被筛选函数截获，禁止包继续传递，但并不释放数据报的资源，这个数据报及其占有的 sk_buff 仍然有效（e.g. 将分片的数据报一一截获，然后将其装配起来再进行其他处理）；
- NF_QUEQUE(3)：将数据报加入用户空间队列，使用户空间的程序可以直接进行处理；
    - 在 nf_hook_slow()以及 nf_reinject()函数（位于 net/core/netfilter.c，Line449，Line505）中，当由调用 nf_iterate()函数（位于 net/core/netfilter.c，Line339，作用为遍历所有注册的 HOOK 函数，并返回相应的 NF_XX 值）而返回的 verdict 值为 NF_QUEUE 时（即当前正在执行的这个 HOOK 筛选函数要求将数据报加入用户空间队列），会调用

nf_queue()函数（位于net/core/netfilter.c，Line407）

- nf_queue()函数将这个数据报加入用户空间队列nf_info（位于include/linux/netfilter.h，Line77），并保存其设备信息以备用

- NF_REPEAT(4)：再次调用当前这个HOOK的筛选函数，进行重复处理。

4．HOOK的注册和注销

HOOK的注册和注销分别是通过nf_register_hook()函数和nf_unregister_hook()函数（分别位于net/core/netfilter.c，Line60，76）实现的，其参数均为一个nf_hook_ops结构，二者的实现也非常简单。

nf_register_hook()的工作是首先遍历nf_hooks[][]，由HOOK的优先级确定在HOOK链表中的位置，然后根据优先级将该HOOK的nf_hook_ops加入链表；

nf_unregister_hook()的工作更加简单，其实就是将该HOOK的nf_hook_ops从链表中删除。

四、IPTables系统

1．表－规则系统

IPTables是基于Netfilter基本架构实现的一个可扩展的数据报高级管理系统，利用table、chain、rule三级来存储数据报的各种规则。系统预定义了三个table：

- filter：数据报过滤表（文件net/ipv4/netfilter/iptable_filter.c）

  监听NF_IP_LOCAL_IN、NF_IP_FORWARD和NF_IP_LOCAL_OUT三个HOOK，作用是在所有数据报传递的关键点上对其进行过滤。

- nat：网络地址转换表

  监听NF_IP_PRE_ROUTING、NF_IP_POST_ROUTING和NF_IP_LOCAL_OUT三个HOOK，作用是当新连接的第一个数据报经过时，在nat表中决定对其的转换操作；而后面的其它数据报都将根据第一个数据报的结果进行相同的转换处理。

- mangle：数据报修改表（位于net/ipv4/netfilter/iptable_mangle.c）

  监听NF_IP_PRE_ROUTING和NF_IP_LOCAL_OUT两个HOOK，作用是修改数据报报头中的一些值。

2．表的实现

- 表的基本数据结构是ipt_table（位于include/linux/netfilter_ipv4/ip_tables.h，Line413）：

```
struct ipt_table
{
struct list_head list; // 一个双向链表
char name[IPT_TABLE_MAXNAMELEN]; // 被用户空间使用的表函数的名字
struct ipt_replace *table; // 表初始化的模板，定义了一个初始化用的该 // 表的所默认的 HOOK 所包含的规则等信息，
// 用户通过系统调用进行表的替换时也要用
unsigned int valid_hooks; // 表所监听的 HOOK，实质是一个位图
rwlock_t lock; // 整个表的读/写自旋锁
struct ipt_table_info *private; // 表所存储的数据信息，也就是实际的数据区，
// 仅在处理 ipt_table 的代码内部使用
struct module *me; // 如果是模块，那么取 THIS_MODULE，否则取 NULL
};
```

其中：

- `unsigned int valid_hooks;`这个位图有两个作用：一是检查 Netfilter 中哪些 HOOK 对应着合法的 entries；二是用来为 ipt_match 以及 ipt_target 数据结构中的 checkentry() 函数核算可能的 HOOK。
- `struct module *me;`当取值为 THIS_MODULE 时，可以阻止用户 rmmod 一个仍然被某个规则指向的模块的尝试。
- `struct ipt_replace *table;`的数据结构是被用户空间用来替换一个表的，其定义位于 include/linux/netfilter_ipv4/ip_tables.h，Line230：

```
struct ipt_replace
{
char name[IPT_TABLE_MAXNAMELEN];
unsigned int valid_hooks;
unsigned int num_entries; // 规则表入口的数量
unsigned int size; // 新的规则表的总大小
/* Hook entry points. */
unsigned int hook_entry[NF_IP_NUMHOOKS]; // 表所监听 HOOK 的规则入口，
// 是对于 entries[ ]的偏移
unsigned int underflow[NF_IP_NUMHOOKS]; // 规则表的最大下界
unsigned int num_counters; // 旧的计数器数目，即当前的旧 entries 的数目
struct ipt_counters *counters; // 旧的计数器
struct ipt_entry entries[0]; // 规则表入口
};
```

- 上文所提到的 filter、nat 和 mangle 表分别是 ipt_table 这个数据结构的三个实例：packet_filter（位于 net/ipv4/netfilter/iptable_filter.c，Line84）、nat_table（位于 net/ipv4/netfilter/ip_nat_rule.c，Line104）以及 packet_mangler（位于

net/ipv4/netfilter/iptable_mangle.c，Line117）

- ipt_table_info（位于net/ipv4/netfilter/ip_tables.c，Line86）是实际描述规则表的数据结构：

```
struct ipt_table_info
{
unsigned int size;
unsigned int number; // 表项的数目
unsigned int initial_entries; // 初始表项数目
unsigned int hook_entry[NF_IP_NUMHOOKS]; // 所监听HOOK的规则入口
unsigned int underflow[NF_IP_NUMHOOKS]; // 规则表的最大下界
char entries[0] ____cacheline_aligned; // 规则表入口，即真正的规则存储结构 //
ipt_entry组成块的起始地址，对多CPU，每个CPU对应一个
};
```

3．规则的实现

IPTables中的规则表可以在用户空间中使用，但它所采用的数据结构与内核空间中的是一样的，只不过有些成员不会在用户空间中使用。

一个完整的规则由三个数据结构共同实现，分别是：

- 一个ipt_entry结构，存储规则的整体信息；
- 0或多个ipt_entry_match结构，存放各种match，每个结构都可以存放任意的数据，这样也就拥有了良好的可扩展性；
- 1个ipt_entry_target结构，存放规则的target，类似的，每个结构也可以存放任意的数据。

下面将依次对这三个数据结构进行分析：

i.存储规则整体的结构ipt_entry，其形式是一个链表（位于include/linux/netfilter_ipv4/ip_tables.h，Line122）：

```
struct ipt_entry
{
struct ipt_ip ip;
unsigned int nfcache;
u_int16_t target_offset;
u_int16_t next_offset;
unsigned int comefrom;
struct ipt_counters counters;
unsigned char elems[0];
};
```

其成员如下：

- `struct ipt_ip ip;`：这是对其将要进行匹配动作的IP数据报报头的描述，其定义

于 include/linux/netfilter_ipv4/ip_tables.h，Line122，其成员包括源/目的 IP 及其掩码，出入端口及其掩码，协议族、标志/取反 flag 等信息。

- `unsigned int nfcache;`：HOOK 函数返回的 cache 标识，用以说明经过这个规则后数据报的状态，其可能值有三个，定义于 include/linux/netfilter.h，Line23：

```
#define NFC_ALTERED 0x8000 //已改变
#define NFC_UNKNOWN 0x4000 //不确定
```

另一个可能值是 0，即没有改变。

- `u_int16_t target_offset;`：指出了 target 的数据结构 ipt_entry_target 的起始位置，即从 ipt_entry 的起始地址到 match 存储结束的位置
- `u_int16_t next_offset;`：指出了整条规则的大小，也就是下一条规则的起始地址，即 ipt_entry 的起始地址到 match 偏移再到 target 存储结束的位置
- `unsigned int comefrom;`：所谓的“back pointer”，据引用此变量的代码（主要是 net/ipv4/netfilter/ip_tables.c 中）来看，它应该是指向数据报所经历的上一个规则地址，由此实现对数据报行为的跟踪
- `struct ipt_counters counters;`：说明了匹配这个规则的数据报的计数以及字节计数（定义于 include/linux/netfilter_ipv4/ip_tables.h，Line100）
- `unsigned char elems[0];`：表示扩展的 match 开始的具体位置（因为它是大小不确定的），当然，如果不存在扩展的 match 那么就是 target 的开始位置

ii.扩展 match 的存储结构 ipt_entry_match，位于 include/linux/netfilter_ipv4/ip_tables.h，Line48：

```
struct ipt_entry_match
{
union {
struct {
u_int16_t match_size;
char name[IPT_FUNCTION_MAXNAMELEN];
} user;
struct {
u_int16_t match_size;
struct ipt_match *match;
} kernel;
u_int16_t match_size; //总长度
} u;
unsigned char data[0];
};
```

其中描述match大小的`u_int16_t match_size;`，从涉及这个变量的源码看来，在使用的时候需要注意使用一个宏IPT_ALIGN（位于include/linux/netfilter_ipv4/ip_tables.h，Line445）来进行4的对齐处理（0x3 & 0xfffffffc），这应该是由于match、target扩展后大小的不确定性决定的。

在结构中，用户空间与内核空间为不同的实现，内核空间中的描述拥有更多的信息。在用户空间中存放的仅仅是match的名称，而在内核空间中存放的则是一个指向ipt_match结构的指针

结构ipt_match位于include/linux/netfilter_ipv4/ip_tables.h，Line342：

```
struct ipt_match
{
struct list_head list;
const char name[IPT_FUNCTION_MAXNAMELEN];
int (*match)(const struct sk_buff *skb,
const struct net_device *in,
const struct net_device *out,
const void *matchinfo, // 指向规则中match数据的指针，
// 具体是什么数据结构依情况而定
int offset, // IP数据报的偏移
const void *hdr, // 指向协议头的指针
u_int16_t datalen, // 实际数据长度，即数据报长度-IP头长度
int *hotdrop);
int (*checkentry)(const char *tablename, // 可用的表
const struct ipt_ip *ip,
void *matchinfo,
unsigned int matchinfosize,
unsigned int hook_mask); // 对应HOOK的位图
void (*destroy)(void *matchinfo, unsigned int matchinfosize);
struct module *me;
};
```

其中几个重要成员：

- `int (*match)(……);`：指向用该match进行匹配时的匹配函数的指针，match相关的核心实现。返回0时hotdrop置1，立即丢弃数据报；返回非0表示匹配成功。
- `int (*checkentry)(……);`：当试图插入新的match表项时调用这个指针所指向的函数，对新的match表项进行有效性检查，即检查参数是否合法；如果返回false，规则就不会被接受（譬如，一个TCP的match只会TCP包，而不会接受其它）。
- `void (*destroy)(……);`：当试图删除一个使用这个match的表项时，即模块释放时，调用这个指针所指向的函数。我们可以在checkentry中动态地分配资源，并在destroy中将其释放。

iii.扩展target的存储结构ipt_entry_target，位于include/linux/netfilter_ipv4/ip_tables.h，Line71，这个结构与ipt_entry_match结构类似，同时其中描述内核空间target的结构ipt_target（位于include/linux/netfilter_ipv4/ip_tables.h，Line375）也与ipt_match类似，只不过其中的target()函数返回值不是0/1，而是verdict。

而target的实际使用中，是用一个结构ipt_standard_target专门来描述，这才是实际

的 target 描述数据结构（位于 include/linux/netfilter_ipv4/ip_tables.h，Line94），它实际上就是一个 ipt_entry_target 加一个 verdict。

- 其中成员 verdict 这个变量是一个很巧妙的设计，也是一个非常重要的东东，其值的正负有着不同的意义。我没有找到这个变量的中文名称，在内核开发者的新闻组中称这个变量为“a magic number”。 它的可能值包括 IPT_CONTINUE、IPT_RETURN 以及前文所述的 NF_DROP 等值，那么它的作用是什么呢?
    - 由于 IPTables 是在用户空间中执行的，也就是说 Netfilter/IPTables 这个框架需要用户态与内核态之间的数据交换以及识别。而在具体的程序中，verdict 作为`struct ipt_standard_target`的一个成员，也是对于`struct ipt_entry_target`中的 target()函数的返回值。这个返回值标识的是 target()所对应的执行动作，包括系统的默认动作以及外部提交的自定义动作。
    - 但是，在用户空间中提交的数据往往是类似于“ACCPET”之类的字符串，在程序处理时则是以`#define NF_ACCEPT 1`的形式来进行的；而实际上，以上那些执行动作是以链表的数据结构进行存储的，在内核空间中表现为偏移。
    - 于是，verdict 实际上描述了两个本质相同但实现不同的值：一个是用户空间中的执行动作，另一个则是内核空间中在链表中的偏移——而这就出现了冲突。
    - 解决这种冲突的方法就是：用正值表示内核中的偏移，而用负值来表示数据报的那些默认动作，而外部提交的自定义动作则也是用正值来表示。这样，在实际使用这个 verdict 时，我们就可以通过判断值的正负来进行相应的处理了。
    - 位于 net/ipv4/netfilter/ip_tables.h 中的函数 ipt_do_table()中有一个典型的 verdict 使用（Line335，其中 v 是一个 verdict 的实例）：

```
if (v !=IPT_RETURN) {
verdict = (unsigned)(-v) - 1;
break;
}
```

其中的 IPT_RETURN 定义为：

```
#define IPT_RETURN (-NF_MAX_VERDICT — 1)
```

而宏 NF_MAX_VERDICT 实际上就是：

```
#define NF_MAX_VERDICT NF_REPEAT
```

这样，实际上 IPT_RETURN 的值就是-NF_REPEAT-1，也就是对应 REPEAT，这就是对执行动作的实际描述；而我们可以看到，在下面对 verdict 进行赋值时，它所使用的值是`(unsigned)(-v) — 1`，这就是在内核中实际对偏移进行定位时所使用的值。

那么总之呢，表和规则的实现如图 http://blog.chinaunix.net/photo/24896_061206192551.jpg 所示：

从上图中不难发现，match 的定位如下：

- 起始地址为：当前规则（起始）地址＋sizeof(struct ipt_entry)；
- 结束地址为：当前规则（起始）地址＋ipt_entry->target_offset；
- 每一个 match 的大小为：ipt_entry_match->u.match_size。

target 的定位则为：

- 起始地址为 match 的结束地址，即：当前规则（起始）地址＋ipt_entry-> target_offset；
- 结束地址为下一条规则的起始地址，即：当前规则（起始）地址＋ipt_entry-> next_offset；
- 每一个 target 的大小为：ipt_entry_target->u.target_size。

这些对于理解 match 以及 target 相关函数的实现是很有必要明确的。

同时，include/linux/netfilter_ipv4/ip_tables.h 中提供了三个“helper functions”，可用于使对于 entry、tartget 和 match 的操作变得方便，分别是：

- 函数 ipt_get_target()：Line274，作用是取得 target 的起始地址，也就是上面所说的当前规则（起始）地址＋ipt_entry-> target_offset；
- 宏 IPT_MATCH_ITERATE()：Line281，作用是遍历规则的所有 match，并执行同一个（参数中）给定的函数。其参数为一个 ipt_entry_match 结构和一个函数，以及函数需要的参数。当返回值为 0 时，表示遍历以及函数执行顺利完成；返回非 0 值时则意味着出现问题已终止。
- 宏 IPT_ENTRY_ITERATE()：Line300，作用是遍历一个表中的所有规则，并执行同一个给定的函数。其参数为一个 ipt_entry 结构、整个规则表的大小，以及一个函数和其所需参数。其返回值的意义与宏 IPT_MATCH_ITERATE()类似。
    - 那么如何保证传入的 ipt_entry 结构是整个规则表的第一个结构呢？据源码看来，实际调用这个宏的时候传入的第一个参数都是某个 ipt_table_info 结构的实例所指向的 entries 成员，这样就保证了对整个规则表的完整遍历。

4．规则的使用

当一个特定的 HOOK 被激活时，数据报就开始进入 Netfilter/IPTables 系统进行遍历，首先检查`struct ipt_ip ip`，然后数据报将依次遍历各个 match，也就是`struct ipt_entry_match`，并执行相应的 match 函数，即 ipt_match 结构中的*match 所指向的函数。当 match 函数匹配不成功时返回 0，或者 hotdrop 被置为 1 时，遍历将会停止。

对 match 的遍历完成后，会开始检查`struct ipt_entry_target`，其中如果是一个标准的 target，那么会检查`struct ipt_standard_target`中的 verdict，如果 verdict 值是正的而偏移却指向不正确的位置，那么 ipt_entry 中的 comefrom 成员就有了用武之地——数据报返回所经历的上一个规则。对于非标准的 target 呢，就会调用 target()函数，然后根据其返回值进行后面的处理。

5．规则的扩展

Netfilter/IPTables 提供了对规则进行扩展的机制：可以写一个 LKM 来扩展内核空间的功能，也可以写一个共享库来扩展用户空间中 IPTables 的功能。

1.内核的扩展

要对内核空间的功能进行扩展，实际上就是写一个具有表、match 以及 target 增加功能的模块，相关的函数为（位于 net/ipv4/netfilter/ip_tables.c，Line1318 to 1444）：

- ipt_register_table()、ipt_unregister_table()，参数为 struct ipt_table

*。

- ipt_register_table()函数是这三对函数中最复杂的一个，涉及了内存、信号量等方方面面的东西，但总起来说就做了初始化表以及加入双向链表两件事。
- 其复杂一是因为涉及到多CPU的处理（每个CPU拥有各自独立的“规则空间”），需要首先将新的entries放入第一个CPU空间，在检查完毕后再复制到其他CPU中；二是就是上面所说对新table各个entry的检查，包括边界检查以及完整性检查等。
- 其中的重要函数有这么几个：
  - translate_table()（位于net/ipv4/netfilter/ip_tables.c，Line797）：这个函数的主要作用是检查并应用用户空间传来的规则：
    1. 对新表进行边界检查（由宏IPT_ENTRY_ITERATE()调用函数check_entry_size_and_blocks()，位于net/ipv4/netfilter/ip_tables.c，Line732），包括对齐、过大过小等，特别是保证赋给hook_entries和underflows值的正确性。
    2. 调用函数make_source_chains()（位于net/ipv4/netfilter/ip_tables.c，Line499）检查新的表中是否存在规则环，同时将HOOK的规则遍历顺序存入comefrom变量。（这个函数我没有仔细看，只是大概略了一下）
    3. 对ipt_entry依次进行ipt_ip头、match以及target的完整性检查（由宏IPT_ENTRY_ITERATE()调用函数check_entry()，位于net/ipv4/netfilter/ip_tables.c，Line676），保证ipt_entry的正确性。
    4. 将正确的 ipt_tables复制给其他的CPU。

    这个函数另外还在do_replace()函数（仅在一个源码中没有被调用过的函数中被调用，不予分析）中被调用。
  - replace_table()（位于net/ipv4/netfilter/ip_tables.c，Line877）：这个函数的主要作用是：将得到模块初始值的ipt_table_info结构(newinfo)中的值传给ipt_table中的private，并返回ip_table中旧的private。
  - list_prepend()（位于include/linux/netfilter_ipv4/listhelp.h，Line75）：在这个函数被调用之前，整个初始化的过程就已经结束了，这个函数的主要作用是：互斥地调用Linux源码中的list_add()函数（位于include/linux/list.h，Line55），将新的table加入到双向链表之中。
- ipt_register_match()、ipt_unregister_match()，参数为struct ipt_match *。
- ipt_register_target()、ipt_unregister_target()，参数为struct ipt_target *。

这三对函数除了ipt_register_table()外的5个函数主要就是互斥地将table/match/target加入到双向链表中或者从双向链表中删除。

其中向双向链表中加入新节点是通过调用list_named_insert()函数（位于include/linux/netfilter_ipv4/listhelp.h，Line101）实现的。这个函数的主要工作是首先确定待插入的match名字是否已经存在，只有不存在时才进行插入的操作。

2.用户空间的扩展

用户空间中的扩展用的是共享库配合libiptc库的机制，但这种机制是在单独的IPTbales程序中提供的，内核源码中并没有提供，这里就不做分析了。

五、数据报过滤模块——filter表

1. 概述

filter 表的功能仅仅是对数据报进行过滤，并不对数据报进行任何的修改。

filter 模块在 Netfilter 中是基于下列 HOOK 点的：

- NF_IP_LOCAL_IN
- NF_IP_FORWARD
- NF_IP_LOCAL_OUT

这几个 HOOK 分别对应着 filter 表中的 INPUT、FORWARD、OUTPUT 三条规则链，对于任何一个数据报都会经过这 3 个 HOOK 之一。

filter 模块的接口位于文件 net/ipv4/netfilter/iptables_filter.c 中。

2. filter 表的定义和初始化

filter 表是前面所述数据结构 ipt_table 的一个实例，它的定义和初始化位于 net/ipv4/netfilter/iptable_filter.c，Line84：

```
static struct ipt_table packet_filter
= { { NULL, NULL }, "filter", &initial_table.repl,
FILTER_VALID_HOOKS, RW_LOCK_UNLOCKED, NULL, THIS_MODULE };
```

对照结构 ipt_table 的定义，我们可以发现，filter 表的初始化数据为：

- 链表初始化为空
- 表名为 filter
- 初始化的模板为&initial_table.repl
    - 初始化的模板表定义于 net/ipv4/netfilter/iptable_filter.c，Line30，是一个很简单的数据结构，只是赋值有些复杂，因为要对所涉及的各个 HOOK 进行不同的处理：

```
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"
}
}
```

```
};
```

我们可以看到，一个 initial_table 包含三个成员：

- `struct ipt_replace repl;`：是对一个表进行初始化的最主要部分，这个 ipt_replace 结构在前面已经分析过了；
- `struct ipt_standard entries[3];`：是对这个表所监听的各个 HOOK 上对应的初始化信息，实际上就是一个 ipt_entry 结构加一个 ipt_standard_target 结构；
- `struct ipt_error term;`：是这个表出错时对应的信息，实际上就是一个 ipt_entry 结构、一个 ipt_entry_target 结构再加一个 errorname。

- 当前表所监听的 HOOK 位图为 FILTER_VALID_HOOKS，位于 net/ipv4/netfilter/iptable_filter.c，Line9：

```
#define FILTER_VALID_HOOKS ((1 << NF_IP_LOCAL_IN) | (1 << NF_IP_FORWARD) | (1 << NF_IP_LOCAL_OUT))
```

  我们可以看到，实际上就是 IN，FORWARD 和 OUT。
- 读写锁为 RW_LOCK_UNLOCKED，即为打开状态
- 实际数据区 ipt_table_info 为空
- 定义为模块

3．filter 表的实现

filter 表的实现函数实际上就是模块 iptable_filter.o 的 init 函数，位于 net/ipv4/netfilter/iptable_filter.c，Line128。其主要工作是首先通过 ipt_register_table()函数进行表的注册，然后用 nf_register_hook()函数注册表所监听的各个 HOOK。

其中，对 HOOK 进行注册时，是通过对数据结构 nf_hook_ops 的一个实例 ipt_ops 进行操作来实现的，这个实例的定义及初始化位于 net/ipv4/netfilter/iptable_filter.c，Line117：

```
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 }
};
```

对应前面所分析 nf_hook_ops 的各个成员，不难确定这些初始化值的意义。

其中，对应 IN 和 FORWARD 的处理函数均为 ipt_hook，OUT 的处理函数则为

ipt_local_out_hook，下面依次分析之：

- ipt_hook，定义于net/ipv4/netfilter/iptable_filter.c，Line89：

```
static unsigned int
ipt_hook(unsigned int hook,
struct sk_buff **pskb,
const struct net_device *in,
const struct net_device *out,
int (*okfn)(struct sk_buff *))
{
return ipt_do_table(pskb, hook, in, out, &packet_filter, NULL);
}
```

实际上它就是调用了ipt_do_table()函数，也就是说，注册时首先注册一个ipt_hook()函数，然后ipt_hook()通过调用ipt_do_table()函数对传入的数据进行真正的处理。下面我们来看一下ipt_do_table()这个函数：

它位于net/ipv4/netfilter/ip_tables.c，Line254，是一个很长的函数，其主要功能是对数据报进行各种匹配、过滤（包括基本规则、matches以及target），具体些说，其工作大致为：

1. 初始化各种变量，如IP头、数据区、输入输出设备、段偏移、规则入口及偏移量等等；
2. 进行规则的匹配，首先调用ip_packet_match()函数（位于net/ipv4/netfilter/ip_tables.c，Line121）确定IP数据报是否匹配规则，若不匹配则跳到下一条规则（这个函数的主要工作大致为：依次处理源/目的IP地址、输入输出接口，然后对基本的规则进行匹配）；
3. 如果数据报匹配，则下面开始继续匹配matches和target，首先利用宏IPT_MATCH_ITERATE调用do_match()函数（下面单独分析）对扩展的match进行匹配，若不匹配则跳到下一条规则；
4. 扩展match匹配后，首先调用ipt_get_target()获得target的偏移地址，然后对target进行匹配，这个匹配的过程要比match的匹配过程复杂一些，同样在下面单独分析。

下面首先来分析do_match()函数，它位于net/ipv4/netfilter/ip_tables.c，Line229，它的实现只有一个if语句：

```
if (!m->u.kernel.match->match(skb, in, out, m->data, offset, hdr, datalen,
hotdrop))
return 1;
else
return 0;
```

其中的`m->u.kernel.match->match(skb, in, out, m->data, offset, hdr, datalen, hotdrop)`是用来定位match的。

因为如果仅仅是根据match的名字遍历链表来进行查找的话，效率会非常低下。Netfilter源码中采用的方法是在进行match的检测之前，也就是在ipt_register_table()函数中通过translate_table()函数由宏IPT_ENTRY_ITERATE调用函数check_entry()时，在check_entry()中通过宏IPT_MATCH_ITERATE调用了check_match()函数（位于net/ipv4/netfilter/ip_tables.c，Line640），在这个函数中，有一个对m->u.kernel.match的赋值：

```
m->u.kernel.match = match;
```

这样，每条规则的u.kernel.match 就与内核模块中的struct ipt_match链表关联了起来，也就是说，这样一来，根据match的名字，其对应的match函数就与链表中对应的函数关联了起来。于是，上面的那条定位match的语句的意义也就开始明了了：

利用宏IPT_MATCH_ITERATE来遍历规则中的所有mach，然后直接调用`m->u.kernel.match->match`来进行对数据报的匹配工作——这样的效率显然要比简单的遍历要高许多。

然后我们来看一下对target的匹配，从数据结构的实现上看，似乎这个过程与match的匹配应该是相似的，但实际上target存在标准的和非标准的两种，其中标准的target与非标准的target的处理是不一样的。在这里我遇到了问题，如下：

首先，在Netfilter的源码中，存在两个ipt_standard_target，其中一个是一个struct，位于include/linux/netfilter_ipv4/ip_tables.h，Line94；另一个是`struct ipt_target`的一个实例，位于net/ipv4/netfilter/IPtables.c，Line1684，而在target的匹配过程中，它是这样处理的（ipt_do_tables()，net/ipv4/netfilter/ip_tables.c，Line329）：

```
/* Standard target? */
if (!t->u.kernel.target->target) {……}
```

从这里看来，它应该是当 t->u.kernel.target的target函数为空时，表明其为标准的target。那么结合上述两者的定义，似乎用的是后者，因为后者的定义及初始化如下：

```
/* The built-in targets: standard (NULL) and error. */
static struct ipt_target ipt_standard_target
= { { NULL, NULL }, IPT_STANDARD_TARGET, NULL, NULL, NULL };
```

但是问题出现在：初始化中的IPT_STANDARD_TARGET被定义为”“！！并且在整个源码中，用到实例化的ipt_standard_target的地方仅有两处，即上面的这个定义以及ip_tables.c中将ipt_standard_target加入到target链表之中。也就是说这个实例的名字一直为空，这一点如何理解？

- ipt_local_out_hook，定义于net/ipv4/netfilter/iptable_filter.c，Line99其功能与ipt_hook()相似，只不过为了防止DOS攻击而增加了对ratelimit的检查。

这样，到这里，filter 表的实现已经分析完毕，至于具体的过滤功能如何实现，那就是每个 HOOK 处理函数的问题了。

六、连接跟踪模块（Conntrack）

1．概述

连接跟踪模块是 NAT 的基础，但作为一个单独的模块实现。它用于对包过滤功能的一个扩展，管理单个连接（特别是 TCP 连接），并负责为现有的连接分配输入、输出和转发 IP 数据报，从而基于连接跟踪来实现一个“基于状态”的防火墙。

当连接跟踪模块注册一个连接建立包之后，将生成一个新的连接记录。此后，所有属于此连接的数据报都被唯一地分配给这个连接。如果一段时间内没有流量而超时，连接将被删除，然后其他需要使用连接跟踪的模块就可以重新使用这个连接所释放的资源了。

如果需要用于比传输协议更上层的应用协议，连接跟踪模块还必须能够将建立的数据连接与现有的控制连接相关联。

Conntrack 在 Netfilter 中是基于下列 HOOK 的：

- NF_IP_PRE_ROUTING
- NF_IP_LOCAL_OUT
- 同时当使用 NAT 时，Conntrack 也会有基于 NF_IP_LOCAL_IN 和 NF_IP_POST_ROUTING 的，只不过优先级很小。

在所有的 HOOK 上，NF_IP_PRI_CONNTRACK 的优先级是最高的（-200），这意味着每个数据报在进入和发出之前都首先要经过 Conntrack 模块，然后才会被传到钩在 HOOK 上的其它模块。

Conntrack 模块的接口位于文件 net/ipv4/netfilter/ip_conntrack_standalone.c 中。

2．连接状态的管理

1.多元组

在连接跟踪模块中，使用所谓的“tuple”，也就是多元组，来小巧锐利地描述连接记录的关键部分，主要是方便连接记录的管理。其对应的数据结构是 ip_conntrack_tuple（位于 include/linux/netfilter_ipv4/ip_conntrack_tuple.h，Line38）：

```
struct ip_conntrack_tuple
{
struct ip_conntrack_manip src;
struct {
u_int32_t ip;
```

```
union {
u_int16_t all;
struct { u_int16_t port; } tcp;
struct { u_int16_t port; } udp;
struct { u_int8_t type, code; } icmp;
} u;
u_int16_t protonum;
} dst;
};
```

从它的定义可以看出，一个多元组实际上包括两个部分：一是所谓的“unfixed”部分，也就是源地址以及端口；二是所谓的“fixed”部分，也就是目的地址、端口以及所用的协议。这样，连接的两端分别用地址+端口，再加上所使用的协议，一个 tuple 就可以唯一地标识一个连接了（对于没有端口的 icmp 协议，则用其它东东标识）。

2.连接记录

那么真正的完整连接记录则是由数据结构 ip_conntrack（位于 include/linux/netfilter_ipv4/ip_conntrack.h，Line160）来描述的，其成员有：

- `struct nf_conntrack ct_general;`：nf_conntrack 结构定义于 include/linux/skbuff.h，Line89，其中包括一个计数器 use 和一个 destroy 函数。计数器 use 对本连接记录的公开引用次数进行计数
- `struct ip_conntrack_tuple_hash tuplehash[IP_CT_DIR_MAX];`：其中的 IP_CT_DIR_MAX 是一个枚举类型 ip_conntrack_dir（位于 include/linux/netfilter_ipv4/ip_conntrack_tuple.h，Line65）的第 3 个成员，从这个结构实例在源码中的使用看来，实际上这是定义了两个 tuple 多元组的 hash 表项 tuplehash[IP_CT_DIR_ORIGINAL/0]和 tuplehash[IP_CT_DIR_REPLY/1]，利用两个不同方向的 tuple 定位一个连接，同时也可以方便地对 ORIGINAL 以及 REPLY 两个方向进行追溯
- `unsigned long status;`：这是一个位图，是一个状态域。在实际的使用中，它通常与一个枚举类型 ip_conntrack_status（位于 include/linux/netfilter_ipv4/ip_conntrack.h，Line33）进行位运算来判断连接的状态。其中主要的状态包括：
    - IPS_EXPECTED(_BIT)，表示一个预期的连接
    - IPS_SEEN_REPLY(_BIT)，表示一个双向的连接
    - IPS_ASSURED(_BIT)，表示这个连接即使发生超时也不能提早被删除
    - IPS_CONFIRMED(_BIT)，表示这个连接已经被确认（初始包已经发出）

- `struct timer_list timeout;`：其类型 timer_list 位于 include/linux/timer.h，Line11，其核心是一个处理函数。这个成员表示当发生连接超时时，将调用此处理函数
- `struct list_head sibling_list;`：所谓“预期的连接”的链表，其中存放的是我们所期望的其它相关连接
- `unsigned int expecting;`：目前的预期连接数量
- `struct ip_conntrack_expect *master;`：结构 ip_conntrack_expect 位于 include/linux/netfilter_ipv4/ip_conntrack.h，Line119，这个结构用于将一个预期的连接分配给现有的连接，也就是说本连接是这个 master 的一个预期连接
- `struct ip_conntrack_helper *helper;`：helper 模块。这个结构定义于 include/linux/netfilter_ipv4/ip_conntrack_helper.h，Line11，这个模块提供了一个可以用于扩展 Conntrack 功能的接口。经过连接跟踪 HOOK 的每个数据报都将被发给每个已经注册的 helper 模块（注册以及卸载函数分别为 ip_conntrack_helper_register()以及 ip_conntrack_helper_unregister()，分别位于 net/ipv4/netfilter/ip_conntrack_core.c，Line1136、1159）。这样我们就可以进行一些动态的连接管理了
- `struct nf_ct_info infos[IP_CT_NUMBER];`：一系列的 nf_ct_info 类型（定义于 include/linux/skbuff.h ，Line92，实际上就是 nf_conntrack 结构）的结构，每个结构对应于某种状态的连接。这一系列的结构会被 sk_buff 结构的 nfct 指针所引用，描述了所有与此连接有关系的数据报。其状态由枚举类型 ip_conntrack_info 定义（位于 include/linux/netfilter_ipv4/ip_conntrack.h，Line12），共有 5 个成员：
    - IP_CT_ESTABLISHED：数据报属于已经完全建立的连接
    - IP_CT_RELATED： 数据报属于一个新的连接，但此连接与一个现有连接相关（预期连接）；或者是 ICMP 错误
    - IP_CT_NEW：数据报属于一个新的连接
    - IP_CT_IS_REPLY：数据报属于一个连接的回复
    - IP_CT_NUMBER：不同 IP_CT 类型的数量，这里为 7，NEW 仅存于一个方向上
- 为 NAT 模块设置的信息（在条件编译中）

3. hash 表

Netfilter 使用一个 hash 表来对连接记录进行管理，这个 hash 表的初始指针为

*ip_conntrack_hash，位于net/ipv4/netfilter/ip_conntrack_core.c，Line65，这样我们就可以使用ip_conntrack_hash[num]的方式来直接定位某个连接记录了。

而hash表的每个表项则由数据结构ip_conntrack_tuple_hash（位于include/linux/netfilter_ipv4/ip_conntrack_tuple.h，Line86）描述：

```
struct ip_conntrack_tuple_hash
{
struct list_head list;
struct ip_conntrack_tuple tuple;
struct ip_conntrack *ctrack;
};
```

可见，一个hash表项中实质性的内容就是一个多元组ip_conntrack_tuple；同时还有一个指向连接的ip_conntrack结构的指针；以及一个链表头（这个链表头不知是干嘛的）。

3．连接跟踪的实现

有了以上的数据结构，连接跟踪的具体实现其实就非常简单而常规了，无非就是初始化、连接记录的创建、在连接跟踪hash表中搜索并定位数据报、将数据报转换为一个多元组、判断连接的状态以及方向、超时处理、协议的添加、查找和注销、对不同协议的不同处理、以及在两个连接跟踪相关的HOOK上对数据报的处理等。

下面重点说明一下我在分析中遇到的几个比较重要或者比较难理解的地方：

- 所谓“预期链接”

可以将预期连接看作父子关系来理解，如图http://blog.chinaunix.net/photo/24896_061206192612.jpg

- ip_conntrack的状态转换

ip_conntrack的状态转换分两种，同样用图来描述。首先是正常的状态转换，如图http://blog.chinaunix.net/photo/24896_061206192631.jpg，

然后是ICMP error时的状态转换（由函数icmp_error_track()来判断，位于net/ipv4/netfilter/ip_conntrack_core.c，Line495），

如图http://blog.chinaunix.net/photo/24896_061206192648.jpg

- 在NF_IP_PRE_ROUTING上对分片IP数据报的处理

在经过HOOK中的NF_IP_PRE_ROUTING时（函数ip_conntrack_in()，位于net/ipv4/netfilter/ip_conntrack_core.c，Line796），由于外来的数据报有可能是经过分片的，所以必须对分片的情形进行处理，将IP数据报组装后才能分配给连接。

具体的操作是首先由函数ip_ct_gather_frags()对分片的数据报进行收集，然后调用ip_defrag()函数（位于net/ipv4/ip_fragment.c，Line632）组装之

4．协议的扩展

由于我们可能要添加新的协议，所以单独对协议的扩展进行分析。

各种协议使用一个全局的协议列表存放，即 protocol_list（位于 include/linux/netfilter_ipv4/ip_conntrack_core.h，Line21），使用结构 ip_conntrack_protocol（位于 include/linux/netfilter_ipv4/ip_conntrack_protocol.h，Line6）来表示：

```
struct ip_conntrack_protocol
{
struct list_head list;
u_int8_t proto; //协议号
const char *name;
int (*pkt_to_tuple)(const void *datah, size_t datalen,
struct ip_conntrack_tuple *tuple);
int (*invert_tuple)(struct ip_conntrack_tuple *inverse,
const struct ip_conntrack_tuple *orig);
unsigned int (*print_tuple)(char *buffer,
const struct ip_conntrack_tuple *);
unsigned int (*print_conntrack)(char *buffer,
const struct ip_conntrack *);
int (*packet)(struct ip_conntrack *conntrack,
struct iphdr *iph, size_t len,
enum ip_conntrack_info ctinfo);
int (*new)(struct ip_conntrack *conntrack, struct iphdr *iph,
size_t len);
void (*destroy)(struct ip_conntrack *conntrack);
int (*exp_matches_pkt)(struct ip_conntrack_expect *exp,
struct sk_buff **pskb);
struct module *me;
};
```

其中重要成员：

- `int (*pkt_to_tuple)(……)`：其指向函数的作用是将协议加入到 ip_conntrack_tuple 的 dst 子结构中
- `int (*invert_tuple)(……)`：其指向函数的作用是将源和目的多元组中协议部分的值进行互换，包括 IP 地址、端口等
- `unsigned int (*print_tuple)(……)`：其指向函数的作用是打印多元组中的协议信息
- `unsigned int (*print_conntrack)(……)`：其指向函数的作用是打印整个连接记录
- `int (*packet)(……)`：其指向函数的作用是返回数据报的 verdict 值
- `int (*new)(……)`：当此协议的一个新连接发生时，调用其指向的这个函数，调用返回 true 时再继续调用 packet()函数

- `int (*exp_matches_pkt)(……)`：其指向函数的作用是判断是否有数据报匹配预期的连接

添加/删除协议使用的是函数 ip_conntrack_protocol_register()以及 ip_conntrack_protocol_unregister()，分别位于 net/ipv4/netfilter/ip_conntrack_standalone.c，Line298 & 320，其工作就是将 ip_conntrack_protocol 添加到全局协议列表 protocol_list。

七、网络地址转换模块（Network Address Translation）

1．概述

网络地址转换的机制一般用于处理 IP 地址转换，在 Netfilter 中，可以支持多种 NAT 类型，而其实现的基础是连接跟踪。

NAT 可以分为 SNAT 和 DNAT，即源 NAT 和目的 NAT，在 Netfilter 中分别基于以下 HOOK：

- NF_IP_PRE_ROUTING：可以在这里定义 DNAT 的规则，因为路由器进行路由时只检查数据报的目的 IP 地址，所以为了使数据报得以正确路由，我们必须在路由之前就进行 DNAT
- NF_IP_POST_ROUTING：可以在这里定义 SNAT 的规则，系统在决定了数据报的路由以后在执行该 HOOK 上的规则
- NF_IP_LOCAL_OUT：定义对本地产生的数据报的 DNAT 规则
- CONFIG_IP_NF_NAT_LOCAL 定义后，NF_IP_LOCAL_IN 上也可以定义 DNAT 规则。

同时，MASQUERADE（伪装）是 SNAT 的一种特例，它与 SNAT 几乎一样，只有一点不同：如果连接断开，所有的连接跟踪信息将被丢弃，而去使用重新连接以后的 IP 地址进行 IP 伪装；而 REDIRECT（重定向）是 DNAT 的一种特例，这时候就相当于将符合条件的数据报的目的 IP 地址改为数据报进入系统时的网络接口的 IP 地址。

2．基于连接跟踪的相关数据结构

NAT 是基于连接跟踪实现的，NAT 中所有的连接都由连接跟踪模块来管理，NAT 模块的主要任务是维护 nat 表和进行实际的地址转换。这样，我们来回头重新审视一下连接跟踪模块中由条件编译决定的部分。

首先，是连接的描述 ip_conntrack，在连接跟踪模块部分中提到，这个结构的最后有“为 NAT 模块设置的信息”，即：

```
#ifdef CONFIG_IP_NF_NAT_NEEDED
struct {
struct ip_nat_info info;
union ip_conntrack_nat_help help;
#if defined(CONFIG_IP_NF_TARGET_MASQUERADE) || \
defined(CONFIG_IP_NF_TARGET_MASQUERADE_MODULE)
int masq_index;
#endif
} nat;
#endif /* CONFIG_IP_NF_NAT_NEEDED */
```

这是一个叫做 nat 的子结构，其中有 3 个成员：

- 一个 ip_nat_info 结构，这个结构下面会具体分析
- 一个 ip_conntrack_nat_help 结构，是一个空结构，为扩展功能而设
- 一个为伪装功能而设的 index，从源码中对这个变量的使用看来，是对应所伪装网络接口的 ID，也就是 net_device 中的 ifindex 成员

好，下面我们来看一下这个 ip_nat_info 结构，这个结构存储了连接中的地址绑定信息，其定义位于 include/linux/netfilter_ipv4/ip_nat.h，Line98：

```
struct ip_nat_info
{
int initialized;
unsigned int num_manips;
struct ip_nat_info_manip manips[IP_NAT_MAX_MANIPS];
const struct ip_nat_mapping_type *mtype;
struct ip_nat_hash bysource, byipsproto;
struct ip_nat_helper *helper;
struct ip_nat_seq seq[IP_CT_DIR_MAX];
};
```

- `int initialized;`：这是一个位图，表明源地址以及目的地址的地址绑定是否已被初始化
- `unsigned int num_manips;`：这个成员指定了存放在下面的 manip 数组中的可执行操作的编号，在不同的 HOOK 以及不同的方向上，可执行操作是分别进行计数的。
- `struct ip_nat_info_manip manips[IP_NAT_MAX_MANIPS];`：ip_nat_info_manip 结构定义于 include/linux/netfilter_ipv4/ip_nat.h，Line66，一个 ip_nat_info_manip 结构对应着一个可执行操作或地址绑定，其成员包括方向（ORIGINAL 以及 REPLY）、HOOK 号、操作类型（由一个枚举类型 ip_nat_manip_type 定义，有 IP_NAT_MANIP_SRC 和 IP_NAT_MANIP_DST 两种）以及一个 ip_conntrack_manip 结构
- `const struct ip_nat_mapping_type *mtype;`：ip_nat_mapping_type 这个结构在整个内核源码中都没有定义，根据注释来看应该也是一个预留的扩展，一般就是 NULL
- `struct ip_nat_hash bysource, byipsproto;`：ip_nat_hash 结构定义于 include/linux/netfilter_ipv4/ip_nat.h，Line89，实际上就是一个带表头的 ip_conntrack 结构，跟连接跟踪中 hash 表的实现类似。其中，
  - bysource 表是用来管理现有连接的
  - byipsproto 表则管理已经完成的转换映射，以保证同一个 IP 不会同时有两个

映射，避免地址转换冲突

- `struct ip_nat_helper *helper;`：扩展用
- `struct ip_nat_seq seq[IP_CT_DIR_MAX];`：这是为每一个方向（其实就两个方向）记录一个序列号。ip_nat_seq 结构定义于 include/linux/netfilter_ipv4/ip_nat.h，Line33，这个结构用得并不多，应该是用于 TCP 连接的计数和对涉及 TCP 的修改的定位

3．nat 表的实现

nat 表的初始化和实现与 filter 极为相似。

在初始化上，其初始化位于 net/ipv4/netfilter/ip_nat_rule.c，Line104，初始化所用模板则位于 net/ipv4/netfilter/ip_nat_rule.c，Line50。

在实现上，其实现函数就是 NAT 模块的初始化函数 init_or_cleanup()（位于 net/ipv4/netfilter/ip_nat_standalone.c，Line278）。其工作主要是依次调用 ip_nat_rule_init()、ip_nat_init()以及 nf_register_hook()。

1. 首先 ip_nat_rule_init()（位于 net/ipv4/netfilter/ip_nat_rule.c，Line278）调用 ipt_register_table()来初始化并注册 nat 表，然后利用 ipt_register_target()来初始化并注册 SNAT 和 DNAT，在这个注册过程中，关键的函数是 ip_nat_setup_info（位于 net/ipv4/netfilter/ip_nat_core.c，Line511），其工作是:
    - 首先调用 invert_tuple()（net/ipv4/netfilter/ip_conntrack_core.c，Line879），将记录反转
    - 然后调用 get_unique_tuple()（net/ipv4/netfilter/ip_nat_core.c，Line393，在指定的地址范围（ip_nat_multi_range 结构）中查找空闲的地址），如果没有空闲地址可用则会返回 NF_DROP
    - 判断源和目的是否改变，如果改变，则更新 ip_nat_info。
2. 然后 ip_nat_init()（位于 net/ipv4/netfilter/ip_nat_core.c，Line953）会给 nat 所用的两个 hash 表（bysource、byipsproto）分配空间并初始化各种协议
3. 最后会通过 nf_register_hook()注册相应 HOOK 的函数 ip_nat_fn()、ip_nat_local_fn()和 ip_nat_out()，并增加连接跟踪的计数器。

在具体的 HOOK 函数实现上，后两者其实都是基于 ip_nat_fn()的，而这其中最重要的处

理函数，也就是实际的网络地址转换函数是 do_bindings()，下面将对其进行分析：

do_bindings()位于 net/ipv4/netfilter/ip_nat_core.c，Line747，其主要工作是将 ip_nat_info 中的地址绑定应用于数据报：

1. 它首先在 ip_nat_info->manip 数组中查找能够匹配的绑定
2. 然后调用 manip_pkt()函数（位于 net/ipv4/netfilter/ip_nat_core.c，Line701）进行相应的地址转换：
   - manip_pkt()这个函数会根据不同的方向进行转换，并且对校验和进行处理
   - 同时，它是递归调用自己以处理不同协议的情况
3. 最后调用 helper 模块进行执行（当然，如果有的话），特别是避免在同一个数据报上执行多次同一个 helper 模块

nat 表与 filter 表还有一个较大的不同：在一个连接中，只有第一个数据报才会经过 nat 表，而其转换的结果会作用于此连接中的其它所有数据报。

4．协议的扩展

要想扩展 NAT 的协议，那么必须写入两个模块，一个用于连接跟踪，一个用于 NAT 实现。

与连接跟踪类似，nat 中协议也由一个列表 protos 存放，位于 include/linux/netfilter_ipv4/ip_nat_core.h，Line17。协议的注册和注销分别是由函数 ip_nat_protocol_register()和 ip_nat_protocol_unregister()实现的，位于 net/ipv4/netfilter/ip_nat_standalone.c，Line242。

nat 的协议是由结构 ip_nat_protocol 描述的，其定义位于 include/linux/netfilter_ipv4/ip_nat_protocol.h，Line10：

```
struct ip_nat_protocol
{
struct list_head list;
const char *name;
unsigned int protonum;
void (*manip_pkt)(struct iphdr *iph, size_t len,
const struct ip_conntrack_manip *manip,
enum ip_nat_manip_type maniptype);
int (*in_range)(const struct ip_conntrack_tuple *tuple,
enum ip_nat_manip_type maniptype,
const union ip_conntrack_manip_proto *min,
```

```
const union ip_conntrack_manip_proto *max);
int (*unique_tuple)(struct ip_conntrack_tuple *tuple,
const struct ip_nat_range *range,
enum ip_nat_manip_type maniptype,
const struct ip_conntrack *conntrack);
unsigned int (*print)(char *buffer,
const struct ip_conntrack_tuple *match,
const struct ip_conntrack_tuple *mask);
unsigned int (*print_range)(char *buffer,
const struct ip_nat_range *range);
};
```

其中重要成员：

- `void (*manip_pkt)(……)`：其指向的函数会根据 ip_nat_info->manip 参数进行数据报的转换，即 do_bindings()中调用的 manip_pkt()函数
- `int (*in_range)(……)`：其指向的函数检查多元组的协议部分值是否在指定的区间之内
- `int (*unique_tuple)(……)`：其指向函数的作用是根据 manip 的类型修改多元组中的协议部分，以获得一个唯一的地址，多元组的协议部分被初始化为 ORIGINAL 方向

八、数据报修改模块——mangle 表

1．概述

mangle 这个词的原意是撕裂、破坏，这里所谓“packet mangling”是指对 packet 的一些传输特性进行修改。mangle 表被用来真正地对数据报进行修改，它可以在所有的 5 个 HOOK 上进行操作。

从源码看来，在 mangle 表中所允许修改的传输特性目前有：

- TOS（服务类型）：修改 IP 数据报头的 TOS 字段值
- TTL（生存时间）：修改 IP 数据报头的 TTL 字段值
- MARK：修改 skb 的 nfmark 域设置的 nfmark 字段值。
    - nfmark 是数据报的元数据之一，是一个用户定义的数据报的标记，可以是 unsigned long 范围内的任何值。该标记值用于基于策略的路由，通知 ipqmpd（运行在用户空间的队列分拣器守护进程）将该数据报排队给哪个进程

等信息。

- TCP MSS（最大数据段长度）：修改 TCP 数据报头的 MSS 字段值

2．mangle 表的实现

遍历整个源码，没有发现 mangle 表的独有数据结构。

mangle 表的初始化和实现与 filter 极为相似。在初始化上，其初始化位于 net/ipv4/netfilter/iptable_mangle.c，Line117，初始化所用模板则位于 net/ipv4/netfilter/iptable_mangle.c，Line43。

在实现上，其实现函数就是 mangle 模块的初始化函数 init()（位于 net/ipv4/netfilter/iptable_mangle.c，Line183）。其工作就是依次注册 packet_mangler 表以及 5 个 HOOK。其中 NF_IP_LOCAL_OUT 的处理函数为 ipt_local_hook()，其余均为 ipt_route_hook()，分别位于 net/ipv4/netfilter/iptable_mangle.c，Line132 & 122，二者的关键实现都是通过调用 ipt_do_table()实现的。

3．数据报的修改

对数据报不同位的修改都是通过单独的模块实现的，也就是说由 ipt_tos.o、ipt_ttl.o、ipt_mark.o、ipt_tcpmss.o 实现上面所说的四种传输特性的修改。

以 TOS 为例，其涉及的文件为 net/ipv4/netfilter/ipt_tos.c 以及 include/linux/netfilter_ipv4/ipt_tos.h。

其专有数据结构为 ipt_tos_info，定义于 ipt_tos.h 中：

```
struct ipt_tos_info {
u_int8_t tos;
u_int8_t invert;
};
```

其模块的添加/卸载函数很简单，其实就是添加/删除 TOS 的 MATCH：tos_match（定义并初始化于 ipt_tos.c，Line37）：

```
static struct ipt_match tos_match
= { { NULL, NULL }, "tos", &match, &checkentry, NULL, THIS_MODULE };
```

而在 tos_match 的处理函数 match()中，已经完成了对相应位的赋值，这似乎是违反 match 仅仅匹配而不修改的一个特例。

九、其它高级功能模块

Netfilter 中还有一些其它的高级功能模块，基本是为了用户操作方便的，没有对它们进行分析，如：

- REJECT，丢弃包并通知包的发送者，同时返回给发送者一个可配置的 ICMP 错误信息，由 ipt_REJECT.o 完成
- MIRROR，互换源和目的地址以后并重新发送，由 ipt_MIRROR.o 完成
- LOG， 将匹配的数据报传递给系统的 syslog()进行记录，由 ipt_LOG.o 完成
- ULOG，Userspace logging，将数据报排队转发到用户空间中，将匹配的数据适用用户空间的 log 进程进行记录，由 ip_ULOG.o 完成。这是 Netfilter 的一个关键技术，可以使用户进程可以进行复杂的数据报操作，从而减轻内核空间中的复杂度
- Queuing，这是上面 ULOG 技术的基础，由 ip_queue.o 完成，提供可靠的异步包处理以及性能两号的 libipq 库来进行用户空间数据报操作的开发。
- 等等……

P.S. 字体好乱啊。。。要是 CUBlog 支持直接导入 odt 文档就好了。。。粘过来就大变样。。。-_-!