Netfilter ALG 读书笔记

fist<fist_hust@sina.com.cn>

初稿：2004-07-11

目录

目录.....1
前言.....1
什么是 ALG?.....1
Netfilter 中 ALG 的组成部分.....1
ALG 的框架.....2
1． 主要文件.....2
2．如何 track connection?.....2
3. 如何 expected and related?.....3
4．如何 nat mangling?.....11
ALG Conntrack Helper 的构成.....42
ALG Nat helper 的构成.....45
可改进的之处.....47

前言

本文以 NAT 中的应用层网关为线索，以 ftp ALG 为例子，分析了 netfilter 中 NAT 部分的实现。Netfilter 虽然集成了 Firewall，NAT 和 mangel 的功能，但本人觉得 NAT 最为复杂。当理解了 NAT 部分的功能，再看 Firewall 和 mangle 的功能，会有“会当临绝顶，一览众山小”的感觉。

曾经在网络上搜索过 netfilter 的资料，其中以 iptables 的用法和 netfilter 的框架描述比较多。因此，本文没有花篇幅介绍这些，希望读者对 iptables 有些了解，至少可以 man iptables. 另外了解 netfilter 有 5 个 hook 点，能完成 NAT，firewall，mangle 等功能。网络上分析 netfilter 源码的文章比较少，希望能够对需要了解 netfilter 实现的人有帮助。

什么是 ALG?

ALG---Application Layer Gateway.一种为了解决 NAT 问题，针对不同应用程序的内核补丁。不同的应用，具有不同的 ALG，常见的 ALG 有 ftp ALG, irc ALG 等。

Netfilter 中 ALG 的组成部分

在 Linux 系统中，netfilter 是一个集防火墙、NAT 和一些其他功能的综合模块。很显然，

NAT ALG 也实现在 netfilter 中。对一个 ALG 在 Linux netfilter 实现中有两部分组成：conntrack helper 和 nat helper 两部分组成。Conntrack helper 顾名思义是 connection track helper 的意思，根据数据报的源，目的地址和端口等信息，把不同的数据报联系起来，之间的媒介就是 ip_conntrack 数组。Nat helper 完成的是根据数据报的内容，例如源，目的地址，端口，协议找到对应存储在的 ip_conntrack 数组的信息，根据 ALG 的要求对数据报进行修改，解决 NAT 有关的问题。例如在 linux/net/ipv4/netfilter 下：ftp 的 ALG 的两部分分别实现在 ip_conntrack_ftp.c 和 ip_nat_ftp.c 两个文件中。在 netfilter 中 ALG 是一个可扩充的灵活框架，增加不同应用的 NAT 文件，就可以增加该应用的 ALG.这些文件必须实现 conntrack helper 和 nat helper 两部分。

ALG 的框架

1． 主要文件

实现 conntrack 和 nat mangle 两部分的核心集中在四个文件中：
Linux/net/ipv4/netfilter/
ip_conntrack_core.c -----与 conntrack 有关的一些核心函数
ip_conntrack_standalone.c---- ip conntrack 模块的所有函数(netfilter 可以作为模块编译)
ip_nat_core.c---与 nat mangle 有关的一些核心函数
ip_nat_standalone.c---ip nat 模块的所有函数(netfilter 可以作为模块编译)

2．如何 track connection?

Ip_conntrack_standalone.c 定义并注册了 nefilter ops.(参考 netfilter 的有关资料)、
Struct nf_hook_ops ip_conntrack_in_ops; //For PRE_ROUTING
Struct nf_hook_ops ip_conntrack_local_out_ops;//For LOCAL_OUT
Struct nf_hook_ops ip_conntrack_out_ops;//For POST_ROUTING
Struct nf_hook_ops ip_conntrack_local_in_ops;//For LOCAL IN.

值得指出的是优先级分别为 NF_IP_PRI_CONNTRACK, NF_IP_PRI_CONNTRACK, NF_IP_PRI_LAST .NF_IP_PRI_LAST-1,分别为第一个和最后几个。也就是说当数据报进入 IP 层或者离开 IP 层的时候，就会对数据报进行 connection tracking.具体就是调用 call back 函数 ip_conntrack_in,ip_conntrack_local,ip_refrag,ipconfirm.
在 ip_conntrack_in 中，根据协议不同进行不同的 connection 分析。例如，对于 icmp 调用 icmp_error_track.对于 TCP/UDP 调用 resolve_normal_ct.
以 resolve_normal_ct 为例，首先在数组 ip_conntrack 中是否已经有响应的信息，没有则建立。如果已经建立，则更新连接信息。如果第一次出现的连接，状态置为 IP_CT_NEW。如果收到的是回复方向的数据报，置为 IP_CT_ESTABLSIHED 和 IP_CT_IS_REPLY.对于已经收到了回复包，并且当前方向是发起方向，置为 ESTABLISHED 状态。如果设置了 IPS_EXPECTED_BIT,则置为 IP_CT_RELATED.并更新 skb->nfct;这里和 ALG 联系最紧密的状态是 IP_CT_RELATED.

在 resolv_normal_ct 之后是调用与协议有关的函数更新 connection tracking info.例如对于 tcp 调用的是 ip_conntrack_proto_tcp.c 中的 tcp_packet 更新与 tcp 协议有关的状态，例如确定

当前 TCP 的状态（例如 SYN_RECV,SYN_SENT）。

再次之后就是 ALG conntrack helper 的关键部分---调用各 ALG 的 conntrack helper 的 help 函数.例如对于 ftp 其 help 函数定义在 ip_conntrack_ftp.c 中。其主要内容是在 ftp 数据部分查找需要进行特殊 NAT 处理的数据内容，例如把 ftp 的内部 IP 地址变成外部地址，以便正确建立 ftp 中的 data connection.

特别指出四点：

1）在 resolv_normal_ct(icmp_error_track) 和后面的 tcp_packet 之类的函数进行 TCP/UDP/ICMP 的 NAT 处理。

2)针对不同应用的 connection track 实现在对应用层数据的处理(废话？TCP/UDP/ICMP 之后的层次)，主要靠 conntrack helper 实现。

3）如何调用不同的 conntrack helper?在 ip_conntrack_core.c 的 init_conntrack()函数中，

```
conntrack->helper=ip_ct_find_helper(&repl_tuple);
struct ip_conntrack_helper *ip_ct_find_helper(const struct ip_conntrack_tuple *tuple)
{
    return LIST_FIND(&helpers,helper_cmp,tuple);
}
```

也就是根据数据报的源，目的地址，协议和应用层协议找到 ALG 的 conntrack helper.与特定应用相关的信息放在 ip_conntrack->help 中。不同的 ALG conntrack helper 就是通过 ip_conntrack_helper_register 链接在 helpers 结构中。

4）对于具有多于一条连接的应用，例如 IRC,FTP，另外的连接与最初的连接是一个 Related 的关系。最初的连接需要建立 ip_conntrack_expect 结构。

至此，conntrack helper 模块就可以把不同的数据报联系在 ip_conntrack 结构中，而且把一些与 ALG 相关的应用程序信息放在了这个结果中。在 nat helper 中需要这些信息的时候，就可以从 ip_conntrack 中取得。可谓万事具备，只欠东风————当需要进行 NAT 处理的时候，处理就行了。

3. 如何 expected and related?

很多应用之所以需要 ALG 是因为在应用层的数据里有 IP 地址和端口信息，当在 TCP、UDP 或 ICMP 做了常规的 NAT（更改了 ip 地址或端口）之后，这与在应用层的数据和端口信息不一致。这些地址和端口信息通常用来建立新的连接。例如在 ftp 中，在控制连接(control connection）的数据报中，应用层协议含有了数据连接（data connection）的地址和端口信息。当客户端（或者服务器）建立数据连接的时候，需要使用这些地址和端口信息。如果这些地址没有正确更正为一致，势必让连接不正常。问题在于数据连接和控制连接如何联系起来。显然是 connection tracking。Connection tracking 使用 related 的关系来描述这种关系：数据连接 related to 控制连接。控制连接在发现了在应用层数据含有地址和端口信息的时候，控制连接 expected 数据连接。

具体以 ftp 为例说明：

1）ip_conntrack_ftp.c 注册 conntrack helper。

```
struct ip_conntrack_helper ftp;
ftp.tuple.src..tcp.port=htons(FTP_PORT);
ftp.tuple.dst.protonum=IPPROTO_TCP;
ftp.mask.src.tcp.port=0xFFFF;
ftp.mask.dst.protonum=0xFFFF;
```

ftp.help=help;
ip_conntrack_helper_register(&ftp);
2)work flow

假定 ftp 客户端在 NAT 后面的 192.168.1.2,穿过 NAT 设备,访问在公共网络上的 202.100.100.2 的 ftp server.

a).src/dst/srcpt/dstpt:192.168.1.2/202.100.100.2/3333/21 syn client-→server control connection

当数据报到达 NAT 设备的时候,调用 ip_conntrack_in().进而进入 resolve_normal_ct().
在 resolve_normal_ct()(ip_conntrack_core.c)中,
get_tuple(skb->nh.iph,skb->len,&tuple,proto);
/*
tuple.src.ip=192.168.1.2
tuple.dst.ip=202.100.100.2
tuple.dst.protonum=tcp
tuple.src,u.tcp.port=3333
tuple.dst.u.tcp.port=21
*/
h=ip_conntrack_find_get(&tuple,NULL)
if(!h){
h=init_conntrack(&tuple,proto,skb);
}
在 init_conntrack()(ip_conntrack_core.c)中,
ip_conntrack_core.c
ip_conntrack_in()-→resolve_normal_ct()→init_conntrack();

invert_tuple(&repl_tuple,tuple,protocol);
/*
repl_tuple.dst.ip=192.168.1.2
repl_tuple.src.ip=202.100.100.2
repl_tuple.dst.protonum=tcp
repl_tuple.dst,u.tcp.port=3333
repl_tuple.src.u.tcp.port=21
*/
conntrack->tuplehash[IP_CT_DIR_ORIGINAL].tuple=*tuple;
conntrack->tuplehash[IP_CT_DIR_ORIGINAL].ctrack=conntrack;
conntrack->tuplehash[IP_CT_DIR_REPLY].tuple=repl_tuple;
conntrack->tuplehash[IP_CT_DIR_REPLY].ctrack=conntrack;

protocol->new(conntrack,skb->nh.iph,skb->len);

```
/*
  tcp_new():
  conntrack->proto.tcp.state=newconntrack;
  */
  init_timer(&conntrack->timeout);
  conntrack->timerout.data=(unsigned long)conntrack;
  conntrack->timeout.function= deatch_by_timeoutl;
  /*
    当 conntrack 超时后,自动删除该 conntrack  信息.
    */
  expected=LIST_FIND(&ip_conntrack_expect_list,expect_cmp,struct ip_conntrack expect
*,tuple);
  /*
    此时,很显然这是控制连接,没有 expected conntrack.expected=NULL;
    */
  if(!expected)
       conntrack->helper =ip_ct_find_helper(&repl_tuple);
  /*
    上面有了 repl_tuple 的值,可知,这里
    conntrack->helper=(ip_conntrack_helper*)&ftp;
    */
  回到 remove_normal_ct();
       *ctinfo=IP_CT_NEW;
       *set_reply=0;
       skb->nfct =&h->conntrack->infos[*ctinfo];
       /*
         在 skb 结构中含有一些 conntrack info 的信息.
         */
  返回到 ip_conntrack_in()中,
  ret=proto->packet(ct,(*pskb)->nh..iph,(*pskb)->len,ctinfo);
  /*
    ip_conntrack_tcp.c    tcp_packet():
    更新 tcp 状态 conntrack->proto.tcp.state ;
    处理 tcp conntrack 定时器.
  */
  if(ret!=NF_DROP &&ct->helper)
       ct->helper->help((*pskb)->nh.iph,(*pskb)->len,ct,ctinfo);

  这里调用的是 ip_conntrack_ftp.c 中的 help(),在该函数中,ctinfo==IP_CT_NEW,故,
  ret=NF_ACCEPT;
  至此,本数据包的 conntrack  处理完毕.但在 nat helper 中,一般会根据配置的 iptables 规则把
本数据报改为: src/dst/srcpt/dstpt:202.100.100.1/202.100.100.2/2222/21 client-→server
control connection.同时在 nat helper 中,调用 ip_nat_setup_info(ip_nat_core.c)时,会调用函数
ip_conntrack_alter_reply(conntrack,&reply)(ip_conntrack_core.c)把
```

conntrack->tuplehash[IP_CT_DIR_REPLY].tuple=*newreply;
/*
*newreply dst.ip=202.100.100.1
*newreply.src.ip=202.100.100.2
*newreply.dst.protonum=tcp
*newreply.dst,u.tcp.port=2222
*newreply.src.u.tcp.port=21
*/
conntrack->helper=LIST_FIND(&helpers,helper_cmp,struct ip_conntrack_helper *,newreply);
/*
conntrack->helper=(struct ip_conntrack_helper*)&ftp;
*/

b) src/dst/srcpt/dstpt:202.100.100.2/202.100.100.1/21/2222 syn+ack server-→client control connection

再次进入函数 ip_conntrack_in().
Proto=ip_ct_find_proto((*pskb)->nh.iph->protocol);
/*
proto=ip_conntrack_protocol_tcp;
*/
resolve_normal_ct(*pskb,proto,&set_reply,hooknum,&ctinfo);
暂且看看 resolve_normal_ct()函数:
get_tuple(skb->nh.iph,skb->len.&tuple,proto);
/*
tuple.src.ip=202.100.100.2
tuple.dst.ip=202.100.100.1
tuple.dst.protonum=tcp
tuple.src,u.tcp.port=21
tuple.dst.u.tcp.port=2222
*/
h=ip_conntrack_find_get(&tuple,NULL);
/*
* h!=NULL; h 就是上面我们已经填写好的 conntrack->tuplehash[IP_CT_DIR_REPLY];
*/
*ctinfo=IP_CT_ESTABLISHED + IP_CT_IS_REPLY;
*set_reply=1;
返回到 ip_conntrack_in()中,
proto->packet(ct,(*pskb)->nh.iph,(*pskb)->len,ctinfo);
/*
TCP 有关的状态信息更新
*/
ct->helper->help((*pskb)->nh.iph,(*pskb)->len,ct,ctinfo);
上面的调用实际上进入 ip_conntrack_ftp.c/help 中的 help()函数.在该函数中,

```
array[0]=(ntohl(ct->tuplehash[dir].tuple.src.ip)>>24)&0xFF;
array[1]=(ntohl(ct->tuplehash[dir].tuple.src.ip)>>16)&0xFF;
array[2]=(ntohl(ct->tuplehash[dir].tuple.src.ip)>>8)&0xFF;
array[3]=(ntohl(ct->tuplehash[dir].tuple.src.ip)>>0)&0xFF;
/*
 * array[0-3]—记录了方向 dir 源 ip 地址.
 */
for(…){
    if(search[i].dir!=dir) continue;
    found=find_pattern(data,datalen,
                       search[i].pattern,
                       search[i].plen,
                       search[i].skip,
                       search[i].term,
                       &matchoff,&matchlen,
                       array,search[i].getnum);
    if(found) break;
    /*
     * 这是 ftp conntrack helper 的主要部分了.它搜索长度为 datalen 的 data 中,看是否含有
     ip 地址信息.如果找到退出循环.
     */
}
if(found==-1)
    return NF_DROP;//丢包
else if(found==0)
    return NF_ACCEPT;//接收数据
/*
* 当应用层数据含有地址或端口信息的时候,就执行下面的语句了.
*/
struct ip_ct_ftp_master *ct_ftp_info =&ct->help.ct_ftp_info;
struct ip_conntrack_expect expect,*exp=&expectl
struct ip_ct_ftp_expect *exp_ftp_info =&exp->help.exp_ftp_info;
if(htonl((array[0]<<24)|(array[1]<<16)|array[2]<<8)|array[3])==ct->tuplehash[dir].tuple.src.ip){
    exp->seq=ntohl(tcph->seq)+matchoff;
    exp_ftp_info->len=matchlen;
    exp_ftp_info->ftptype=search[i].ftptype;
    exp_ftp_info->port =array[4]<<8|array[5];
    /*
    *记录了一些 expected info 的特征.
    */
}else{

}
exp->tuple =((struct ip_conntrack_tuple)
```

{{ct->tuplehash[!dir].tuple.src.ip},
{0}},
{htonl((array[0]<<24|(array[1]<<16)|(array[2]<<8)|(array[3]),
{.tcp={htons(array[4]<<8)|(array[5]}},
IPPROTO_TCP}});
exp->mask=((struct ip_conntrack_tuple)
{ {0xffffffff,{0}},
{0xffffffff,{.tcp={0xffff}},0xffff}});
exp->expectfn =NULL;
例如在数据包的 ftp 数据部分函数"227 Entering Passive Mode (*202,100,100,2,5,6)\r\n"*
server→client 为例,上面的结果是
exp_ftp_info->ftptype=IP_CT_FTP_PASV;
exp_ftp_info->port=256*5+6=1286;
exp->tuple.src.ip=192.168.1.2
exp->tuple.src.u.all=0;
exp->tuple.dst.ip=202.100.100.2;
exp->tuple.dst.tcp=1286;
exp->protonum=IPPROTO_TCP;
ip_conntrack_expect_related(ct,&expect);
查看 ip_conntrack_core.c 中的 ip_conntrack_expect_related();
in ip_conntrack_expect_related(struct ip_conntrack *related_to,struct ip_conntrack_expect *expect)
{
struct ip_conntrack_expect *new;
new=kmalloc();
new->expectant=related_to;
new->sibling=NULL;
/* Add to expected list for this connection */
list_add(&new->expected_list,&related_to->sibling_list);
/*Add to global list of expectations*/
list_prepend(ip_conntrack_expect_list,&new->list);
related_to->expecting++;
}
回到 ip_conntrack_in()中设置 IPS_SEEN_REPLY 值.
set_bit(IPS_SEEN_REPLY_BIT,&ct->status);

c)假如 server 向 client 发送了 port 命令, client 就会向 server 发起数据连接(data connection).这时候的数据报为
src/dst/srcpt/dstpt:192.168.1.2/202.100.100.2/1111/1286 syn client-→server data connection
数据报到达 NAT 设备时,进入 ip_conntrack_in 函数,在该函数中,
proto=ip_ct_find_proto((*pskb)->nh.iph->protocol);
/*
proto=&ip_conntrack_protocol_tcp;
*/

resolve_normal_ct(*pskb,proto,&set_reply,hooknum,&ctinfo);

再来看看 resolve_normal_ct 函数的东东.

get_tuple(skb->nh.iph,skb->len,&tuple,proto);

```
/*
  tuple.src.ip=192.168.1.2
  tuple.dst.ip=202.100.100.2
  tuple.dst.protonum=tcp
  tuple.src,u.tcp.port=1111
  tuple.dst.u.tcp.port=1286
*/
  h=ip_conntrack_find_get(&tuple,NULL);
  /*h=NULL*/
```

init_conntrack(&tuple,proto,skb);

现在看看 init_conntrack()函数:

ip_conntrack_core.c

ip_conntrack_in()-→resolve_normal_ct()→init_conntrack();

```
invert_tuple(&repl_tuple,tuple,protocol);
/*
  repl_tuple.dst.ip=192.168.1.2
  repl_tuple.src.ip=202.100.100.2
  repl_tuple.dst.protonum=tcp
  repl_tuple.dst,u.tcp.port=1286
  repl_tuple.src.u.tcp.port=1111
*/
conntrack->tuplehash[IP_CT_DIR_ORIGINAL].tuple=*tuple;
conntrack->tuplehash[IP_CT_DIR_ORIGINAL].ctrack=conntrack;
conntrack->tuplehash[IP_CT_DIR_REPLY].tuple=repl_tuple;
conntrack->tuplehash[IP_CT_DIR_REPLY].ctrack=conntrack;

protocol->new(conntrack,skb->nh.iph,skb->len);
/*
  tcp_new():
  conntrack->proto.tcp.state=newconntrack;
  */
  init_timer(&conntrack->timeout);
  conntrack->timerout.data=(unsigned long)conntrack;
  conntrack->timeout.function= deatch_by_timeout;

  expected=LIST_FIND(&ip_conntrack_expected_list,expect_cmp,
                     struct ip_conntrack_expect * ,tuple);
/*
```

前面的我们加入链表的 expect 结构如下

exp_ftp_info->port=256*5+6=1286;

exp->tuple.src.ip=192.168.1.2
exp->tuple.src.u.all=0;
exp->tuple.dst.ip=202.100.100.2;
exp->tuple.dst.tcp=1286;
exp->protonum=IPPROTO_TCP;
正好与 tuple 相匹配,
所以(expected=exp)!=NULL;
*/
if(expected){
    __set_bit(IPS_EXPECTED_BIT,&conntrack->status);
    conntrack->master=expected;
    expected->sibling=conntrack;
    LIST_DELETE(&ip_conntrack_expect_list,expected);
    expected->expectant->expecting--;
}

至此,我们已经如下的关系图.
expected->expectant=ct_ftp_ctrl_info;
ct_ftp_ctrl_info->sibling_list --------→ expected->expected_list;
ct_ftp_data_info->master=expected;
expected->sibling=ct_ftp_data_info;
其中：
ct_ftp_ctrl_info:
IP_CT_DIR_ORIGINAL:src/dst/srcprt/dstprt: 192.168.1.2/202.100.100.2/3333/21
IP_CT_DIR_REPLY: src/dst/srcprt/dstprt: 202.100.100.2/202.100.100.1/21/2222
ct_ftp_ctrl_info->helper=(struct ip_conntrack_helper*)&ftp;
Ct_ftp_data_info:
IP_CT_DIR_ORIGINAL:src/dst/srcprt/dstprt: 192.168.1.2/202.100.100.2/1111/1286
IP_CT_DIR_REPLY: src/dst/srcprt/dstprt: 202.100.100.2/192.168.1.2/1286/1111
ct_ftp_data_info->helper=NULL;
Expected:
exp_ftp_info->port=256*5+6=1286;
exp->tuple.src.ip=192.168.1.2
exp->tuple.src.u.all=0;
exp->tuple.dst.ip=200.100.100.2;
exp->tuple.dst.tcp=1286;
exp->protonum=IPPROTO_TCP;
返回到 resolve_normal_ct()
ip_conntrack_core.c
在 ip_conntack_in-→resolve_normal_ct()中
if(test_bit(IPS_EXPECTED_BIT,&h->ctrack->status)){
    *ct_info=IP_CT_RELATED;//相关啦,哈哈.
    *set_reply=0;
}

返回到 ip_conntrack_in()中,
ip_conntrack_core.c
在 ip_conntrack_in()中.接下来做一些与 TCP 协议有关的操作而已.
接下来,执行 nat helper,在函数 ip_nat_fn()(ip_nat_standalone.c)

```
if(ct->master &&
   master_ct(ct)->nat.info.helper &&
   master_ct(ct)->nat.info.helper->expect){
      nat = call_expect(master_ct(ct),pskb,hooknum,ct,info);
}
/*
```

这里会调用 ip_nat_helper nat_ftp 的函数 ftp_nat_expected()(ip_nat_ftp.c);同时设置新的 conntrack 内容使得数据报变成了:
src/dst/srcpt/dstpt:202.100.100.1/202.100.100.2/4444/1286 syn client-→server data connection
另外还要设置调用 ip_nat_setup_info 之后的 ct_ftp_data_info 是
Ct_ftp_data_info:
IP_CT_DIR_ORIGINAL:src/dst/srcprt/dstprt: 192.168.1.2/202.100.100.2/1111/1286
IP_CT_DIR_REPLY: src/dst/srcprt/dstprt: 202.100.100.2/202.100.100.1/1286/4444
ct_ftp_data_info->helper=NULL;

```
*/
```

小结:

1) 常规 NAT 规则通过 iptables 命令设定.可以设定的协议,有 tcp/udp/icmp,当然也可能有其他一些 iptables extenstions.ALG 的规则会受这些规则的影响,因为每次调用 ip_nat_setup_info 的时候需要改变 ip_conntrack 的一些信息.例如上面的需要更改 IP_CT_DIR_REPLY 方向的地址和端口信息,让 conntrack 可以识别回复方向的数据报.
2) ALG 主要通过 expected 和 related 关系处理同一应用的不同连接,例如上面 ftp 的数据连接和控制连接正是通过这种关系联系起来,以便后面的 nat helper 进行处理.
3) Nat helpr 实际上需要 conntrack 设置 ctinfo 为 IP_CT_RELATED 来调用 nat helper 的 expect 函数.
4) 如果客户端使用 port 命令，也就是使用私有 IP 地址的一方使用 port 命令，结果是有问题的，可以作些试验证明该想法，这也就是本修订版的改进之处。另外从这里也许可以改进 ftp ALG，让它能够处理更多的情况。

(To be continued.)

4．如何 nat mangling?

对于进入 NAT 设备的每一个数据包，connection 能够把它和其他数据包通过 connection 的概念联系起来。如果数据包是整个通讯过程的第一个数据包，ip_conntrack_info 为 IP_CT_NEW；如果反方向的数据包，不管是第一个还是后面的数据包，ip_conntrack_info 为 IP_CT_ESTABLISHED+IP_CT_IS_REPLY;在收到第一个回应方向的数据包后，发起方的所有属于该 connection 数据包经过都有 ip_conntrack_info 为 IP_CT_ESTABLISHED.当新 connection 的第一个数据包正好是先前某个 connection 所期望的(expected),这时

conntrack->status 设置为 IPS_EXPECTED_BIT.这时 ip_conntrack_info 为 IP_CT_RELATED.在 ip_conntrack 中存放有发起方向和回复方向的地址和端口信息。并且使用数据包的 IP 地址和端口信息与 ip_conntrack 中的信息对比，可以得知该数据包的方向是发起还是接收。(参见 ip_conntrack_core.c 中的 resolve_normal_ct 函数)

connection tracking 是 nat helper 的基础，因为后者需要前者提供的 connection 信息对不同的 connection 进行 nat 转换。对于一个 connection 是否进行 nat 转换，以及是否对与之 related 的 connection 连接取决与什么？也许可以根据使用的 IP 地址是否是私有地址(private ip address).但实际上这是一个错误的答案。因为很多时候，nat 设备连接的网段都使用私有地址，为了能够更方便地互联这些网段，同样需要进行 nat 转换。假如使用配置路由的方法是可以解决问题，但是网段数与路由配置数是一个灾难的平方关系。现在的常用方法是决定于用户。从这一点来说，用户才是上帝，如果一种技术要有非技术人员使用。对了，就是用户层的工具 iptables.用户有权利和能力根据自己的需要，使用 iptables 决定把哪些网段使用 nat 进行地址转换为其他网段的地址。常见的当然是私有网段（LAN）地址转换成可在 Internet 上路由的 WAN 网段。

既然 connection tracking 是 nat helper 的基础，当然需要在 nat helper 进行之前 tracking connection.

在 linux/net/ipv4/netfilter/ip_nat_standalone.c 中分别注册 PREROUTING 和 POSTROUTING 的 net filter hook.对应的回调函数分别为 ip_nat_fn,ip_nat_out,分别进行 Destination NAT 和 Source NAT.如果定义了 CONFIG_IP_NF_NAT_LOCAL，还注册了 Local_IN 和 Local_OUT 点的 filter hook.注意它的优先级与 connection tracking 比较，它确保了这些回调函数在 connection tracking 之后被调用。

继续前面的例子，假如在 NAT 后面的一个 PC 的 IP 地址为 192.168.1.2,网关设置为 192.168.1.1 也就是 NAT 设备的一个 LAN 端地址。NAT 设备的 WAN 端地址为 202.100.100.1.在 INTERNET 上有一台 ftp server 地址为 202.100.100.2.在 PC 上的 ftp 客户端需要访问 ftp server.而且 NAT 设备通过 iptables 配置成把 192.168.1.0/24 网段的地址使用 Source NAT 转换为 202.100.100.1 访问 INTERNET 上的任何 PC 或者服务器，这是在 POSTROUTING 点完成的。

a).src/dst/srcpt/dstpt:192.168.1.2/202.100.100.2/3333/21 syn client-→server control connection

当数据包经过 conntrack 模块的时候，建立的 ip_conntrack 信息如下：
ct_ftp_ctrl_info:
IP_CT_DIR_ORIGINAL:src/dst/srcprt/dstprt: 192.168.1.2/202.100.100.2/3333/21
IP_CT_DIR_REPLY: src/dst/srcprt/dstprt: 202.100.100.2/192.168.1.2/21/3333
ct_ftp_ctrl_info->helper=(struct ip_conntrack_helper*)&ftp;
ctinfo=IP_CT_NEW;
当数据包进入 nat helper 部分，第一个 net filter hook 点是 PREROUTING。调用的 netfilter hook 函数为 ip_nat_fn().在该函数中，先得到在 conntracking helper 中得到的 ip_conntrak 信息。Nat helper 的处理跟 ctinfo 的值有关，它代表了不同的连接阶段，例如，IP_CT_RELATED,IP_CT_NEW,IP_CT_RELATED 等。第一个数据包的 ctinfo 显然为 IP_CT_NEW.如果该连接是第一次进入 hook 点，而且不是被其它连接所 expected 的连接，直接调用 ip_nat_rule_find().该函数显然就是根据 iptables 的对 nat 的配置，在 ip_conntrack 中的 nat 记录 NAT 对该连接的处理信息。这里最重要的就是调用函数 ip_nat_setup_info 在

ip_conntrack->nat.info.manips[]记录对该连接数据包的应该进行的 IP 和端口信息的处理。在建立 nat 处理的规则后，真正对数据包的处理就在 do_bindings 函数里.

[ip_nat_standalone.c-----ip_nat_fn()]

```
ct= ip_conntrack_get(*pskb,&ctinfo);
/*
在 ip_conntrack_core.c 的 resolve_normal_ct() 中， 最后设置 skb->nfct=&h->ctrack->infos[*ctinfo].则
ct=(struct ip_conntrack*)(pskb->nfct)->master. 其中 (pskb->nfct)->master=&h->conntrack->ct_general;
最后结果为
ct＝&ct_ftp_ctrl_info;
ctinfo=IP_CT_NEW.
*/
switch(ctinfo){
    case IP_CT_NEW:
        info =& ct->nat.info;
        /*
        *这是第一个数据包，显然 info->initialized 为 0。
        */
        if (!(info->initialized & (1 << maniptype))) {
            if(ct->master&&master_ct(ct)->nat.info.helper&&
                    master_ct(ct)->nat.info.helper->expect) {
                ret = call_expect(master_ct(ct), pskb,
                        hooknum, ct, info);
            } else {
                ret = ip_nat_rule_find(pskb, hooknum, in, out, ct, info);
            }
            /*
            ct->master 对于被 related 的 connection 才有，这里为 NULL.所以调用的是
            ip_nat_rule_find(pskb,hooknum,in,out,ct,info).
             */
        }
    break;
}
```

[ip_nat_rule.c------ ip_nat_fn()-→ip_nat_rule_find()]

```
ret=ipt_do_table(pskb,hooknum,in,out,&nat_table,NULL);
/*
```

根据 iptables 配置的把 nat 处理的规则记录在 ip_conntrack->nat.info.manips 中。在该函数中首先用本数据包的端口和 IP 地址信息与 iptables 配置的规则条件进行比较，如果条件匹配，调用 target 函数。例如在本例中使用的 target 是 masquerade_target().在该 target 中首先查找路由表确定到目的 IP 地址使用的网络接口；然后，把该接口的地址 rt->rt_src 作为该数据包的新原地址。然后调用 ip_nat_setup_info 在 ip_conntrack->nat.info.manips 中记录该 NAT 处理规则。这里正好验证了前面提到的 NAT 转换取决于 iptables 规则。对于本例，iptables 没有配置规则，故使用默认规则，一般为 ACCEPT。

```
*/
if(ret==NF_ACCEPT){
    if(!info->initialized &(1<<HOOK2MANIP(hooknum))))
        ret= alloc_null_binding(ct,info,hooknum);
    /*
        这里没有初始化过，显然调用 alloc_null_binding.
    */
}
```

alloc_null_binding()———给该连接信息分配空的 NAT 处理规则。

```
[ip_nat_rule.c------ ip_nat_fn()-→ip_nat_rule_find()-→alloc_null_binding()]
alloc_null_binding(ct,info,hooknum){
/*
ct:
IP_CT_DIR_ORIGINAL:src/dst/srcprt/dstprt: 192.168.1.2/202.100.100.2/3333/21
IP_CT_DIR_REPLY: src/dst/srcprt/dstprt: 202.100.100.2/192.168.1.2/21/3333
ct_ftp_ctrl_info->helper=(struct ip_conntrack_helper*)&ftp;
ctinfo=IP_CT_NEW;
hooknum=NF_IP_PRE_ROUTING
*/
u_int32_t ip=conntrack->tuplehash[IP_CT_REPLY].tuple.src.ip;
/*
HOOK2MANIP(hooknum)== IP_NAT_MANIP_DST.
Ip=202.100.100.2
*/
struct ip_nat_multi_range mr=
{1,{{IP_NAT_RANGE_MAP_IPS,ip,ip,{0},{0}}}};
/*
 mr.size=1
 mr.range[0].flags=IP_NAT_RANGE_MAP_IPS
 mr.range[0].min_ip=202.100.100.2
 mr.range[0].max_ip=202.100.100.2
 mr.range[0].min=0;
 mr.range[0].max=0
 */
return ip_nat_setup_info(ct,&mr,hooknum);
}
```

从上面的分析可知道，实际上调用 alloc_null_binding()记录的 NAT 处理规则和原数据包是一样的，例如上面例子目的地址实际上是没有改变的。还要注意一点，实际上在 ip_nat_setup_info 中会把 ip_conntrack->nat.info.initialized | =(1 <<HOOK2MANIP(hooknum)), 也就是标记 IP_NAT_MANIP_DST 已经初始化了。

在 ip_nat_setup_info 中，可以看到，如果在 NAT 处理规则中 NAT 处理前的源和目的元组(ip_conntrack_tuple)都没有什么变化，实际上 ip_conntrack->nat.info.manips 没有改变 ip_conntrack->nat.info.num_manips 表示的 IP 和端口处理也没有增加或减少。这可能也就是前面命名为 alloc_null_binding.由于 ip_nat_setup_info 比较的复杂，在后面用具体例子来分析。

返回到函数 ip_nat_fn()中，当 NAT 处理规则(bindings)的已经建立之后，就进行 NAT 处理了。

[ip_nat_standalone.c-----ip_nat_fn()]

```
return do_bindings(ct,ctinfo,info,hooknum,pskb);
/*
ct=&ct_ctrl_ftp;
ct:
IP_CT_DIR_ORIGINAL:src/dst/srcprt/dstprt: 192.168.1.2/202.100.100.2/3333/21
IP_CT_DIR_REPLY: src/dst/srcprt/dstprt: 202.100.100.2/192.168.1.2/21/3333
ct_ftp_ctrl_info->helper=(struct ip_conntrack_helper*)&ftp;
ctinfo=IP_CT_NEW;
hooknum=NF_IP_PRE_ROUTING
*/
```

do_bindings---Do packet manipulation according to bindings.也就是根据 NAT 处理规则对数据包进行地址和端口的转换。这里我们建立的是空 NAT 处理规则(null bindings).实际上是不会进行任何地址转换。

接着数据包会经过其他 HOOK 点-----FORWARD 和 POSTROUTING.在 FORWARD 点 NAT 实际上是不会注册什么 net filter hook 函数的。直接来到本数据包的最后一个 net filter hook 点------POSTROUTING 了。在该点上注册了 MASQUERADE 为目标(target)的 SNAT 规则。

在 ip_nat_standalone.c 中，在 NF_IP_POST_ROUTING 上注册的 net filter hook 函数是 ip_nat_out().该函数先是进行一些诸如 raw socket 和分片数据包的检查，最重要的工作都是调用 ip_nat_fn()中进行。

[ip_nat_standalone.c---------ip_nat_out()]

```
static unsigned int ip_nat_out(unsigned int hooknum,      struct sk_buff **pskb,
        const struct net_device *in, const struct net_device *out,
        int (*okfn)(struct sk_buff *))
{
    /* root is playing with raw sockets. */
    if ((*pskb)->len < sizeof(struct iphdr) || (*pskb)->nh.iph->ihl * 4 < sizeof(struct iphdr))
    return NF_ACCEPT;
    if ((*pskb)->nh.iph->frag_off & htons(IP_MF|IP_OFFSET)) {
        *pskb = ip_ct_gather_frags(*pskb);
        if (!*pskb)      return NF_STOLEN;
    }
    return ip_nat_fn(hooknum, pskb, in, out, okfn);
}
```

回到函数 ip_nat_fn(),不同的是 hooknum＝NF_IP_POST_ROUTING.在该函数中实际上和上次以 hooknum＝NF_IP_PRE_ROUTING 进入该函数的流程是一样的。主要的区别是它会在 ip_nat_rule_find()建立的不再是 null bindings，是实实在在的 bindings.do_bindings 也需要进行能够真正意义上的 NAT 处理了。

```
static unsigned int
ip_nat_fn(unsigned int hooknum,
        struct sk_buff **pskb,
```

```
        const struct net_device *in,
        const struct net_device *out,
        int (*okfn)(struct sk_buff *))
{
      /* maniptype == SRC for postrouting. */
      enum ip_nat_manip_type maniptype = HOOK2MANIP(hooknum);
      /*
      hooknum=NF_IP_POST_ROUTING
     maniptype= IP_NAT_MANIP_SRC
   */
      ct = ip_conntrack_get(*pskb, &ctinfo);
/*
ct=&ct_ctrl_ftp;
ct:
IP_CT_DIR_ORIGINAL:src/dst/srcprt/dstprt: 192.168.1.2/202.100.100.2/3333/21
IP_CT_DIR_REPLY: src/dst/srcprt/dstprt: 202.100.100.2/192.168.1.2/21/3333
ct_ftp_ctrl_info->helper=(struct ip_conntrack_helper*)&ftp;
 */
      switch (ctinfo) {
      case IP_CT_NEW:
            info = &ct->nat.info;
            /*
              由于该连接第一次经过POSTROUTING点，这里会调用ip_nat_rule_find建立在该
hook点的NAT bindings。
              */
            if (!(info->initialized & (1 << maniptype))) {
                  ret = ip_nat_rule_find(pskb, hooknum, in, out, ct, info);
            }
            break;
      }
      /*
      进行实际的NAT处理(nat manipulation)
       */
      return do_bindings(ct, ctinfo, info, hooknum, pskb);
}
```

ip_nat_rule_find()----在nat table中查找本数据包符合条件的规则，并建立NAT处理规则(nat bindings).首先调用的是函数ipt_do_table(),它实际上是根据数据包pskb中的信息去匹配nat_table中的规则，如果条件满足就执行该规则的目标(target)。在nat_table中常见的target有MASQUERADE(伪装),SNAT(源地址NAT),DNAT(目的地址NAT),ACCEPT(立即结束在nat_table中匹配规则),RETURN(返回到上一层规则).(可以man iptables参考各种target的具体用法)。一般该target会调用ip_nat_setup_info,并且在info->initialized给IP_NAT_MANIP_SRC置位。如果没有设置该位，接着会调用alloc_null_binding()函数建立空的NAT bindings。

【ip_nat_rule.c------ip_nat_out()-→ip_nat_fn()-→ip_nat_rule_find()】

```
int ip_nat_rule_find(struct sk_buff **pskb,
```

```
                unsigned int hooknum, const struct net_device *in,
                const struct net_device *out, struct ip_conntrack *ct,
                struct ip_nat_info *info)
{
    ret = ipt_do_table(pskb, hooknum, in, out, &nat_table, NULL);
    if (ret == NF_ACCEPT) {
        if (!(info->initialized & (1 << HOOK2MANIP(hooknum))))
            ret = alloc_null_binding(ct, info, hooknum);
    }
    return ret;
}
```

ipt_do_table()实际上是一个很复杂的函数。这里并不想详细分析它的流程，因为它涉及了iptables 的很多知识，简直需要先对一般的读者大篇幅地介绍 iptables 常识。为了简单，分析一下本例中使用的 target———masquerade_target().具体分析它是如何把 iptables 规则转换为 NAT bindings。

masquerade_target()完成哪些功能？根据路由表决定 NAT 转换之后的源 IP 地址，并且调用 ip_nat_setup_info()函数建立 NAT 处理规则(nat bindings).没有分析过 netfilter 的人一定会认为每次收到数据包 target 都会执行一次，其实不然。它只是在收到该连接的第一数据包时执行，它完成的任务是根据 iptables 配置转换成 NAT bindings。后面收到该连接的数据包后直接调用 do_bindings 执行 NAT 处理.

【 ipt_MASQUERADE.c------ip_nat_out()-→ip_nat_fn()-→ip_nat_rule_find()-→ipt_do_table()-→ masquerade_target()】

```
static unsigned int masquerade_target(struct sk_buff **pskb,
            unsigned int hooknum,       const struct net_device *in,
            const struct net_device *out,    const void *targinfo,
            void *userinfo)
{
    ct = ip_conntrack_get(*pskb, &ctinfo);
    IP_NF_ASSERT(ct && (ctinfo == IP_CT_NEW
                    || ctinfo == IP_CT_RELATED));
    /*
        从上面的语句可以看到，iptables的target对任意连接只是执行一次，因为ctinfo必须为IP_CT_NEW,IP_CT_RELATED.
    */
    mr = targinfo;
    /*
    mr记录了地址转换信息，例如对于SNAT，记录了源地址的转换后的IP地址或端口。对于MASQUERADE记录的可能有端口信息，因为MASQUERADE可以和端口转换信息连用。
    */
    key.dst = (*pskb)->nh.iph->daddr;
    key.src = 0; /* Unknown: that's what we're trying to establish */
    key.tos = RT_TOS((*pskb)->nh.iph->tos)|RTO_CONN;
    key.oif = 0;
    if (ip_route_output_key(&rt, &key) != 0) {
```

```
    }
    /*
    根据目的地址，查找路由表
    */
    newsrc = rt->rt_src;
    ip_rt_put(rt);
    /*
    newsrc是路由表决定的到达该目的地址需要使用的源地址。对于本例子，这里使用IP地址是202.100.100.1
    */
    ct->nat.masq_index = out->ifindex;
    /* Transfer from original range. */
    newrange = ((struct ip_nat_multi_range)
        { 1, { { mr->range[0].flags | IP_NAT_RANGE_MAP_IPS,
            newsrc, newsrc,
            mr->range[0].min, mr->range[0].max } } });
    /*
    newrange.rangesize=1.
    newrange.range[0].flags |=IP_NAT_RANGE_MAP_IPS;
    newrange.min_ip=202.100.100.2
    newrange.max_ip=202.100.100.2
    newrange.min=mr.range[0].min可能记录了端口信息。
    newrange.max=mr.range[0].max.  可能记录了端口信息
    */
    /* Hand modified range to generic setup. */
    return ip_nat_setup_info(ct, &newrange, hooknum);
}
```

ip_nat_setup_info()的功能是建立NAT处理规则。其依据是conntrack信息和mr信息。例如对于本例，conntrack记录了连接的original和reply方向的端口和ip地址信息，以及该连接的当前状态。mr记录了SNAT之后的IP地址和端口信息。该函数首先调用get_unique_tuple()得到SNAT之后的地址和端口信息。这是一个核心的NAT处理函数，因为它决定了NAT之后的IP地址和端口信息。然后修改conntrack中reply方向的端口和IP地址信息，因为它需要更正这些信息来识别reply方向的数据包。真正的NATbinding信息记录在ip_conntrack->nat.info.manips和ip_conntrack->nat.info.num_manip中。比较get_unique_tuple返回的结果和原来的tuple，可以确定需要建立的NAT bindings。

值得指出的是在ip_conntrack_alter_reply()中基于新的tuple重新查找conntrack helper，在成功建立NAT bindings之后，会更新该连接的nat helper。一般情况下，这也是第一次记录nat helper。并把nat helper记录在ip_conntrack->nat.info->helper中，本例中它的值正好是ftp的nat helper.

本例中需要建立SNAT的NAT binding。具体的，对于original方向的数据包，要对在POSTROUTING其进行SNAT操作，src/dst/srcprt/dstprt: 192.168.1.2/202.100.100.2 /3333/21转换成202.100.100.1/202.100.100.2/2222/21.同样该连接的reply方向的数据包是202.100.100.2/202.100.100.1/21/2222.,通过NAT设备时他需要进行reply方向的DNAT，进行转换的hook点是PREROUTING，转换的NAT规则是：src/dst/srcprt/dstprt:

202.100.100.2/202.100.100.1/21/2222 -→202.100.100.2/192.168.1.2/21/3333.
[ip_nat_core.c------
ip_nat_fn()-→ip_nat_rule_find()-→alloc_null_binding()-→ip_nat_setup_info()]
unsigned int ip_nat_setup_info(struct ip_conntrack *conntrack,
const struct ip_nat_multi_range *mr, unsigned int hooknum)
{
IP_NF_ASSERT(hooknum == NF_IP_PRE_ROUTING
|| hooknum == NF_IP_POST_ROUTING || hooknum == NF_IP_LOCAL_OUT);
/*
建立NAT binding的hook点只有三个
NF_IP_PRE_ROUTING,NF_IP_POST_ROUTING,NF_IP_LOCAL_OUTPUT
*/
IP_NF_ASSERT(info->num_manips < IP_NAT_MAX_MANIPS);
IP_NF_ASSERT(!(info->initialized & (1 << HOOK2MANIP(hooknum))));
/*
对于每一个SNAT或者DNAT建立NAT bindings都执行一次。
*/
invert_tuplepr(&orig_tp, &conntrack->tuplehash[IP_CT_DIR_REPLY].tuple);
/*
orig_tp:
src/dst/srcprt/dstprt: 192.168.1.2/202.100.100.2/3333/21.
为什么要invert_tuplepr()来得到original tuple????对于在PREROUTING点进行的NAT binding，这样肯定没有问题，因为它还没有进行DNAT绑定改变REPLY方向ip_conntrack_tuple的值。对于POSTROUTING点的SNAT，如果先前没有DNAT，也不会改变reply方向的ip_conntrack_tuple,得到的original tuple。如果先前进行了DNAT，reply方向的ip_conntrack_tuple改变了，invert之后的tuple当然和原先不同了，但是分析get_unique_tuple可以知道，希望NAT binding使用前一个binding之后的结果。
*/
do {
get_unique_tuple(&new_tuple, &orig_tp, mr, conntrack, hooknum)；
/*
get_unique_tuple根据mr中的内容，调整orignal tuple里的值，把结果记录在new_tuple中。对于本例进行SNAT之后，结果如下：
orig_tp:
src/dst/srcprt/dstprt: 192.168.1.2/202.100.100.2/3333/21.
new_tuple:
src/dst/srcprt/dstprt: 202.100.100.1/202.100.100.2/2222/21.
有一点值得指出，orig_tp和new_tuple的端口可能改变，它们之间的关系参见后面的get_unique_tuple分析
*/
invert_tuplepr(&reply, &new_tuple);
/*
reply:
src/dst/srcprt/dstprt:202.100.100.2/202.100.100.1/21/2222

```
        */
        /* Alter conntrack table so it recognizes replies.
                        If fail this race (reply tuple now used), repeat. */
    } while (!ip_conntrack_alter_reply(conntrack, &reply));
    /*
```

实际上任何NAT处理规则都不会对ORIGNAL 方向的tuple进行改变，只是对reply方向的NAT处理规则进行改变，因为tuple的作用是识别各个方向的数据包。对于original方向的数据包其特征是没有经过NAT改变的，而相反方向的数据包是变化过的。
ip_conntrack_alter_reply就是改变REPLY方向的tuple信息。为什么这里要使用循环？原程序注释里解释是确保有竞争的条件下，也能成功。前面也提到了这里进行conntrack信息修正，同时会修正conntrack helper的信息。

```
    */
    /* Create inverse of original: C/D/A/B' */
    invert_tuplepr(&inv_tuple, &orig_tp);
    /*
      inv_tuple:
          src/dst/srcprt/dstprt: 202.100.100.2/192.168.1.2/21/3333
      */

    /* Has source changed?. */
    /*
```

如果orig_tp和new_tuple的源地址不相同，可以断定这是SNAT。建立NAT bindings放在ip_conntrack->nat.info.manips[],并增加ip_conntrack->nat.info.num_manips.具体到本例，正好源地址改变了，建立SNAT的NAT bindings。

```
    */
    if (!ip_ct_tuple_src_equal(&new_tuple, &orig_tp)) {
        /* In this direction, a source manip. */
        info->manips[info->num_manips++] =
            ((struct ip_nat_info_manip)
              { IP_CT_DIR_ORIGINAL, hooknum,
                IP_NAT_MANIP_SRC, new_tuple.src });
        /*
        manip.direction=IP_CT_DIR_ORIGINAL
        manip.hooknum=NF_IP_POST_ROUTING
        manip.maniptype=IP_NAT_MANIP_SRC;
        manip.manip.ip=202.100.100.1
        manip.manip.u.tcp.port=2222
        */
        /* In the reverse direction, a destination manip. */
        info->manips[info->num_manips++] =
            ((struct ip_nat_info_manip)
              { IP_CT_DIR_REPLY, opposite_hook[hooknum],
                IP_NAT_MANIP_DST, orig_tp.src });
        /*
```

```
            manip.direction=IP_CT_DIR_REPLY
            manip.hooknum=NF_IP_PRE_ROUTING
            manip.maniptype=IP_NAT_MANIP_DST;
            manip.manip.ip=192.168.1.2
            manip.manip.u.tcp.port=3333
            */
        }
/*
    如果orig_tp和new_tuple的目的地址不相同，可以断定这是DNAT。建立NAT bindings放在ip_conntrack->nat.info.manips[],并增加ip_conntrack->nat.info.num_manips.具体到本例，正好源地址改变了，建立DNAT的NAT bindings。
    */
        /* Has destination changed? */
        if (!ip_ct_tuple_dst_equal(&new_tuple, &orig_tp)) {
            /* In this direction, a destination manip */
            info->manips[info->num_manips++] =
                ((struct ip_nat_info_manip)
                 { IP_CT_DIR_ORIGINAL, hooknum,
                   IP_NAT_MANIP_DST, reply.src });
            /* In the reverse direction, a source manip. */
            info->manips[info->num_manips++] =
                ((struct ip_nat_info_manip)
                 { IP_CT_DIR_REPLY, opposite_hook[hooknum],
                   IP_NAT_MANIP_SRC, inv_tuple.src });
        }
        /* If there's a helper, assign it; based on new tuple. */
        if (!conntrack->master)
            info->helper = LIST_FIND(&helpers, helper_cmp, struct ip_nat_helper *, &reply);
        /*
        在这里挂接nat helper。显然本例中需要挂接ftp的nat helper。
        */
        /* It's done. -----这里确保了所有的NAT binding只会建立一次*/
        info->initialized |= (1 << HOOK2MANIP(hooknum));
        /*
        更新一些hash表。
        */
        if (in_hashes) {
            IP_NF_ASSERT(info->bysource.conntrack);
            replace_in_hashes(conntrack, info);
        } else {
            place_in_hashes(conntrack, info);
        }
        return NF_ACCEPT;
}
```

get_unique_tuple()基于tuple和mr，查找NAT转换之后的IP地址和端口信息。对于net fileter，它的规则是首先对于DNAT确定该tuple是否已经有映射，如果有，使用好了。然后调用find_best_ips_proto_fast在给定的范围内mr查找合适的转换结果，并且改变tuple的值。如果可以查找到结果，进一步确保该tuple没有被使用。如果该tuple没有使用，万事大吉，OK，就是它了。否则调用proto的unique_tuple改变端口找到符合要求的转换结果。如果没有找到，循环在给定的范围内查找下一个符合条件的结果，直到找到为止。

[ip_nat_core.c------

ip_nat_fn()-→ip_nat_rule_find()-→alloc_null_binding()-→ip_nat_setup_info()-→get_unique_tuple()]

```
static int get_unique_tuple(struct ip_conntrack_tuple *tuple,
                const struct ip_conntrack_tuple *orig_tuple, const struct ip_nat_multi_range *mrr,
                struct ip_conntrack *conntrack, unsigned int hooknum)
{
    struct ip_nat_protocol *proto = find_nat_proto(orig_tuple->dst.protonum);
    /*
    proto=&ip_nat_protcol_tcp;
    */
    struct ip_nat_multi_range *mr = (void *)mrr;

    /* 1) If this srcip/proto/src-proto-part is currently mapped,
       and that same mapping gives a unique tuple within the given
       range, use that.

       This is only required for source (ie. NAT/masq) mappings.
       So far, we don't do local source mappings, so multiple
       manips not an issue.    */
    /*
    这是一种特殊情况：orig_tuple中的地址和端口信息已经有相应的映射。
    */
    if (hooknum == NF_IP_POST_ROUTING) {
        manip = find_appropriate_src(orig_tuple, mr);
        if (manip) {
            /* Apply same source manipulation. */
            *tuple = ((struct ip_conntrack_tuple) { *manip, orig_tuple->dst });
            return 1;
        }}

    /* 2) Select the least-used IP/proto combination in the given
       range.——在给定的IP地址和端口范围mr中查找最合适的地址和端口。
    */
    *tuple = *orig_tuple;
    while ((rptr = find_best_ips_proto_fast(tuple, mr, conntrack, hooknum))
               != NULL) {
        /* 3) The per-protocol part of the manip is made to
```

```
        map into the range to make a unique tuple. */
    /* Only bother mapping if it's not already in range
        and unique */
    /*
```

按是否查找到的nat范围中是否指定了端口分成两种情况。一种是mr中指定了端口，且要查找到的IP地址和端口信息在给定的地址范围内。另一种是没有指定端口，但无论如何都要确保该元组没有被nat使用。为什么？例如有两个connection都对应相同的元组src/dst/srcprt/dstprt:202.100.100.1/202.100.100.2/2222/21.那么在REPLY方向有个数据包被收到，它的src/dst/srcprt/dstprt:为202.100.100.2/202.100.100.1/21/2222,这时候NAT无法却分如何做NAT，因为netfilter不可能用抛硬币的方法决定选择那个元组作为转换的结果。如果可以确保该tuple没有被使用，就是用该结果，否则执行下面的语句。

```
    */
    if ((!(rptr->flags & IP_NAT_RANGE_PROTO_SPECIFIED)
         || proto->in_range(tuple, HOOK2MANIP(hooknum),
                &rptr->min, &rptr->max))
        && !ip_nat_used_tuple(tuple, conntrack)) {
        ret = 1;
        goto clear_fulls;
    } else {
        /*
```

如果该结果被一些conntrack使用，我们一般可以使用不同的端口，来确保NAT转换之后的tuple在NAT设备唯一。这里调用的是ip_nat_protocol_tcp. 在ip_nat_protocol_tcp.unique_tuple()中实际上会调用ip_nat_used_tuple()确保该tuple不曾被使用;

```
        */
        if (proto->unique_tuple(tuple, rptr,
                    HOOK2MANIP(hooknum),
                    conntrack)) {
            /* Must be unique. */
            IP_NF_ASSERT(!ip_nat_used_tuple(tuple,
                                conntrack));
            ret = 1;
            goto clear_fulls;
        } else if (HOOK2MANIP(hooknum) == IP_NAT_MANIP_DST) {
            /* Try implicit source NAT; protocol may be able to play with ports to
                                make it unique. */
            /*
            改变端口范围，尝试新的端口。使用privileged端口????。
            */
            struct ip_nat_range r
                = { IP_NAT_RANGE_MAP_IPS,
                    tuple->src.ip, tuple->src.ip,
                    { 0 }, { 0 } };
            if (proto->unique_tuple(tuple, &r,
                        IP_NAT_MANIP_SRC,
```

```
                                conntrack)) {
                        /* Must be unique. */
                        IP_NF_ASSERT(!ip_nat_used_tuple
                                (tuple, conntrack));
                        ret = 1;
                        goto clear_fulls;
                    }
                }
                DEBUGP("Protocol can't get unique tuple %u.\n",
                       hooknum);
                /*
                没有找到合适的tuple
                */
            }
            /* Eliminate that from range, and try again.  尝试下一个合适的tuple*/
            rptr->flags |= IP_NAT_RANGE_FULL;
            /*去掉当前的NAT范围,重新循环后尝试给定范围内的下一个端口*/
            *tuple = *orig_tuple;
        }

        ret = 0;

     clear_fulls:
        /* Clear full flags. */
        IP_NF_ASSERT(mr->rangesize >= 1);
        for (i = 0; i < mr->rangesize; i++)
            mr->range[i].flags &= ~IP_NAT_RANGE_FULL;

        return ret;
}
```

对于ftp,使用的协议是tcp，proto->unique_tuple实际上调用的是tcp_unique_tuple.如果是SNAT则更新源port，否则更新目的port。首先确定合适的端口范围，然后在该范围内查找没有被使用的tuple。

[ip_nat_proto_tcp.c------

ip_nat_fn()-→ip_nat_rule_find()-→alloc_null_binding()-→ip_nat_setup_info()-→get_unique_tuple()→tcp_unique_tuple()]

```
static int tcp_unique_tuple(struct ip_conntrack_tuple *tuple,
              const struct ip_nat_range *range,   enum ip_nat_manip_type maniptype,
              const struct ip_conntrack *conntrack)
{
        if (maniptype == IP_NAT_MANIP_SRC)
            portptr = &tuple->src.u.tcp.port;
        else
            portptr = &tuple->dst.u.tcp.port;
```

```
/*
如果是SNAT则需要修改的是source port,否则修改destination port
*/

/* If no range specified... 没有指定端口范围*/
if (!(range->flags & IP_NAT_RANGE_PROTO_SPECIFIED)) {
    /* If it's dst rewrite, can't change port——如果是DNAT则不能修改端口 */
    if (maniptype == IP_NAT_MANIP_DST)
        return 0;

    /* Map privileged onto privileged. 对于小于1024的端口映射之后的端口要在原来的端口范围。这里有很复杂的原因，例如防火墙的规则等。而大于1024的端口也尽量映射为相同的端口范围*/
    if (ntohs(*portptr) < 1024) {
        /* Loose convention: >> 512 is credential passing */
        if (ntohs(*portptr)<512) {
            min = 1;
            range_size = 511 - min + 1;
        } else {
            min = 600;
            range_size = 1023 - min + 1;
        }
    } else {
        min = 1024;
        range_size = 65535 - 1024 + 1;
    }
} else {//如果指定了端口范围，只能在指定的端口内配置了
    min = ntohs(range->min.tcp.port);
    range_size = ntohs(range->max.tcp.port) - min + 1;
}
/*
在指定的范围内查找没有被占用的四元组。
*/
for (i = 0; i < range_size; i++, port++) {
    *portptr = htons(min + port % range_size);
    if (!ip_nat_used_tuple(tuple, conntrack)) {
        return 1;
    }
}
return 0;
}
```

当 ip_nat_setup_info 调用 get_unique_tuple 得到适合的 NAT 处理规则后，除建立 NAT binding 之外，还需要修改 conntrack 信息，让 reply 方向的数据包可以识别为该连接。完成该任务的函数就是 ip_conntrack_alter_reply。另外，可能要更新 conntrack helpe。r 该函数结

束后，该连接的 conntrack info 变为：

ct=&ct_ctrl_ftp;

ct:

IP_CT_DIR_ORIGINAL:src/dst/srcprt/dstprt: 192.168.1.2/202.100.100.2/3333/21

IP_CT_DIR_REPLY: src/dst/srcprt/dstprt: 202.100.100.2/202.100.100.1/21/2222

[ip_nat_proto_tcp.c------

ip_nat_fn()-→ip_nat_rule_find()-→alloc_null_binding()-→ip_nat_setup_info()-→get_unique_tuple()→ip_conntrack_alter_reply ()]

```
/* Alter reply tuple (maybe alter helper).    If it's already taken, return 0 and don't do alteration. */
int ip_conntrack_alter_reply(struct ip_conntrack *conntrack,
                             const struct ip_conntrack_tuple *newreply)
{
        WRITE_LOCK(&ip_conntrack_lock);
        if (__ip_conntrack_find(newreply, conntrack)) {
                WRITE_UNLOCK(&ip_conntrack_lock);
                return 0;
        }
        /* Should be unconfirmed, so not in hash table yet */
        IP_NF_ASSERT(!is_confirmed(conntrack));
        conntrack->tuplehash[IP_CT_DIR_REPLY].tuple = *newreply;
        if (!conntrack->master)
                conntrack->helper = LIST_FIND(&helpers, helper_cmp,
                                              struct ip_conntrack_helper *,
                                              newreply);
        WRITE_UNLOCK(&ip_conntrack_lock);
        return 1;
}
```

至此，针对本连接的 NAT bindings 已经建立，建立的结果为：

ORIGINAL方向：

manip.direction=IP_CT_DIR_ORIGINAL

manip.hooknum=NF_IP_POST_ROUTING

manip.maniptype=IP_NAT_MANIP_SRC;

manip.manip.ip=202.100.100.1

manip.manip.u.tcp.port=2222

REPLY方向

manip.direction=IP_CT_DIR_REPLY

manip.hooknum=NF_IP_PRE_ROUTING

manip.maniptype=IP_NAT_MANIP_DST;

manip.manip.ip=192.168.1.2

manip.manip.u.tcp.port=3333

返回到函数 ip_nat_fn().接下来的工作就是进行 NAT 处理 do_bindings().do_bindings 根据 ip_conntrack.nat.info.中的 NAT bindings 对数据包进行 NAT 处理。如果对该连接有 nat helper 的情况还需要执行 nat helper 的功能。

do_bindings()不仅要根据已经建立的 NAT bindings 对数据包的地址和端口信息进行转

换，还需要根据是否有 nat helper 进行 ALG 的处理，更改数据包应用层的地址和端口信息。对于第一项工作，将执行前面建立的 SNAT bindings，结果是把原数据包的信息：src/dst/srcprt/dstprt:192.168.1.2/202.100.100.2/3333/21 转换成 src/dst/srcprt/dstprt:202.100.100.1 / 202.100.100.2/2222/21.这里端口进行转换，很可能是 tuple src/dst/srcprt/dstprt:202.100.100.1 / 202.100.100.2/3333/21 已经存在，所以找了一个没有被占用的端口 2222，构成一个没有被使用的元组 src/dst/srcprt/dstprt:202.100.100.1 / 202.100.100.2/2222/21。对于第二项任务，因为没有建立 expection，不会真正被执行。

[ip_nat_core.c------ ip_nat_fn()-→do_bindings()]

```
/* Do packet manipulations according to binding. */
unsigned int
do_bindings(struct ip_conntrack *ct,            enum ip_conntrack_info ctinfo,
            struct ip_nat_info *info,        unsigned int hooknum,
            struct sk_buff **pskb)
{
      struct ip_nat_helper *helper;
      enum ip_conntrack_dir dir = CTINFO2DIR(ctinfo);
      /*dir=IP_CT_DIR_ORIGNAL*/
      int is_tcp = (*pskb)->nh.iph->protocol == IPPROTO_TCP;
      /*is_tcp=true*/

      /* Need nat lock to protect against modification, but neither
          conntrack (referenced) and helper (deleted with
          synchronize_bh()) can vanish. */
      for (i = 0; i < info->num_manips; i++) {
            /* raw socket (tcpdump) may have clone of incoming
                        skb: don't disturb it --RR */
            if (info->manips[i].direction == dir
                  && info->manips[i].hooknum == hooknum) {
                  manip_pkt((*pskb)->nh.iph->protocol,
                        (*pskb)->nh.iph,
                        (*pskb)->len,
                        &info->manips[i].manip,
                        info->manips[i].maniptype,
                        &(*pskb)->nfcache);
            }
      }
      /*
      根据前面建立的NAT bindings，这里会执行SNAT，把源地址转换为202.100.100.1/2222.。具体的工作在manip_pkt中完成。
      */
      helper = info->helper;
      /*helper为ftp的nat helper*/

      if (helper) {
```

```
int ret = NF_ACCEPT;
int helper_called = 0;
list_for_each(cur_item, &ct->sibling_list) {
    /*
    只有建立了expection，并调用了ip_conntrack_expect_related()链表
ct->sibling_list才能有链表表项，所以这里的工作不会执行。
    */
    exp = list_entry(cur_item, struct ip_conntrack_expect,
                expected_list);

    /* if this expectation is already established, skip */
    if (exp->sibling)
        continue;

    if (exp_for_packet(exp, pskb)) {
        /* FIXME: May be true multiple times in the
          * case of UDP!! */
        DEBUGP("calling nat helper (exp=%p) for   packet\n", exp);
        ret = helper->help(ct, exp, info, ctinfo,
                        hooknum, pskb);
        if (ret != NF_ACCEPT) {
            READ_UNLOCK(&ip_conntrack_lock);
            return ret;
        }
        helper_called = 1;
    }
}
/* Helper might want to manip the packet even when there is no
  matching expectation for this packet */
/*
有些应用即使没有expection，也同样需要改变数据包的内容。对于ftp是没有这种必
要的。
*/
if (!helper_called && helper->flags & IP_NAT_HELPER_F_ALWAYS) {
    ret = helper->help(ct, NULL, info, ctinfo,
                    hooknum, pskb);
    if (ret != NF_ACCEPT) {
        return ret;
    }
}


/* Adjust sequence number only once per packet
  (helper is called at all hooks) */
```

```
            /*
            如果是tcp，需要调整序列号。当然也需要更改检验和。
            */
            if (is_tcp && (hooknum == NF_IP_POST_ROUTING
                           || hooknum == NF_IP_LOCAL_IN)) {
                DEBUGP("ip_nat_core: adjusting sequence number\n");
                /* future: put this in a l4-proto specific function,
                 * and call this function here. */
                ip_nat_seq_adjust(*pskb, ct, ctinfo);
            }
            return ret;
        } else return NF_ACCEPT;
        /* not reached */
}
```

至此完成了 NAT 转换，ip_nat_fn()结束，返回到函数 ip_nat_out。在 conntrack helper 中，如果结果的数据包大于 MTU，把数据包分片。最后 ip 层会把数据发送出去。

小结：在收到数据包，经过 conntrack 之后，进入 nat helper。在这些 hook 点 nat helper 注册了回调函数。如果没有建立 NAT 处理规则,则调用 ip_nat_find_rule()需要建立 NAT 处理规则.当 NAT 处理规则建立了之后，调用 do_bindings()进行 NAT 转换。同时 ALG 的 NAT 处理，也会在 do_bindings 中完成。

b) src/dst/srcpt/dstpt:202.100.100.2/202.100.100.1/21/2222 syn+ack server-→client control connection

当客户端向 ftp 服务器发送了 syn 包后，服务器会向客户端发送 syn＋ack 数据包。地址为 202.100.100.2/202.100.100.1/21/2222.首先经过 PREROUTING hook 点,调用在该 hook 点注册的回调函数 ip_nat_fn().

[ip_nat_standalone.c------ ip_nat_fn()]

```
static unsigned int
ip_nat_fn(unsigned int hooknum,
          struct sk_buff **pskb,
          const struct net_device *in,
          const struct net_device *out,
          int (*okfn)(struct sk_buff *))
{
        /* maniptype == DST for prerouting. */
        enum ip_nat_manip_type maniptype = HOOK2MANIP(hooknum);
        /*
        hooknum=NF_IP_PRE_ROUTING
      maniptype= IP_NAT_MANIP_DST
    */
        ct = ip_conntrack_get(*pskb, &ctinfo);
/*
ct=&ct_ctrl_ftp;
```

ct:
IP_CT_DIR_ORIGINAL:src/dst/srcprt/dstprt: 192.168.1.2/202.100.100.2/3333/21
IP_CT_DIR_REPLY: src/dst/srcprt/dstprt: 202.100.100.2/202.100.100.1/21/2222
ct_ftp_ctrl_info->helper=(struct ip_conntrack_helper*)&ftp;
*/
switch (ctinfo) {
default:
info=&ct->nat.info;
/*
先前已经建立的NAT binding信息。
*/
}
/*
进行实际的NAT处理(nat manipulation)
*/
return do_bindings(ct, ctinfo, info, hooknum, pskb);
}

以上可以看到，建立了 NAT bindding 之后，直接进行 NAT 转换就行了。

在 PREROUTING 点 REPLY 方向的 NAT binding 有 DNAT，正好把数据包的信息 src/dst/srcpt/dstpt:202.100.100.2/202.100.100.1/21/2222NAT 转换成 202.100.100.2/192.168.1.2/21/3333.同时由于此时 expection 没有建立，ftp nat helper 和前面的分析一样，不会做什么实际的转换。

[ip_nat_core.c------ ip_nat_fn()-→do_bindings()]

```
/* Do packet manipulations according to binding. */
unsigned int
do_bindings(struct ip_conntrack *ct,            enum ip_conntrack_info ctinfo,
            struct ip_nat_info *info,        unsigned int hooknum,
            struct sk_buff **pskb)
{
    struct ip_nat_helper *helper;
    enum ip_conntrack_dir dir = CTINFO2DIR(ctinfo);
    /*dir=IP_CT_DIR_REPLTY */
    int is_tcp = (*pskb)->nh.iph->protocol == IPPROTO_TCP;
    /*is_tcp=true*/
    for (i = 0; i < info->num_manips; i++) {
        /* raw socket (tcpdump) may have clone of incoming
                    skb: don't disturb it --RR */
        if (info->manips[i].direction == dir
            && info->manips[i].hooknum == hooknum) {
            manip_pkt((*pskb)->nh.iph->protocol,
                    (*pskb)->nh.iph,
                    (*pskb)->len,
                    &info->manips[i].manip,
                    info->manips[i].maniptype,
```

```
                        &(*pskb)->nfcache);
        }
    }
    /*
    在a）中已经在PREROUTING点建立了NAT bindings：
        manip.direction=IP_CT_DIR_REPLY
        manip.hooknum=NF_IP_PRE_ROUTING
        manip.maniptype=IP_NAT_MANIP_DST;
        manip.manip.ip=192.168.1.2
        manip.manip.u.tcp.port=3333
    因此执行DNAT，把目的地址转换为192.168.1.2//3333.。具体的工作在manip_pkt中完成。
    */
    helper = info->helper;
    /*helper为ftp的nat helper*/

    if (helper) {
        int ret = NF_ACCEPT;
        int helper_called = 0;
        list_for_each(cur_item, &ct->sibling_list) {
            /*
            只有建立了expection，并调用了ip_conntrack_expect_related()链表
ct->sibling_list才能有链表表项，所以这里的工作不会执行。
            */
        。。。。。。
        }
        /* Helper might want to manip the packet even when there is no
          matching expectation for this packet */
        /*
        有些应用即使没有expection，也同样需要改变数据包的内容。对于ftp是没有这种必
        要的。
        */
        if (!helper_called && helper->flags & IP_NAT_HELPER_F_ALWAYS) {
            ret = helper->help(ct, NULL, info, ctinfo,
                               hooknum, pskb);
            if (ret != NF_ACCEPT) {
                return ret;
            }
        }

        /* Adjust sequence number only once per packet
          (helper is called at all hooks) */
        /*
        如果是tcp，需要调整序列号。当然也需要更改检验和。
        */
```

```
        if (is_tcp && (hooknum == NF_IP_POST_ROUTING
                       || hooknum == NF_IP_LOCAL_IN)) {
            DEBUGP("ip_nat_core: adjusting sequence number\n");
            /* future: put this in a l4-proto specific function,
             * and call this function here. */
            ip_nat_seq_adjust(*pskb, ct, ctinfo);
        }
        return ret;
    } else return NF_ACCEPT;
    /* not reached */
}
```

数据包已经还会经过FORWARD和POSTROUTING点，但是在FORWARD点nat helper没有注册响应的回调函数。在POSTROUTING，应为没有建立REPLY方向的NATbindings，实际上不会进行任何NAT转换。当数据包变成src/dst/srcprt/dstprt:202.100.100.2/192.168.1.2/21/3333，选择正确的网络接口，数据包可以正确到达机器192.168.1.2,并且目的端口还原为该机器正在等待数据的端口3333。

上面的分析也就是说，reply方向的数据可以通过NAT设备正确到达具有私有地址的内部机器，完成正确的ftp控制连接的建立。但ftp的数据连接是否能够正常建立，还需要进一步往下分析。

c) src/dst/srcpt/dstpt:202.100.100.2/202.100.100.1/21/2222 server-→client control connection 且数据包中有字符串”227 Entering Passive Mode (202,100,100,2,5,6)\r\n”

根据前面的分析，当 conntrack helper 收到这个数据包时，会创建 expection：

exp_ftp_info->ftptype=IP_CT_FTP_PASV;
exp_ftp_info->port=256*5+6=1286;
exp->tuple.src.ip=192.168.1.2
exp->tuple.src.u.all=0;
exp->tuple.dst.ip=202.100.100.2;
exp->tuple.dst.tcp=1286;
exp->protonum=IPPROTO_TCP;
exp->expectant=related_to;
exp->sibling=NULL;

在 PREROUTING 注册的 nat helper 回调函数 ip_nat_fn()被调用，进而 do_bindings()被调用。在 do_bindings()中，REPLY 方向的 NAT bindings 会将数据包的地址信息 NAT 转换为 src/dst/srcprt/dstprt:202.100.100.2/192.168.1.2/21/3333. 同时 nat helper 的 ALG 部分由于 expection 已经建立，将会被执行。

[ip_nat_core.c------ ip_nat_fn()-→do_bindings()]

```
/* Do packet manipulations according to binding. */
unsigned int
do_bindings(struct ip_conntrack *ct,            enum ip_conntrack_info ctinfo,
            struct ip_nat_info *info,       unsigned int hooknum,
            struct sk_buff **pskb)
{
```

```
…… /*NAT转换处理*/
if (helper) {
    list_for_each(cur_item, &ct->sibling_list) {
        exp=list_entry(cur_item,struct ip_conntrack_expect,expected_list);
        /*
        exp就是前面已经建立的expection。
                    exp_ftp_info->ftptype=IP_CT_FTP_PASV;
                    exp_ftp_info->port=256*5+6=1286;
                exp->tuple.src.ip=192.168.1.2
                exp->tuple.src.u.all=0;
                exp->tuple.dst.ip=202.100.100.2;
                exp->tuple.dst.tcp=1286;
                exp->protonum=IPPROTO_TCP;
                exp->expectant=related_to;
                exp->sibling=NULL;
        */
        if(exp_for_packet(exp,pskb)){
            /*
            exp_for_packet()根据协议ip_conntrack_tcp()判断本数据包是否时所期望的(expected).在ip_conntrack_tcp.c的tcp_exp_matches_pkt()可以看到实际上是判断数据包的序列号是否包含了exp->seq.即exp->seq在tcp->seq和tcp->seq+datalen之间。在ip_conntrack_ftp.c的help()中，可以看到：
                    exp->seq=ntohl(tcph->seq)+matchoff.
            matchoff为字符串"227 Entering Passive Mode (202,100,100,2,5,6)\r\n" 中202的位置，显然不会大于datalen。所以，这个数据报正是我们expect的数据包。
            */
            ret=helper->help(ct,exp,info,ctinfo,hooknum,pskb);
            /*调用ip_nat_ftp.c中的help(),也就是ftp ALG的nat核心部分了*/
        }
    }
      ……
    return ret;
} else return NF_ACCEPT;
/* not reached */
}
```

可以想象ftp ALG所要完成的任务就是修改应用层与IP地址和端口信息的部分，使之与NAT bindings中的信息一致。从代码中可以看出，ftp ALG的作者比较paranoia☺，但常常无论我们怎么小心都不为过.一些正确性检查(sanity)实际上在help()中是没有必要的。

[ip_nat_ftp.c------ ip_nat_fn()-→do_bindings()→help()]

```
static unsigned int help(struct ip_conntrack *ct,      struct ip_conntrack_expect *exp,
                         struct ip_nat_info *info, enum ip_conntrack_info ctinfo,
                         unsigned int hooknum, struct sk_buff **pskb)
{
    ct_ftp_info = &exp->help.exp_ftp_info;
```

```
        /* Only mangle things once: original direction in POST_ROUTING
           and reply direction on PRE_ROUTING. */
        dir = CTINFO2DIR(ctinfo);
        /*dir=IP_CT_DIR_REPLY*/

        datalen = (*pskb)->len - iph->ihl * 4 - tcph->doff * 4;
        /* If it's in the right range... */
        if (between(exp->seq + ct_ftp_info->len,
                    ntohl(tcph->seq),
                    ntohl(tcph->seq) + datalen)) {
            /*如果不是一个数据包的一部分，就修正ftp的数据部分了。*/
            if (!ftp_data_fixup(ct_ftp_info, ct, pskb, ctinfo, exp)) {
                UNLOCK_BH(&ip_ftp_lock);
                return NF_DROP;
            }
        } else {
            ……
        }
        UNLOCK_BH(&ip_ftp_lock);

        return NF_ACCEPT;
}
```

ftp ALG要处理的情况有四种：port，eport，pasv，epasv。针对不同的类型，相应的动作也不同。下面的分析以类型IP_CT_FTP_PASV为例。

[ip_nat_ftp.c------ ip_nat_fn()-→do_bindings()→help()－>ftp_data_fixup()]

```
static int ftp_data_fixup(const struct ip_ct_ftp_expect *ct_ftp_info,
                          struct ip_conntrack *ct,      struct sk_buff **pskb,
                          enum ip_conntrack_info ctinfo,      struct ip_conntrack_expect *expect)
{
        u_int32_t newip;
        struct iphdr *iph = (*pskb)->nh.iph;
        struct tcphdr *tcph = (void *)iph + iph->ihl*4;
        u_int16_t port;
        struct ip_conntrack_tuple newtuple;

        /* Change address inside packet to match way we're mapping
           this connection. */
        if (ct_ftp_info->ftptype == IP_CT_FTP_PASV
            || ct_ftp_info->ftptype == IP_CT_FTP_EPSV) {
            /* PASV/EPSV response: must be where client thinks server
               is */
            newip = ct->tuplehash[IP_CT_DIR_ORIGINAL].tuple.dst.ip;
            /* Expect something from client->server */
```

```
            newtuple.src.ip =
                ct->tuplehash[IP_CT_DIR_ORIGINAL].tuple.src.ip;
            newtuple.dst.ip =
                ct->tuplehash[IP_CT_DIR_ORIGINAL].tuple.dst.ip;
            /*
                由于本数据包的方向是server→client,而且期望的数据连接是client->server,所
            以expection的tuple应该如下：
                newip=202.100.100.2
                newtuple.src.ip=192.168.1.2
                newtuple.dst.ip=202.100.100.2
             */
        } else {
            …….
        }
        newtuple.dst.protonum = IPPROTO_TCP;/*TCP*/
        newtuple.src.u.tcp.port = expect->tuple.src.u.tcp.port;/*0*/

        /* Try to get same port: if not, try to change it. */
        for (port = ct_ftp_info->port; port != 0; port++) {
            newtuple.dst.u.tcp.port = htons(port);
            if (ip_conntrack_change_expect(expect, &newtuple) == 0)
                break;
        }
        /*
            目的port是什么？当然最好是控制连接中指定的1286，即ct_ftp_info->port.当时如果其他
的expection信息已经占用了该port,选用其他的也无妨。应为client不知道我们换了。但服务器
可能不知道，因为它还以为client的数据连接的目的port为1286。怎么办?当数据连接到来的
时候，根据exp_ftp_info->port信息转换为1286????稍后的分析也许可以给我们一个正确的答
案。
        */
        if (port == 0)
            return 0;

        if (!mangle[ct_ftp_info->ftptype](pskb, newip, port,
                                expect->seq - ntohl(tcph->seq),
                                ct_ftp_info->len, ct, ctinfo))
            return 0;
        /*调用mangle_rfc595_packet()修正ftp data的内容*/

        return 1;
}
```

manage_rfc595_packet()根据新的端口和 IP 地址修改 ftp data 内容。本例中，实际上 IP 地址没有改变为 202.100.100.2,端口没有改变仍然为 1286。但正如前面提到的，如果有相同的 expection，需要选择其他的端口信息，这时需要调用这个函数。该函数的首先构成正确

的端口和 IP 地址字符串，然后调用 ip_nat_mangle_tcp_packet() 修改数据包。Ip_nat_mangle_tcp_packet()把数据部分偏移为 matchoff 的长度为 matchlen 的数据，替换成 buffer 的内容，当然还需要更新 tcp 和 ip 校验和。

[ip_nat_ftp.c------ ip_nat_fn()-→do_bindings()→help() ー > ftp_data_fixup()->mangle_rfc959_packet ()]

```
static int
mangle_rfc959_packet(struct sk_buff **pskb,
                     u_int32_t newip,           u_int16_t port,
                     unsigned int matchoff,         unsigned int matchlen,
                     struct ip_conntrack *ct,        enum ip_conntrack_info ctinfo)
{
    char buffer[sizeof("nnn,nnn,nnn,nnn,nnn,nnn")];
    sprintf(buffer, "%u,%u,%u,%u,%u,%u",
        NIPQUAD(newip), port>>8, port&0xFF);
    /*修改的结果为 “ 202,100,100,2,5,6 ” */
    return ip_nat_mangle_tcp_packet(pskb, ct, ctinfo, matchoff,
                                    matchlen, buffer, strlen(buffer));
}
```

小结：在 do_bindings()中，当根据 NAT bindings 对数据包进行 NAT 处理时，如果有匹配的 nat helper，并且已经在 conntrack helper 中建立了 expection，需要调用 exp_for_packet()判断是否需要对数据包进行 ALG 相关的应用层数据内容修改。对于 ftp ALG，调用的时 ip_nat_ftp.c 中的 help 函数对 ftp data 内容进行修改 ftp_data_fixup().记住，ftp expection 中的端口改变了，怎么重新转换为端口 1286 的问题还没有解决。

d) src/dst/srcprt/dstprt: 192.168.1.2/202.100.100.2/1111/1286 syn data connection

当 ftp 服务器向 ftp client 发送了”227 Entering passive mode 202,100,100,2,5,6”之后，client 会向 ftp 服务器发起数据连接，该连接的目的为 202.100.100.2/1286.在 conntrack helper 中，会正确的把这条新的数据连接相关于(relate to)前面已经建立的控制连接。具体步骤参见 ip_conntrack_core.c 的 init_conntrack()函数，在链表 ip_conntrack_expect_list 中查找是否本连接是否是被其他已经建立的连接所期望的。如果是，则执行语句：

```
ct_ftp_data_info.master=expected;
expected->sibling=&ct_ftp_datainfo;
```

接下来在 resolve_normal_ct()中设置 ctinfo 为 IP_CT_RELATED。

进入 nat helper 中，首先执行 PREROUTING 点注册的回调函数 ip_nat_fn()。该函数的都用不过是如果没有建立 NAT bindings 建立。建立之后，执行 NAT 转换。问题在于这里应该如何建立 NAT bindings？这条数据连接和一般的连接不一样，它和控制连接有关系，不能仅仅根据 nat table 来查找规则，否则它就会有问题，因为有些应用对与它相关的连接做一些和 nat table 规则不同的处理(本例中，基本上是可以的。一种例外是如果前面的 expection 进行了改变，要数据连接正常工作，需要把目的端口转换为 1286，否则 ftp 服务器会不理解数据连接。)

[ip_nat_standalone.c------ip_nat_fn()]

```
static unsigned int
ip_nat_fn(unsigned int hooknum,     struct sk_buff **pskb,
```

```
        const struct net_device *in,    const struct net_device *out,
        int (*okfn)(struct sk_buff *))
{
    enum ip_nat_manip_type maniptype = HOOK2MANIP(hooknum);
    /*manip_type=IP_NAT_MANIP_DST*/

    ct = ip_conntrack_get(*pskb, &ctinfo);
    /*
    Ct_ftp_data_info:
    IP_CT_DIR_ORIGINAL:src/dst/srcprt/dstprt: 192.168.1.2/202.100.100.2/1111/1286
    IP_CT_DIR_REPLY: src/dst/srcprt/dstprt: 202.100.100.2/192.168.1.2/1286/1111
    ct_ftp_data_info->helper=NULL;
    Expected:
    exp_ftp_info->port=256*5+6=1286;
    exp->tuple.src.ip=192.168.1.2
    exp->tuple.src.u.all=0;
    exp->tuple.dst.ip=200.100.100.2;
    exp->tuple.dst.tcp=1286;
    exp->protonum=IPPROTO_TCP;
    */

    switch (ctinfo) {
    case IP_CT_RELATED:
        info = &ct->nat.info;
        /*还没有建立NAT bindings，所以执行响应的NAT bindings处理。*/
        if (!(info->initialized & (1 << maniptype))) {
            if (ct->master&& master_ct(ct)->nat.info.helper
                && master_ct(ct)->nat.info.helper->expect) {
                /*显然本连接是一个被expected的连接，并且有expect()回调函数，调用
expect()函数了*/
                ret = call_expect(master_ct(ct), pskb, hooknum, ct, info);
            } else {
                …….
            }
        }
        break;
    default:
        ….
    }
    return do_bindings(ct, ctinfo, info, hooknum, pskb);
}
```

call_expect()函数实际上调用的是ALG nat helper中的expect回调函数。本例中调用的是ip_nat_ftp.c中的ftp_nat_expected()函数。

[ip_nat_standalone.c------ip_nat_fn()→call_expect()]

```
static inline int call_expect(struct ip_conntrack *master, struct sk_buff **pskb,
                              unsigned int hooknum,    struct ip_conntrack *ct,
                              struct ip_nat_info *info)
{
        return master->nat.info.helper->expect(pskb, hooknum, ct, info);
}
[ip_nat_ftp.c------ip_nat_fn()→call_expect()→ftp_nat_expected]
static unsigned int
ftp_nat_expected(struct sk_buff **pskb, unsigned int hooknum,
                 struct ip_conntrack *ct,  struct ip_nat_info *info)
{
        struct ip_conntrack *master = master_ct(ct);
        /*
        master=&ct_ftp_ctrl_info:
        ct_ftp_ctrl_info:
        IP_CT_DIR_ORIGINAL:src/dst/srcprt/dstprt: 192.168.1.2/202.100.100.2/3333/21
        IP_CT_DIR_REPLY: src/dst/srcprt/dstprt: 202.100.100.2/202.100.100.1/21/2222
        ct_ftp_ctrl_info->helper=(struct ip_conntrack_helper*)&ftp;
        */
        exp_ftp_info = &ct->master->help.exp_ftp_info;
        /*
        Expected:
        exp_ftp_info->port=256*5+6=1286;
        exp_ftp_info->ftp_type=IP_CT_FTP_PASV;
        */

        if (exp_ftp_info->ftptype == IP_CT_FTP_PORT
                || exp_ftp_info->ftptype == IP_CT_FTP_EPRT) {
                /* PORT command: make connection go to the client. */
                newdstip = master->tuplehash[IP_CT_DIR_ORIGINAL].tuple.src.ip;
                newsrcip = master->tuplehash[IP_CT_DIR_ORIGINAL].tuple.dst.ip;
        } else {
                /* PASV command: make the connection go to the server */
                newdstip = master->tuplehash[IP_CT_DIR_REPLY].tuple.src.ip;
                newsrcip = master->tuplehash[IP_CT_DIR_REPLY].tuple.dst.ip;
                /*
                newdstip=202.100.100.2
                newsrcip=202.100.100.1
                */
        }
        UNLOCK_BH(&ip_ftp_lock);

        if (HOOK2MANIP(hooknum) == IP_NAT_MANIP_SRC)
                newip = newsrcip;
```

```
        else
                newip = newdstip;
        /*newip=202.100.100.2*/
        mr.rangesize = 1;
        /* We don't want to manip the per-protocol, just the IPs... */
        mr.range[0].flags = IP_NAT_RANGE_MAP_IPS;
        mr.range[0].min_ip = mr.range[0].max_ip = newip;/*202.100.100.2*/

        /* ... unless we're doing a MANIP_DST, in which case, make
           sure we map to the correct port */
        /*这就是如果在ip_conntrack_change_expect()中改变了端口，需要把端口转换成原来
server设定的端口，该端口存放在exp_ftp_info->port中*/
        if (HOOK2MANIP(hooknum) == IP_NAT_MANIP_DST) {
                mr.range[0].flags |= IP_NAT_RANGE_PROTO_SPECIFIED;
                mr.range[0].min = mr.range[0].max
                        = ((union ip_conntrack_manip_proto)
                                { .tcp = { htons(exp_ftp_info->port) } });
        }
        return ip_nat_setup_info(ct, &mr, hooknum);
        /*建立DNAT的NAT bindings。不过再次提醒，这里不是根据NAT table建立NAT binding，
而且根据ALG的特定规则。实际上在PREOUTING的NAT binding只是转换可能改变的端口，
IP地址不需要做任何改变，并且这里会把ip_conntrack->nat.info.initialized的
IP_NAT_MANIP_DST置位。在本例中，由于没有改变端口，实际上不会建立PREROUTING
的NAT bindings。 */
}
```

接下来的工作就是调用do_bindings()进行NAT转换了，因为没有建立NAT bindings，不会作转换。然后进入在POSTROUTING点注册的回调函数ip_nat_out(),进而调用函数ip_nat_fn()。在函数ip_nat_fn()中，ctinfo=IP_CT_RELATED，继续调用call_expect()建立NAT bindings。对于ftp ALG调用的是ftp_nat_expected().

```
[ip_nat_standalone.c------ip_nat_out()→ip_nat_fn()→call_expect()]
static inline int call_expect(struct ip_conntrack *master, struct sk_buff **pskb,
                              unsigned int hooknum,    struct ip_conntrack *ct,
                              struct ip_nat_info *info)
{
        return master->nat.info.helper->expect(pskb, hooknum, ct, info);
}
[ip_nat_ftp.c------ip_nat_out()->ip_nat_fn()→call_expect()→ftp_nat_expected]
static unsigned int
ftp_nat_expected(struct sk_buff **pskb, unsigned int hooknum,
                 struct ip_conntrack *ct,  struct ip_nat_info *info)
{
        struct ip_conntrack *master = master_ct(ct);
        /*
        master=&ct_ftp_ctrl_info:
```

```
ct_ftp_ctrl_info:
IP_CT_DIR_ORIGINAL:src/dst/srcprt/dstprt: 192.168.1.2/202.100.100.2/3333/21
IP_CT_DIR_REPLY: src/dst/srcprt/dstprt: 202.100.100.2/202.100.100.1/21/2222
ct_ftp_ctrl_info->helper=(struct ip_conntrack_helper*)&ftp;
*/
exp_ftp_info = &ct->master->help.exp_ftp_info;
/*
Expected:
exp_ftp_info->port=256*5+6=1286;
exp_ftp_info->ftp_type=IP_CT_FTP_PASV;
*/

if (exp_ftp_info->ftptype == IP_CT_FTP_PORT
    || exp_ftp_info->ftptype == IP_CT_FTP_EPRT) {
    /* PORT command: make connection go to the client. */
    newdstip = master->tuplehash[IP_CT_DIR_ORIGINAL].tuple.src.ip;
    newsrcip = master->tuplehash[IP_CT_DIR_ORIGINAL].tuple.dst.ip;
} else {
    /* PASV command: make the connection go to the server */
    newdstip = master->tuplehash[IP_CT_DIR_REPLY].tuple.src.ip;
    newsrcip = master->tuplehash[IP_CT_DIR_REPLY].tuple.dst.ip;
    /*
    newdstip=202.100.100.2
    newsrcip=202.100.100.1
    */
}
UNLOCK_BH(&ip_ftp_lock);

if (HOOK2MANIP(hooknum) == IP_NAT_MANIP_SRC)
    newip = newsrcip;
else
    newip = newdstip;
/*newip=202.100.100.1*/
mr.rangesize = 1;
/* We don't want to manip the per-protocol, just the IPs... */
mr.range[0].flags = IP_NAT_RANGE_MAP_IPS;
mr.range[0].min_ip = mr.range[0].max_ip = newip;/*202.100.100.1*/

/* ... unless we're doing a MANIP_DST, in which case, make
   sure we map to the correct port */
/*这就是如果在ip_conntrack_change_expect()中改变了端口，需要把端口转换成原来
server设定的端口，该端口存放在exp_ftp_info->port中*/
if (HOOK2MANIP(hooknum) == IP_NAT_MANIP_DST) {
    mr.range[0].flags |= IP_NAT_RANGE_PROTO_SPECIFIED;
```

```
            mr.range[0].min = mr.range[0].max
                = ((union ip_conntrack_manip_proto)
                    { .tcp = { htons(exp_ftp_info->port) } });
        }
        return ip_nat_setup_info(ct, &mr, hooknum);
        /*由于需要改变源地址，会在这里建立NAT bindings：
ORIGINAL方向：
            manip.direction=IP_CT_DIR_ORIGINAL
            manip.hooknum=NF_IP_POST_ROUTING
            manip.maniptype=IP_NAT_MANIP_SRC;
            manip.manip.ip=202.100.100.1
            manip.manip.u.tcp.port=4444
REPLY方向
            manip.direction=IP_CT_DIR_REPLY
            manip.hooknum=NF_IP_PRE_ROUTING
            manip.maniptype=IP_NAT_MANIP_DST;
            manip.manip.ip=192.168.1.2
            manip.manip.u.tcp.port=1111
        同时数据连接的conntrack信息也会调用ip_conntrack_alter_reply()修改为：
        Ct_ftp_data_info:
        IP_CT_DIR_ORIGINAL:src/dst/srcprt/dstprt: 192.168.1.2/202.100.100.2/1111/1286
        IP_CT_DIR_REPLY: src/dst/srcprt/dstprt: 202.100.100.2/202.100.100.1/1286/4444
        */
}
```

同样建立了POSTROUTING的NAT bindings之后，调用do_bindings()执行NAT转换。此时需要进行的是SNAT，把原数据包的地址信息 src/dst/srcprt/dstprt:192.168.1.2/202.100.100.2/1111/1286 NAT转换成src/dst/srcprt/dstprt::202.100.100.1/202.100.100.2/4444/1286.当该数据报到达ftp server时，服务器知道这是一个建立数据连接请求，然后如果该连接允许，服务器将向客户端发送syn＋ack数据包。

e) src/dst/srcprt/dstprt: 202.100.100.2/202.100.100.1/1286/4444 syn+ack　　data connection

在 resolve_normal_ct()中，设置 ctinfo 为 IP_CT_ESTABLISHED+IP_CT_IS_REPLY.执行 PREOUTING 点的 hook 回调函数 ip_nat_fn() 时，由于已经建立了 NAT bindings(ip_conntrack->nat.info.initialized|(1<<HOOK2MANIP(hooknum)!=0),直接执行先前建立的 REPLY 方向的 NAT bindings。即把原数据包的地址信息由 src/dst/srcprt/dstprt: 202.100.100.2/202.100.100.1/1286/4444 NAT 转换为 src/dst/srcprt/dstprt: 202.100.100.2/192.168.1.2/1286/1111.在 POSTOURING 的回调函数 ip_nat_out()中调用 ip_nat_fn(),这时候已经处理 POSTROUTING 点的 SNAT binding 的建立，但在 POSTROUTING 不用作什么操作，但这里显然可以正确达到客户端 192.168.1.2/1111.然后 ftp 客户端发送 ack 数据包也可以正确无误地到达服务器，连接建立之后，就可以正确地使用 ftp 传输数据文件了。

至此，以后的数据连接的 NAT binding 已经建立，可以正确在 ftp 服务器和 ftp 客户端之间正确地传输数据，也就是说 ftp ALG 的任务胜利完成。祝贺！同喜！

(To be continued)

ALG Conntrack Helper 的构成

对于每一个 ALG 的 conntrack helper 都是调用 ip_conntrack_helper_register()向系统注册.具体的，它是把这个 helper 放在 ip_conntrack_helper.c 的链表 helpers 中，以后 ip_ct_find_helper()也就是在该链表中查找(见 init_conntrack())。查找的依据就是使用当前的 tuple 去比较已经注册的 conntrack helper 的各个参数。

每一个 conntrack helper 是用结构 ip_conntrack_helper 来表示。

[linux/include/linux/netfilter_ipv4/ip_conntrack_helper.h]

```
struct ip_conntrack_helper
{
    struct list_head list;
    const char *name;
    unsigned char flags;
    struct module *me;
    unsigned int max_expected;
    unsigned int timeout;
    struct ip_conntrack_tuple tuple;
    struct ip_conntrack_tuple mask;
    int (*help)(…);
}
```

flags----为 0，或者为 IP_CT_HELPER_F_REUSE 0x01;后者决定当同一连接建立的悬而未决的 expection 太多时，有新的 expection 要加入时，是否允许重用这些 expection。

max_expected---允许的悬而未决的最大 expection 数(related_to->expecting)。

timeout-------是否需要定时器。如果希望建立的 expection 需要超时，设置为 1。否则不设置，即 timeout 清 0。例如希望 ftp 的 data 连接在控制连接指定了端口和 IP 地址之后的一段时间内必须到达，则需要设置 timeout 为 1。

tuple 和 mask 就是用来和数据包的 tuple 进行比较的依据。注意，这里的 tuple 是与服务器端的响应 tuple 比较的。如果需要比较那个参数，则在 mask 的响应位置置 0xffff 例如假定要比较源地址，则 tuple.src.ip=XXXXX.mask.src.ip=0xFFFF.

Help----这是一个回调函数，如果通过与 tuple 和 mask 比较，符合条件，则需要调用这个回调函数进行处理。该函数在 ip_conntrack_in()中被调用,对应的 hook 点就是 PREROUTING 或者 LOCALIN。Help 的主要作用是检查该数据包的应用层部分，查看应用层的内容是否包含需要处理的 IP 地址和端口信息。如果有，则建立 expection。当然这个 expection 可能会在 nat helper 被修改，但 conntrack helper 并不关心这些。

以 ftp ALG 为例，conntrack helper 的设置为：

```
ftp[i].tuple.src.u.tcp.port = htons(ports[i]);//21
ftp[i].tuple.dst.protonum = IPPROTO_TCP;
ftp[i].mask.src.u.tcp.port = 0xFFFF;
ftp[i].mask.dst.protonum = 0xFFFF;
ftp[i].max_expected = 1;
ftp[i].timeout = 0;
ftp[i].flags = IP_CT_HELPER_F_REUSE_EXPECT;
ftp[i].me = ip_conntrack_ftp;
ftp[i].help = help;
```

对于 ftp ALG，需要处理的是 ftp 控制连接的数据包，服务器方发送的 ftp 控制连接的特征是源端口为 21，目的协议为 TCP。建立悬而未决的 expection 最大为 1，不需要考虑超时问题。

[ip_conntrack_ftp.c------init()]

```
static int __init init(void)
{
    for (i = 0; (i < MAX_PORTS) && ports[i]; i++) {
        ftp[i].tuple.src.u.tcp.port = htons(ports[i]);
        ftp[i].tuple.dst.protonum = IPPROTO_TCP;
        ftp[i].mask.src.u.tcp.port = 0xFFFF;
        ftp[i].mask.dst.protonum = 0xFFFF;
        ftp[i].max_expected = 1;
        ftp[i].timeout = 0;
        ftp[i].flags = IP_CT_HELPER_F_REUSE_EXPECT;
        ftp[i].me = ip_conntrack_ftp;
        ftp[i].help = help;
        ret = ip_conntrack_helper_register(&ftp[i]);
    }
    return 0;
}
```

Help()函数需要处理的每个 ftp 控制连接数据包，如果该数据包的数据部分有地址信息，这些信息一般为新的连接作准备，则建立 exption，以使后面的数据包可以正确作 NAT 转换。

[ip_conntrack_ftp.c------init()->help()]

```
static int help(const struct iphdr *iph, size_t len,
        struct ip_conntrack *ct,  enum ip_conntrack_info ctinfo)
{
    /* Until there's been traffic both ways, don't look in packets. */
    if (ctinfo != IP_CT_ESTABLISHED
        && ctinfo != IP_CT_ESTABLISHED+IP_CT_IS_REPLY) {
        return NF_ACCEPT;
    }
    /*对于ftp控制连接，只有在该连接已经建立了，才可能在ftp data中含有地址信息*/
    /* Checksum invalid?    Ignore. */
    /* FIXME: Source route IP option packets --RR */
    if (tcp_v4_check(tcph, tcplen, iph->saddr, iph->daddr,
                  csum_partial((char *)tcph, tcplen, 0))) {
                return NF_ACCEPT;
    }
    /*checksum检查*/

    old_seq_aft_nl_set = ct_ftp_info->seq_aft_nl_set[dir];
    old_seq_aft_nl = ct_ftp_info->seq_aft_nl[dir];
    if ((datalen > 0) && (data[datalen-1] == '\n')) {
        if (!old_seq_aft_nl_set      || after(ntohl(tcph->seq) + datalen, old_seq_aft_nl)) {
```

```
                ct_ftp_info->seq_aft_nl[dir] = ntohl(tcph->seq) + datalen;
                ct_ftp_info->seq_aft_nl_set[dir] = 1;
        }
    }
    if(!old_seq_aft_nl_set ||  (ntohl(tcph->seq) != old_seq_aft_nl)) {
        return NF_ACCEPT;
    }
    /*序列号检查*/

    /* Initialize IP array to expected address (it's not mentioned
           in EPSV responses) */
    array[0] = (ntohl(ct->tuplehash[dir].tuple.src.ip) >> 24) & 0xFF;
    array[1] = (ntohl(ct->tuplehash[dir].tuple.src.ip) >> 16) & 0xFF;
    array[2] = (ntohl(ct->tuplehash[dir].tuple.src.ip) >> 8) & 0xFF;
    array[3] = ntohl(ct->tuplehash[dir].tuple.src.ip) & 0xFF;
    for (i = 0; i < sizeof(search) / sizeof(search[0]); i++) {
        if (search[i].dir != dir) continue;
        found = find_pattern(data, datalen,
                             search[i].pattern,    search[i].plen,
                             search[i].skip, search[i].term,
                             &matchoff, &matchlen,    array,
                             search[i].getnum);
        /*查看ftp data中是否含有需要进行处理的地址信息。如果有指出偏移matchoff和长
度matchlen*/
        if (found) break;
    }
    if (found == -1) {
        return NF_DROP;
    } else if (found == 0) /* No match */
        return NF_ACCEPT;
    /*OK,我们找到了需要处理的地址信息，建立expection信息了。*/
    memset(&expect, 0, sizeof(expect));
    if (htonl((array[0] << 24) | (array[1] << 16) | (array[2] << 8) | array[3])
        == ct->tuplehash[dir].tuple.src.ip) {
        exp->seq = ntohl(tcph->seq) + matchoff;
        exp_ftp_info->len = matchlen;
        exp_ftp_info->ftptype = search[i].ftptype;
        exp_ftp_info->port = array[4] << 8 | array[5];
    } else {
        if (!loose) goto out;
    }

    exp->tuple = ((struct ip_conntrack_tuple)
        { { ct->tuplehash[!dir].tuple.src.ip,
```

```
                { 0 } },
              { htonl((array[0] << 24) | (array[1] << 16)
                  | (array[2] << 8) | array[3]),
                { .tcp = { htons(array[4] << 8 | array[5]) } },
                IPPROTO_TCP }});
    exp->mask = ((struct ip_conntrack_tuple)
          { { 0xFFFFFFFF, { 0 } },
            { 0xFFFFFFFF, { .tcp = { 0xFFFF } }, 0xFFFF }});

    exp->expectfn = NULL;

    /* Ignore failure; should only happen with NAT */
    ip_conntrack_expect_related(ct, &expect);
 out:
    return NF_ACCEPT;
}
```

至此，可以得出结论，conntrack helper是通过ip_conntrack_helper_register()注册在系统中的。其中要求1)ip_conntrack_helper结构指定该ALG需要处理的数据包的特征(以服务器角度)，2)注册一个回调函数来完成对每一个该应用层数据包的中新连接地址信息的处理。该help函数在应用层数据中发现新连接的地址信息时，建立expection，以便当新连接达到时可以把它与当前连接联系起来(relate)，而且正确作NAT转换。该连接的每一个数据包都会调用help函数，或在REROUTING点或LOCALOUT点。

ALG Nat helper 的构成

每一个 nat helper 都是用 struct ip_nat_helper 表示。

[linux/include/linux/netfilter_ipv4/ip_nat_helper.h]

```
struct ip_nat_helper{
        struct list_head list;
        const char *name;
        unsigned char flags;
        struct module *me;
        struct ip_conntrack_tuple tuple;
        struct ip_conntrack_tuple mask;
        unsigned int (*help)(….);
        unsigned int (*expect)(…..);
}
```

上面的数据接种最重要的数据就是用于确定数据包是否属于特定应用的 tuple 和 mask，规则与 conntrack helper 一样。Help 函数指针被主连接(master connection,也就是不被其他连接相关的连接)调用，用来修改应用层数据中的地址和端口信息。Expect 函数指针被 expected 的连接调用，确定新连接需要执行能够的 NAT 特殊处理。

每一个 ip_nat_helper 都调用 ip_nat_helper_register()向系统注册。具体把该结构注册在 ip_nat_helper.c 的链表 helpers 中。在 ip_nat_core.c 的 ip_nat_setup_info()中，当一个连接不被其他连接相关，就会在链表 helpers 中查找相关的 nat helper。例如对于 ftp ALG 中，控制连

接每一次调用 ip_nat_setup_info()的时候都会更新 nat helper 指针(ip_conntrack->nat.info.helper)。查找 nat helper 的过程就是使用 reply 方向的 tuple(因为以服务器的角度比较)与 nat helper 中的 tuple 和 mask 比较，看它们是否时根据 mask 相等。

以 ftp ALG 为例，ftp nat helper 的设置为：

```
ftp[i].tuple.dst.protonum = IPPROTO_TCP;
ftp[i].tuple.src.u.tcp.port = htons(ports[i]);
ftp[i].mask.dst.protonum = 0xFFFF;
ftp[i].mask.src.u.tcp.port = 0xFFFF;
ftp[i].help = help;
ftp[i].me = THIS_MODULE;
ftp[i].flags = 0;
ftp[i].expect = ftp_nat_expected;
```

以上的设置说明了要处理的每一个主连接数据包具有的特征(以服务器角度)是源端口为 21，协议为 TCP。Help 指针和 expect 指针分别赋值为 Help 函数和 ftp_nat_expected 函数。

[ip_nat_ftp.c------init()]

```
static int __init init(void)
{
    for (i = 0; (i < MAX_PORTS) && ports[i]; i++) {
        ftp[i].tuple.dst.protonum = IPPROTO_TCP;
        ftp[i].tuple.src.u.tcp.port = htons(ports[i]);
        ftp[i].mask.dst.protonum = 0xFFFF;
        ftp[i].mask.src.u.tcp.port = 0xFFFF;
        ftp[i].help = help;
        ftp[i].me = THIS_MODULE;
        ftp[i].flags = 0;
        ftp[i].expect = ftp_nat_expected;

        ret = ip_nat_helper_register(&ftp[i]);
    }
    return ret;
}
```

其中 expect 函数在被 expected 的连接的数据包到来时，在函数 ip_nat_fn()中被调用，调用的次数分别为 2，分别在 PREROUTING 和 POSTROUTING 点被调用，用来建立 DNAT 和 SNAT 的 NAT bindings。对于 ftp ALG，调用的是 ip_nat_ftp.c 中的 ftp_nat_expected().前面的分析已经提到，它实际上是建立 ftp 数据连接的 NAT bindings。对于 IP_CT_FTP_PASV 类型的数据包，真正建立的 NAT bindings 是 Original 方向的 SNAT，和 REPLY 方向的 DNAT。

其中 help()函数在主连接的数据包的处理 do_binding 中被调用，主要的功能是在数据包的应用层数据中修改与 IP 地址和端口的信息。例如对于 ftp ALG，调用的是 ip_nat_ftp.c 中的 help 函数。它被调用两次，分别在 hook 点 PREROUTING 和 POSTROUTING。它主要的功能调用 ftp_data_fixup()中完成，根据 FTP 的不同情况，把数据包中应用层的地址信息修改正确。

一个 ALG 的 nat helper 主要的构成有：1)标记每个需要处理的主连接的特征，例如端口号和协议号。2)expect 函数，处理被期望的相关连接的 NAT bindings 建立处理。3)help 函数对于每一个数据包进行修改主连接应用层数据的地址信息进行修正。

可改进的之处

netfilter 实际上是很成功的，它集成了 Firewall，NAT 和 Mangle 的功能。另外它支持对它的不同粒度的扩展。另一方面和原来 Linux 网络协议栈配合的很一致。还有其他很多很多优点。

但个人意见，ALG 部分有很多不是很清晰的地方。例如，help()函数哪些地方被调用，expect 需要完成哪些功能？问题关键在于这些函数是没有明显地区分不同地 hook 点，当然也许有些应该不能这么区分。但以 ftp ALG 为例，可以看到如果把这些函数分成不同的 hook 点的不同操作，也许不用把 help()函数调用几次导致重复。例如对数据包应用层的地址信息修改至少两次被调用。如果只在 POSTROUTING 点进行调用，这时候所有的 NAT 都已经完成，该转换成什么也大致清楚，那么也许不会有重复调用。ALG 是否能够有一个更清晰的构架，值得思考。