# 一、 改进细节

## 1.1 改进一 前端设计

正如 4.5 节所说，使用 Python argparse 进行参数解析，省去传统命令行顺序输入多参数的繁琐过程，便于与用户交互。同时 argparse 可自定义参数检查功能，不需要在前端代码进行冗余的参数校验，减少代码量。

```
[11/27/23]seed@VM:~/.../app$ python command.py --help
usage: command.py [-h] --object {rule,nat,log,conn} --op
                  {show,add,del,default} [--source_ip SOURCE_IP]
                  [--source_port SOURCE_PORT] [--dest_ip DEST_IP]
                  [--dest_port DEST_PORT] [--name NAME] [--position POSITION]
                  [--protocol {TCP,UDP,ICMP,any}] [--act {accept,deny}]
                  [--def_act {accept,deny}] [--log {yes,not}]
                  [--log_num LOG_NUM] [--nat_num NAT_NUM] [--nat_sip NAT_SIP]
                  [--nat_dip NAT_DIP] [--nat_port NAT_PORT]

command for my firewall

optional arguments:
  -h, --help            show this help message and exit
  --object {rule,nat,log,conn}
                        operating object, eg. rule, nat, log
  --op {show,add,del,default}
                        operation to the object, eg. show, add, delete
  --source_ip SOURCE_IP
                        source IP address with subnet mask, default:
                        192.168.164.2/24
  --source_port SOURCE_PORT
                        source port, eg. 80-80, default: any
  --dest_ip DEST_IP     destination IP address with subnet mask, default:
                        192.168.152.2/24
  --dest_port DEST_PORT
                        dest port, eg. 80-80, default: any
  --name NAME           rule name
  --position POSITION   rule position in list, before xxx(your input)
  --protocol {TCP,UDP,ICMP,any}
                        protocol, eg. TCP,UDP,ICMP,any
  --act {accept,deny}   action, accpet or deny
  --def_act {accept,deny}
                        default action, accpet or deny
  --log {yes,not}       whether to log, yes or not
  --log_num LOG_NUM     line number of showed log
  --nat_num NAT_NUM     nat number
  --nat_sip NAT_SIP     nat source IP address with subnet mask, default:
                        192.168.164.2/24
  --nat_dip NAT_DIP     nat IP address, default: 192.168.80.80
  --nat_port NAT_PORT   nat port, eg. 80-80, default: any
```

**图 8 Python 前端帮助信息**

## 1.2 改进二 日志记录修复

源项目的日志记录模块存在问题，即报文的长度记录异常。

验收时初步判断是**网络字节序没有转换成主机字节序**的问题，的确如此。

定位问题到日志记录函数，可以看到记录长度时，源代码直接使用 IP 报文总长度 header->tot_len 减去报文头部长度 head->ihl，并没有做大小端转换。

```c
int addLogBySKB(unsigned int action, struct sk_buff* skb) {
    struct IPLog log;
    unsigned short sport, dport;
    struct iphdr* header;
    struct timeval now = {.tv_sec = 0, .tv_usec = 0};
    do_gettimeofday(&now);
    log.tm = now.tv_sec;
    header = ip_hdr(skb);
    getPort(skb, header, &sport, &dport);
    log.saddr = ntohl(header->saddr);
    log.daddr = ntohl(header->daddr);
    log.sport = sport;
    log.dport = dport;
    // log.len =  header->tot_len - header->ihl * 4;
    log.len = ntohs(header->tot_len) - header->ihl * 4; // modify here
    log.protocol = header->protocol;
    log.action = action;
    log.nx = NULL;
    return addLog(log);
}
```

**图 9 日志记录函数**

回顾 IP 报文结构，可以知道 tot_len 的类型是 **unsigned_16bit**，单位是**字节**；而 ihl 的类型是 **unsigned_8bit**，单位是 **4 个字节**。所以应该对 tot_len 做大小端转换，并且 ihl 的值应该乘 4。

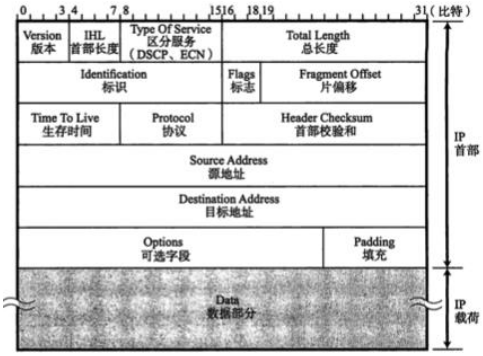


**图 10 IP 报文结构**

修改后重新检查日志记录，配合 wireshark 抓包可以发现包长度计算正确 (tot_len - ihl*4 = 81Byte – 5*4Byte = 61 Byte)，证明修改的正确性。

```
[11/27/23]seed@VM:~/.../app$ python command.py --object log --op show | tail -20
[2023-11-27 10:47:09] [ACCEPT]2023-11-27 10:47:09 192.168.164.2:48036->192.168.152.2:888 proto=TCP len=61B
[2023-11-27 10:47:09] [ACCEPT]2023-11-27 10:47:09 192.168.164.2:48036->192.168.152.2:888 proto=TCP len=61B
[2023-11-27 10:47:09] [ACCEPT]2023-11-27 10:47:09 192.168.152.2:888->192.168.164.2:48036 proto=TCP len=61B
[2023-11-27 10:47:09] [ACCEPT]2023-11-27 10:47:09 192.168.152.2:888->192.168.164.2:48036 proto=TCP len=61B
[2023-11-27 10:47:09] [ACCEPT]2023-11-27 10:47:09 192.168.164.2:48036->192.168.152.2:888 proto=TCP len=61B
[2023-11-27 10:47:09] [ACCEPT]2023-11-27 10:47:09 192.168.164.2:48036->192.168.152.2:888 proto=TCP len=61B
[2023-11-27 10:47:09] [ACCEPT]2023-11-27 10:47:09 192.168.152.2:888->192.168.164.2:48036 proto=TCP len=61B
[2023-11-27 10:47:09] [ACCEPT]2023-11-27 10:47:09 192.168.152.2:888->192.168.164.2:48036 proto=TCP len=61B
[2023-11-27 10:47:09] [ACCEPT]2023-11-27 10:47:09 192.168.164.2:48036->192.168.152.2:888 proto=TCP len=61B
[2023-11-27 10:47:09] [ACCEPT]2023-11-27 10:47:09 192.168.164.2:48036->192.168.152.2:888 proto=TCP len=61B
```

图 10 修改后日志记录

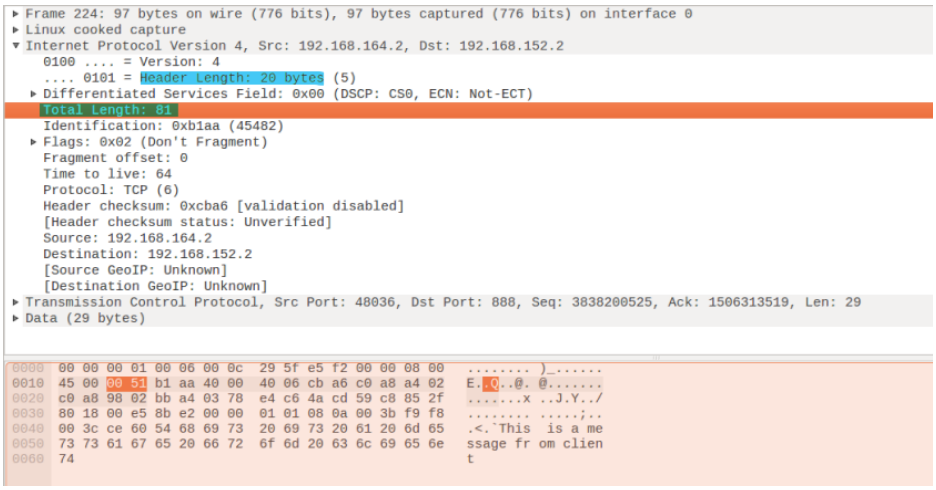


图 11 Wireshark 抓包结果

除此之外，还修改了 hook_main 中的日志记录逻辑，如果连接存在，action 应该是 accept，而不应该是 default_action，因为如果 default_action 是拒绝，那么对于放行的数据包，日志记录同样会是 drop，从而引发歧义。

```
1.  if(conn != NULL) {  //查询到连接，直接放行
2.      action = 1; // modify here
3.      if(conn->needLog) // 记录日志
4.          addLogBySKB(action, skb);
5.      return NF_ACCEPT;
6.  }
```

图 12 日志修改第二处

### 1.3 改进三 包过滤到状态检测

**原项目实现的其实是包过滤防火墙**，虽然使用了连接的概念，但是**连接只是作为规则的一个抽象**。最直观的问题就是：对于原项目而言，在默认拒绝情况下，需要建立两条规则才能允许一条 TCP 连接的建立。这与状态检测的概念是相悖的。

虽然原项目是包过滤防火墙，但是已经封装了连接的结构，我们只需要稍加改进就可以得到一个状态检测的防火墙。

回顾连接的本质，我认为连接的作用是加速报文的处理，对于过去已经匹配规则放行的报文，下次遇到相关报文(如回应包)，不需要再次顺序查询规则表，而是通过查询连接表实现快速转发。因此连接可以视为**通过规则检查的报文的一个 cache**，这也是连接表一般通过哈希等快速查找方法实现的原因，**如果查连接表的速度没有体现出优势，为什么不直接去查规则表呢~** 有了以上概念，对于

连接表、状态机的设计就清楚了。

**对于非 TCP 报文**，第一次遇到时，先查询规则表，如果允许通行，则将四元组(源 IP,源端口,目的 IP,目的端口)记录到连接表中。对于后续报文，如果报文同向，那么使用报文四元组匹配连接表时，就可以匹配到对应的连接；如果报文不同向，连接表是查询不到对应连接的。因为连接表采用的红黑树实现，红黑树的键值是固定的一个四元组。

解决这个问题有两种方案：①第一个方案是在更新连接表时，多添加一条反向的四元组连接，这个虽然可以解决最开始的问题(即两条显式规则才允许一条 TCP 连接通行)，但是一条真正的连接对应连接表中的两个条目，连接表的空间占用翻了一倍，并且查找速度也不会更快(对于红黑树来说)，因此这个方案是次优的；②第二个方案思路更加简单，在查询连接时，**使用双向查询策略**。即查询连接的时候，不仅查询(源 IP,源端口,目的 IP,目的端口)，还查询(目的 IP,目的端口,源 IP,源端口)。代码如图 13。

```
// 查询是否有已有连接
conn = hasConn(sip, dip, sport, dport);
// reverse lookup, modify here
if (conn == NULL) {
    conn = hasConn(dip, sip, dport, sport);
}
if (conn != NULL) {
    if (conn->needLog)  // 记录日志
        addLogBySKB(action, skb);
    return NF_ACCEPT;
}
```

**图 13 双向查询**

**对于 TCP 报文**，除了双向查询策略以外，还需要**额外进行 SYN 包的判断**。具体来说，对于已经存在连接的 TCP 报文，直接放行；如果不存在连接，需要判断该报文是否是 TCP 第一次握手的报文(即 SYN = 1)，如果是，则进入规则匹配阶段，如果放行则更新连接表，匹配失败或非 SYN 包则丢弃。状态机如图 14。

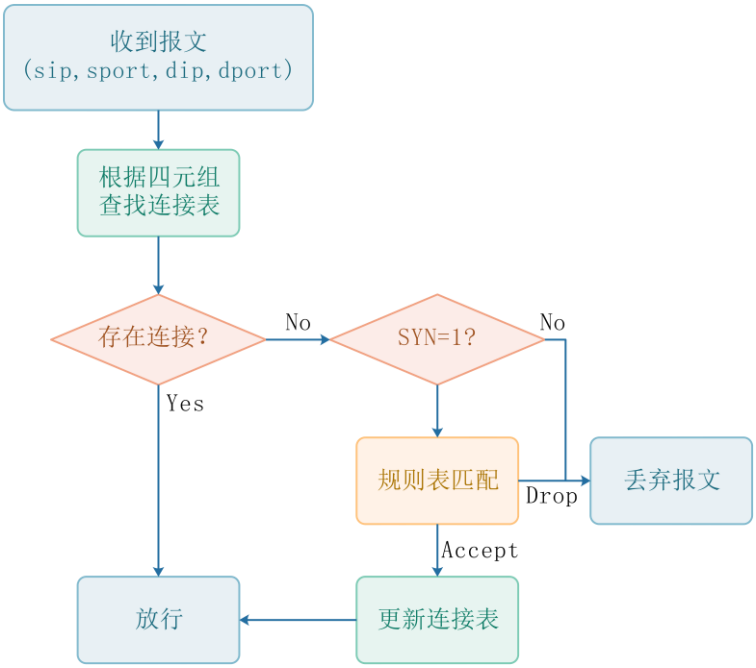


**图 14 TCP 连接状态机**

对应代码如下，放在 hook_main 函数中未找到连接之后，开始规则表匹配之前。这里除了要判断 SYN 标志以外，还需要进一步判断 ACK 标志，因为 ACK=1，SYN=1 代表连接建立，即第三次握手，属于无效包，需要丢弃。

```
// 不存在连接,如果是 TCP 连接，先判断是否是 SYN 包,不是则丢弃
if (header->protocol == IPPROTO_TCP) {
    struct tcphdr* tcphdr = (void*)header + header->ihl * 4;
    if (tcphdr->syn != 1) {  // 非 SYN 包，丢弃
        return NF_DROP;
    } else if (tcphdr->ack != 0) {  // SYN = 1, ACK = 1, 丢弃
        return NF_DROP;
    }
}
```

**图 15 TCP SYN 包代码判断**

### 1.4 改进四 NAT 修复

这一处改进是最复杂且最麻烦的，原项目的 NAT 功能其实是正常的 。只是对于一条经过 NAT 的 TCP 连接，它需要在连接表里创建两个连接，这与直觉相悖。接下来将**从头阐述为什么原项目会(需要)建立两个连接来保证 NAT 功能的实现**。

首先考虑一个经典场景，内网主机 A 访问外网某个主机 B 的公共服务，内网配备一个防火墙 F，并给 A 提供了 NAT 转换功能。那么正常的通信流程应该

是：A 向 B 发送报文，F 修改了报文的源 IP 和源端口，转发给 B；B 接收到报文后，向修改后的源 IP/端口发送回应报文，F 收到回应报文后，修改了回应报文的目的 IP 与端口(即修改成了 A 发送报文时对应的 IP 与端口)，转发给 A。具体流程可见图 16。

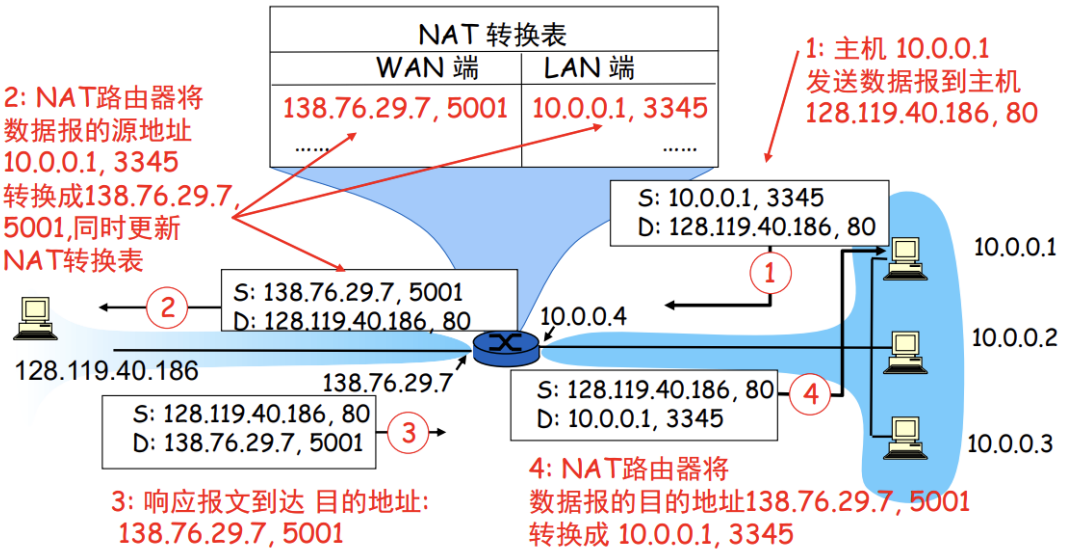


**图 16 正常 NAT 流程**

通过上述的 NAT 场景，我们可以得到防火墙/路由器做 NAT 的一个大致流程：**无论是来自外网还是内网的报文，先做 NAT，修改报文的源(目的)IP 和源(目的)端口，再进行路由转发。**这样的好处在于，**先做 NAT 转换后，留下的四元组(SIP，SPORT,DIP,DPORT)都是真实的，如果存在连接，就一定会在连接表中**。当然也可以统一后做 NAT，只需要**保证 NAT 的流程是对称的**即可。

回到原项目的 NAT 实现，由 4.2 节的讨论可以知道：hook 函数的优先级顺序为：检查连接(hook_main, 路由前) > 检查连接(hook_main, 路由后) > NAT 转换(hook_nat_in, 路由前) > NAT 转换(hook_nat_out，路由后)。这个过程是不对称的，**接下来讨论不对称带来的问题**。

同样选取刚刚的 NAT 场景，即内网主机 A 访问外网 B 服务。

① 首先会执行检查连接操作(hook_main)，建立一条连接，用四元组表示(A_IP,A_PORT,B_IP,B_PORT)；

② 随后执行 NAT 转换操作(hook_nat_out)，在该函数中，报文的源 IP/端口会被替换成 NAT 分配的 IP 与端口，四元组表示为(NAT_IP, NAT_PORT, B_IP, B_PORT)，除此之外，该函数还会记录 NAT 的相关信息到连接表对应的条目中(即①中建立的连接)；

③ 随后该报文会转发给 B，B 向防火墙 F 发送回应报文，F 收到后，执行检查连接操作，**问题在这里便出现了，报文的四元组是(B_IP, B_PORT, NAT_IP,**

**NAT_PORT)，这并不在连接表中，因此会被防火墙拦截**，因为①创建的连接对应的 A 和 B 的 IP 与端口，而回应报文对应的是 NAT 和 B 的 IP 与端口。

为了解决这个问题，**原项目在②操作中添加了一条反向连接**，即往连接表中添加了一条(B_IP, B_PORT, NAT_IP, NAT_PORT)，这保证了在③中，B 的回应报文不会被拦截。从而解决了双向通信的问题。

```
reverseConn = hasConn(dip, record.daddr, dport, record.dport);
if (reverseConn == NULL) {  // 新建反向连接入连接池
    reverseConn = addConn(dip, record.daddr, dport, record.dport, proto, 0);
    if (reverseConn == NULL) {  // 创建反向连接失败，放弃 NAT
        printk(KERN_WARNING "[fw nat] add reverse connection failed!\n");
        return NF_ACCEPT;
    }
    setConnNAT(reverseConn,
               genNATRecord(record.daddr, sip, record.dport, sport),
                NAT_TYPE_DEST);
}
```

**图 17 hook_nat_out 额外添加反向连接**

综上所述，之所以 NAT TCP 会出现两条连接，是因为 NAT 的过程不是对称的，那么修改思路就是将其改成对称的，并去掉 hook_nat_out 中的反向连接。

※首先修改四个 hook 操作的优先级，让 hook_nat_in 的优先级大于 hook_main 的优先级，即发送报文时，先检查连接，再做 NAT；接收报文时，先做 NAT，再检查连接。这里简单交换了 hook_nat_in 和 hook_main 的优先级。

```
static struct nf_hook_ops nfop_in={
    .hook = hook_main,
    .pf = PF_INET,
    .hooknum = NF_INET_PRE_ROUTING,
    .priority = NF_IP_PRI_NAT_DST //modify
};
static struct nf_hook_ops natop_in={
    .hook = hook_nat_in_hj,
    .pf = PF_INET,
     .hooknum = NF_INET_PRE_ROUTING,
     .priority = NF_IP_PRI_FIRST //modify
};
```

**图 18 交换优先级**

※其次是去掉 hook_nat_out 的添加反向连接的逻辑，即图 17。

※再次是重写 hook_nat_in，在该函数中查找 NAT 表，先执行 NAT 的转换。

```
// find record here modify!!!
record = matchNATRule_hj(sip, dip, dport, &isMatch);
if (!isMatch || record == NULL) {  // 不符合 NAT 规则，无需 NAT
    return NF_ACCEPT;
}
header->daddr = htonl(record->saddr);  // modify
```

**图 19 hook_nat_in_hj 查找 NAT 表**

```
struct NATRecord* matchNATRule_hj(unsigned int sip,
                                  unsigned int dip,
                                  unsigned short dport,
                                  int* isMatch) {
    struct NATRecord* now;
    *isMatch = 0;
    read_lock(&natRuleLock);
    for (now = natRuleHead; now != NULL; now = now->nx) {
        if (dip == now->daddr && dport == now->dport) {
            read_unlock(&natRuleLock);
            *isMatch = 1;
            return now;
        }
    }
    read_unlock(&natRuleLock);
    return NULL;
}
```

**图 20 根据目的 IP/端口查找 NAT 表**

※最后需要注意的细节是，四个 hook 操作对同一个报文都会执行，而 NAT 在单个方向只需要做一次，因此需要在 hook_nat_in 和 hook_nat_out 中增加对报文方向的判断，内网到外网方向的报文不需要做 hook_nat_in 操作；外网到内网方向的报文不需要做 hook_nat_out 操作。这里解决方法不太优雅，使用子网判断报文的方向。

```
//////////////////// in hook_nat_in ////////////////////////////////
//magic number 3232277504 = IPstr2IPint('192.168.164.0')
// 如果源 ip 来自内网 192.168.164.0，不需要做 hook_nat_in 操作
if((sip & (unsigned int)3232277504) == (unsigned int)3232277504){
    return NF_ACCEPT;
}
//////////////////// in hook_nat_out////////////////////////////////
// 如果源 ip 不在内网 192.168.164.0，不需要做 hook_nat_out 操作
if((sip & (unsigned int)3232277504) != (unsigned int)3232277504){
    return NF_ACCEPT;
}
```

**图 21 NAT 操作方向判断**

通过以上操作，NAT TCP 功能就算解决了，不会再出现多出一条反向连接的问题，同时 NAT 转换操作也更加符合逻辑。至此，完结撒花~

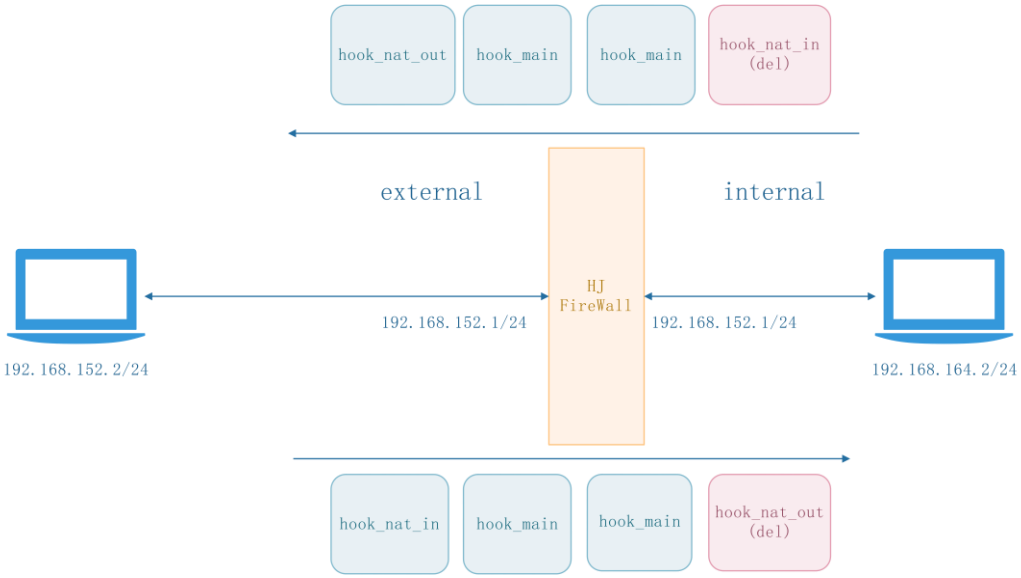


**图 22 NAT 整体操作概览**