IPTABLES 数据结构与流程

解析说明

部　　门: 网络安全线

拟　　制: 2016 年 11 月 21 日

作　　者: 薛萌

版本记录

序号	版本号	生成日期	主 要 修 改 记 录	第一作者	审核人	备注
1	**1.0**	**2016.11.21**	首次生成。	薛萌		
2	**1.1**	**2016.11.25**	增加了对 **hookentry** 和 **underflow** 的详细解释	薛萌		
3	**1.2**	**2016.12.8**	补充了对 **hookentry** 和 **underflow** 的说明	薛萌		
4	**2.0**	**2016.12.21**	增加对 **iptables** 代码配置的解读	薛萌		
5	**2.1**	**2017.6.12**	增加 **iptables** 代码解析的一些说明	薛萌		

引言

解释目前 **CM** 获取 **netfilter** 和解析的代码

相关文档

表 1. 相关设计文档列表

序号	文档名称	版本号	作者	备注
1				
2				

引用的技术标准及规范

表 2. 引用的技术标准及规范

序号	文档名称	版本号	作者	发表日期	出版单位/来源	备注
1						
2						

术语

表 3. 术语及解释

序号	术语	英文	说明
1	CM	Cache management	
2			

适用范围

本文档适用于更新 **CM** 代码时，**netfilter** 规则解析的方法及其数据结构解析和说明。

一、内核数据的获取

1.1、正常的 libevent 事件注册

```
00863:      cm_netlink_sock (NETLINK_ROUTE,
00864:              &cm_netlink,
00865:              (RTMGRP_LINK | RTMGRP_NEIGH |
00866:               RTMGRP_IPV4_ROUTE | RTMGRP_IPV6_ROUTE |
00867:               RTMGRP_IPV4_IFADDR | RTMGRP_IPV6_IFADDR),
00868:              1);
```

```
00831: cm_netlink_sock (int proto, struct nlsock *cmn, long groups, int listen)
00832: {
00833:     if (rtnl_open_byproto(&cmn->rtnl, groups, proto) < 0) {
00834:     /* yzhou modified: vnb isn't supported by native kernels */
00835:         printf("%s: Unable to open netlink socket, proto %d, grp %ld\n",
00836:       __func__, proto, groups);
00837:         return;
00838:     }
00839:     if (listen) {
00840:         event_set (&cmn->ev, cmn->rtnl.fd,
00841:                    EV_READ | EV_PERSIST,
00842:                    cm_nl_recv_event, cmn);
00843:         event_add (&cmn->ev, NULL);
00844:     }
00845:     return;
00846: }
```

图 **1**、**2**

在 **865** 行开始，加入了一个多播组，在这些多播组发生状态改变之类的情况后，会使得 **840** 行的 **fd** 转为可读状态。**841** 行指定了事件的类型，即永久监控可读事件。**fd** 可读后，会调用 **842** 行注册的回调函数进行处理。这里 **event_add** 的第二个参数为空，表明此时没有使用超时相关的事件处理。

而 **netfilter** 消息处理的事件如下：

```
00893: void cm_netlink_netfilter_init(void)
00894: {
00895: #ifdef NF_NETLINK_TABLES
00896:   /* yzhou modified: the NF_NETLINK_TABLES group
00897:    *   was not supported by native kernels, so use
00898:    *   timer instead, for test only */
00899: #ifndef NF_TABLES_REFRESH
00900:     cm_netlink_sock(NETLINK_NETFILTER, &cm_netlink_netfilter_table,
00901:             NF_NETLINK_TABLES, 1);
00902: #else
00903:   event_set(&eNfTbl, 0, EV_TIMEOUT, nftbl_recv, NULL);
00904:   event_add(&eNfTbl, &tv_to);
00905: #endif
00906: #else
00907:     printf("Unknown group: NF_NETLINK_TABLES\n");
00908: #endif
00909:     cm_netlink_sock(NETLINK_NETFILTER, &cm_netlink_netfilter_conntrack,
00910:             (NF_NETLINK_CONNTRACK_NEW | NF_NETLINK_CONNTRACK_UPDATE |
00911:             NF_NETLINK_CONNTRACK_DESTROY), 1);
00912:   /* yzhou test */
00913: }
```

图 **3**

这里由于定义了 **NF_TABLES_REFRESH,** 所以 **900** 行未执行，即没有正常的注册一个 **libevent** 事件。这里 **event_set** 的第二个参数为 **0**，即未关联描述符，且设置为关注超时事件。同时在 **event_add** 里第二个参数中设置了超时时间为 **2**，即两秒触发一次事件。
从后续执行的回调函数可以看到，**2** 秒触发一次的事件为根据表名去获取 **iptables** 命令配置在 **netfilter** 中的规则。

```
void cm_iptc_dump_table(void)
{
#ifdef NF_NETLINK_TABLES
    iptc_dump_table("filter", AF_INET);
    iptc_dump_table("mangle", AF_INET);
    iptc_dump_table("nat", AF_INET);
#endif
}
```

图 **4**

1.2、获取 netfilter 数据

netfilter 数据是通过 **getsockopt** 函数加上特定的协议族来获取的。目前为止，我们并不知道如何去筛选式获取数据，只能是全量的获取数据。想增量获取数据或者增加其他选项可能需要后面更改内核。

两次 **getsockopt** 的意义在 **2.2.2** 中有详细说明， 按目前代码的走向，之后的内容就是拷贝数据到一个 **cp_nftable** 结构中了。这个结构是为了 **CM** 和 **FPM** 交换数据使用的。后面的大致内容就是如何将取得的 **ipt_entry** 条目通过有效性检测后，传给 **fpm**。

二、内核数据的解析

ipt_entry 结构除了基本匹配规则外，还包含了扩展定制的 **match** 和 **target**，我们要从这两个结构中获得相应的数据并发送给 **fpm**

2.1、数据结构格式

2.1.1 ipt_entry 结构

下面看一下 **ipt_entry** 的结构：

```
struct ipt_entry
{
    struct ipt_ip ip;

    /* Mark with fields that we care about. */
    unsigned int nfcache;

    /* Size of ipt_entry + matches */
    u_int16_t target_offset;
```

```
    /* Size of ipt_entry       + matches + target */
    u_int16_t next_offset;

    /* Back pointer */
    unsigned int comefrom;

    /* Packet and byte counters. */
    struct xt_counters counters;

    /* The matches (if any), then the target. */
    unsigned char elems[0];
};
```

这就是 **netfilter** 中存放规则的数据结构 一条就是一个 **rule**

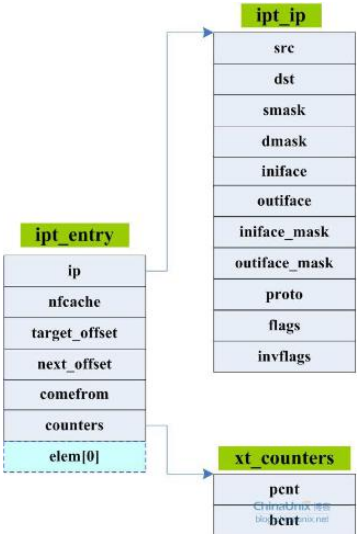

图 **5**

上图就是 **ipt_entry** 头的结构，实际的 **match** 和 **target** 紧跟在后面。

struct ipt_ip ip ：这是标准的匹配结构，主要包含数据包的源、目的 **IP**，出、入接口和掩码等

u_int16_t target_offset： **target** 开始的位置（偏移）即 **sizeof(ipt_entry + n*match)**， 实际的数据存放情况是，**match** 地址紧接在 **ipt_entry** 地址后面

u_int16_t next_offset：下一条规则（**ipt_entry**）的位置。

通过以上三个位置，就可以对取出的消息进行处理了。

unsigned int comefrom ：判断规则链是否存在环路，也在用户自定链执行完返回主链时使用。

以下是 **ipt_ip** 结构：

ipt_ip

```
struct ipt_ip {
    /* Source and destination IP addr */
    struct in_addr src, dst;
```

```
    /* Mask for src and dest IP addr */
    struct in_addr smsk, dmsk;
    char iniface[IFNAMSIZ], outiface[IFNAMSIZ];
    unsigned char iniface_mask[IFNAMSIZ], outiface_mask[IFNAMSIZ];

    /* Protocol, 0 = ANY */
    u_int16_t proto;

    /* Flags word */
    u_int8_t flags;
    /* Inverse flags */
    u_int8_t invflags;
};
```

ipt_entry 结构用于存储链的规则，每一个包过滤规则分为两个部分：

match，即条件，分别为 ：

1、基本元素，也即标准 **match**，如源**/**目的地址，网口，协议，对应 **ipt_ip**

2、扩展定制的 **match**，比如 **tcp,udp,icmp** 等，这就是我们后面要加入的 **match** 规则了

target，即动作， 也是两以上的两个部分。这里不赘述

那么一条规则占用空间，即 **ipt_entry** 结构 **+** **n***扩展 **match** **+** **n***扩展 **target**

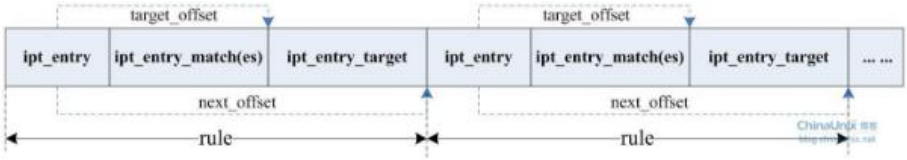

图 **6**

2.1.2 match 和 target 结构

下图是后面跟的 **match** 和 **target** 联合体结构

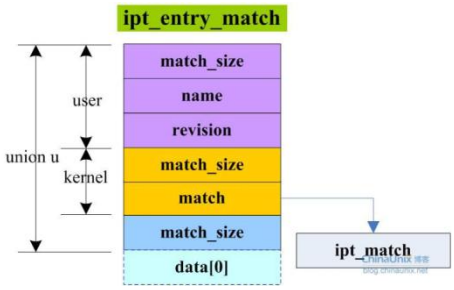

图 **7**

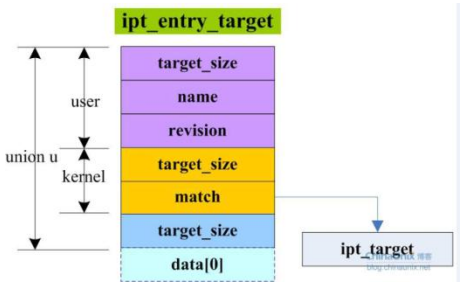

图 8

其中，**match** 指针指向的 **ipt_match** 是匹配的内核数据结构，而 **iptables_match** 是用户空间中的数据结构。当某个具体的 **match** 被应用到防火墙规则时，统一成 **ipt_entry_match**，才是防火墙规则中用到的 **match** 结构，这些名称在代码中并未体现。

另外，结构体定义中可以看到分别为 **user** 和 **kernel** 定义了不同的联合体结构。

```
struct xt_entry_match
{
    union {
        struct {
            __u16 match_size;

            /* Used by userspace */
            char name[XT_FUNCTION_MAXNAMELEN-1];

            __u8 revision;
        } user;
        struct {
            __u16 match_size;

            /* Used inside the kernel */
            struct xt_match *match;
        } kernel;

        /* Total length */
        __u16 match_size;
    } u;

    unsigned char data[0];
};
```

以 **match** 举例，**match_size** 是两者都有的，用户态的时候，在 **name** 中设置 **match** 名称，内核添加规则时就会使用这个 **name** 去查找匹配相应规则。其中 **kernel** 中的 **xt_match** 指向相应的 **ipt_match** 结构。同样 **target** 也有 **ipt_target** 和 **iptables_target** 结构。

对于 **target**，也分内建动作，和跳转到自定义链。标准 **target** 就是内核内建的一些处理动作或者延伸，而扩展 **target** 完全是用户自定义的处理动作，即允许某条 **target** 可以跳转到用户指定的自定义链上，这时候这条规则的 **target** 就是 **verdict** 值大于 **0** 的 **ipt_stand_target**

类型，处理报文后使用 **verdict** 将结果返回给 **netfilter** 框架。如果 **ipt_target.target()**函数是空的,即标准 **target**，不需要用户提供新的 **target** 函数。用户可以通过实例化一个 **ipt_target** 对象并填充相关回调函数，再注册即可实现自己的 **target**。

当 **verdict** 为小于 **0** 时，则 **verdict** 则为数据包下一步的执行操作；当 **verdict** 大于 **0** 时，则说明该 **target** 需要跳转到一个用户自定义链的链首地址，其值为用户自定义链相对于表的第一条链规则的偏移量。

使用 **target** 时，可以用 **ipt_standard_target** 或者 **ipt_entry_target** 结构体进行指针转换使用，例如：

```
00106: static int ip4tc_copy_target(struct ipt_entry_target *t,
00107:                   struct cp_nfrule *rule,
00108:                   struct ipt_get_entries *entries)
00109: {
00110:     /* Standard target */
00111:     if (!strcmp(t->u.user.name, IPT_STANDARD_TARGET)) {
00112:         struct ipt_standard_target *stdtarget = (struct ipt_standard_target *)t;
```

图 **9**

这两个结构体的关系如下图：

图 **10**

2.2、以 filter 表为例

2.2.1 match 规则的解析

"filter"表从**"FILTER_VALID_HOOKS"**这些 **hook** 点（这里即 **INPUT**、**OUTPUT**、**FORWARD**）介入 **Netfilter** 框架，并且 **filter** 表初始化时有 **3** 条规则链，每个 **HOOK** 点**(**对应用户空间的“规则链”**)**初始化成一条链，最后以一条“错误的规则”表示结束，于是，**filter** 表占**(sizeof(struct ipt_standard) * 3 + sizeof(struct ipt_error))**字节的存储空间。

```
00492:   memcpy(table->cpnftable_name, info.name, sizeof(table->cpnftable_name));
00493:   table->cpnftable_family = AF_INET;
00494:   table->cpnftable_valid_hooks = htonl(info.valid_hooks);
00495:
00496:   for(hook = 0; hook < CM_NF_IP_NUMHOOKS && hook < NF_IP_NUMHOOKS; hook++) {
00497:   /* XXX: yzhou: ignore invalid hook entries */
00498:   if (!(info.valid_hooks&(1<<hook)))
00499:     continue ;
00500:       if (ip4tc_offset2index(&table->cpnftable_hook_entry[hook], entries,
00501:                     (struct ipt_entry *)((void *)entries->entrytable +
00502:                             info.hook_entry[hook])) < 0)
00503:           goto out;
00504:       if (ip4tc_offset2index(&table->cpnftable_underflow[hook], entries,
00505:                     (struct ipt_entry *)((void *)entries->entrytable +
00506:                             info.underflow[hook])) < 0)
00507:           goto out;
00508:   }
```

图 **11**

如上图，以 **hook** 为下标进行遍历循环，解析出有效的规则。

```
static int ip4tc_offset2index(u_int32_t *index,
                              struct ipt_get_entries *entries,
                              struct ipt_entry *seek)
```

其中第二个参数是我们获取出的，某个 **hook** 中的所有有效 **entry** 的列表，该函数的作用是，将有效的 **entry** 位置，保存在 **tables** 结构体中。

```
00503:  if (IPT_ENTRY_ITERATE(entries->entrytable, entries->size, ip4tc_copy_entry,
00504:                        table, &info, entries, &i) < 0) {
00505:      printf("%s: failed to parse entry for table %s\n",
00506:             __FUNCTION__, tablename);
00507:      goto out;
00508:  }
```

图 **12**

这里就是拷贝具体的 **entry**（含 **match** 和 **target**）了。

```
if (IPT_MATCH_ITERATE(e, ip4tc_copy_match, rule) < 0) {
    printf("%s: Fail to parse entry\n", __FUNCTION__);
    return -EINVAL;
}
```

图 **13**

如图 **13**，代码中用到了较多的 **IPT_XXXXX_ITERATE** 宏，这里以一个解析 **match** 的宏为例讲解一下。

宏开头的 **IPT**，是指要解析 **ipt_entry** 结构体，**MATCH** 是指目前要解析 **match** 规则，第二个参数是相应的回调函数，之后的参数可能有一个也可能有多个，是回调函数的入参。

展开看定义如下：

1、对于传入的 **e**（**ipt_entry**）和其类型 **type**（**ipt_entry**）；

2、先找到第一条 **match** 的地址（注意第一条 **match** 规则，在位置上，是存放在 **ipt_entry** 后面的，所以这里 **i** 的偏移就是 **sizeof(type)** 了）

3、使用 **fn** 函数进行处理，这里就是 **ip4tc_copy_match**。

4、**i** 再加上当前 **match** 规则的大小偏移（**match** 对应的结构体是 **ipt_entry_match**，**u.match_size** 指明了大小），转移到下一条 **match**

5、**i** 的位移要小与 **target_offset** 的值，因为第一条 **target**，也是在最后一条 **match** 规则后面

```
/* fn returns 0 to continue iteration */
#define XT_MATCH_ITERATE(type, e, fn, args...)
({                                      \
    unsigned int __i;                   \
    int __ret = 0;                      \
    struct xt_entry_match *__m;         \
                                        \
    for (__i = sizeof(type);            \
         __i < (e)->target_offset;      \
         __i += __m->u.match_size) {    \
        __m = (void *)e + __i;          \
                                        \
        __ret = fn(__m , ## args);      \
        if (__ret != 0)                 \
            break;                      \
    }                                   \
    __ret;                              \
})
```

图 **14**

虽然最外层没有循环，但是实际内层是不断在循环且调用 **fn**，所以在一次宏调用的过程中，实际数据解析的过程已经全部完成了。调用循环是：

1、如图 **12** 代码，对于取出的 **entry** 指针，先遍历每一个 **ipt_entry** 结构；

2、如图 **13** 代码，遍历其中的每一个 **match** 规则

2.2.2 hook_entry 和 underflow 的意义

2.2.1 中可以看到通过 **ip4tc_offset2index** 函数进行了 **hook_entry** 和 **underflow** 的转换，这两个字段在后面也经常会看得到，那这两个字段到底是什么意思呢？

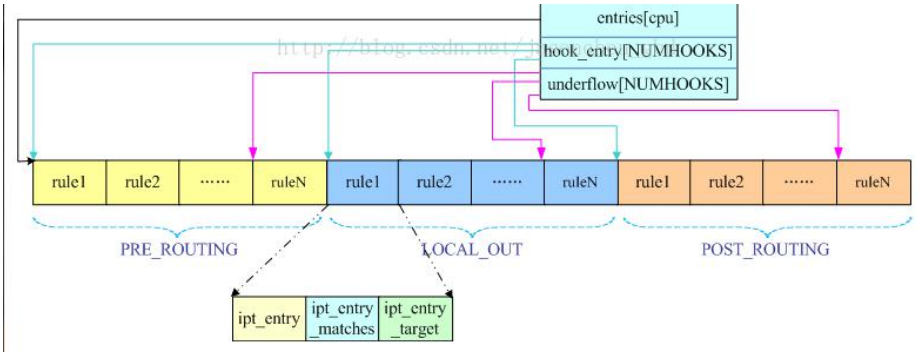

图 15

在 netfilter 里，所有的匹配规则都是注册在相应表的 xt_tables 结构体中 private 指针后面的。对于某张表，其后面跟的规则就如上图所示，先是若干的 PRE_ROUTING 表规则，然后是若干的 LOCAL_OUT 表规则。那么 hook 点为 0、1、2、3、4，那么相应的 hook_entry 和 underflow 就如上图所示：0 表示第一个链即 pre_routing，那么 hook_entry[0]指向第一条链的第一个规则 rule1，相应的 underflow[0]指向 ruleN 即此链的最后一条规则。即，这两个值是一个偏移量，总是指向对应规则的第一条，和最后一条，这样在解析时可以清晰的将不同链的规则分界。但是注意，这时候对于每个链，都有一个默认的 policy(默认处理规则,当所有的 ipt_entry 都走完之后,或者当前链没有配置规则时的默认动作)，这也算是一个 ipt_entry，并在 hook_entry 和 underflow 的计数上占据一个位置。如上图的话，ruleN 都是不同链上默认的处理规则。同时，只在所有内建链的结尾(包括所有五条链)，和自建链的结尾，才有 TARGET_ERROR。自建链结尾之前还有一个 REPEAT。

举个例子，假设对于 filter 表来说，有效的 hook 点为 INPUT/FORWARD/OUTPUT，那么在没有任何 iptables 规则时，相应取值应该是：

	pre	in	for	out	post
hookentry	X	0	1	2	X
underflow	X	0	1	2	X

现在在 filter 表的 FORWARD 链上配置两条规则，相应取值就应该是：

	pre	in	for	out	post
hookentry	X	0	1	4	X
underflow	X	0	3	4	X

再在 OUTPUT 上配置两条规则，相应取值变成：

	pre	in	for	out	post
hookentry	X	0	1	4	X

```
underflow     X      0    3     6     X
```
再在 INPUT 上配置一条规则，相应取值又变成：
```
            pre    in   for   out  post
hookentry     X      0    2     5     X
underflow     X      1    4     7     X
```
回到之前的代码，第一次 getsockopt 时，可以看到获取的结构体为：

getsockopt(sockfd, IPPROTO_IP, IPT_SO_GET_INFO, &info, &s)

这是第一次取数据使用的 **info** 结构体。

```
struct ipt_getinfo
{
    /* Which table: caller fills this in. */
    char name[IPT_TABLE_MAXNAMELEN];

    /* Kernel fills these in. */
    /* Which hook entry points are valid: bitmask */
    unsigned int valid_hooks;

    /* Hook entry points: one per netfilter hook. */
    unsigned int hook_entry[NF_INET_NUMHOOKS];

    /* Underflow points. */
    unsigned int underflow[NF_INET_NUMHOOKS];

    /* Number of entries */
    unsigned int num_entries;

    /* Size of entries. */
    unsigned int size;
};
```

可以看到，这次的宏是 **IPT_SO_GET_INFO**， 用来获取当前表（表名 **name** 是 **key** 值，已经在 **info** 中赋值了），取到的内容主要是当前表中规则的大小 **size**，对于该表，有效链的标志 **valid_hooks**， 和该表下规则在相应链中的起始位置 **hook_entry** 及结束位置**(**最后一条规则**)underflow.**

再看第二次匹配：

getsockopt(sockfd, IPPROTO_IP, IPT_SO_GET_ENTRIES, entries, &size)

注意变成了 **IPT_SO_GET_ENTRIES**，**entries** 的数据结构为

```
struct ipt_get_entries
{
    /* Which table: user fills this in. */
    char name[IPT_TABLE_MAXNAMELEN];

    /* User fills this in: total entry size. */
```

```
    unsigned int size;

    /* The entries. */
    struct ipt_entry entrytable[0];
};
```

最后是一个 **ipt_entry** 类型的柔性数组，后面跟具体的 **ipt_entry** 类型数据，此类型描述了 **match** 和 **target** 的规则。但是注意这个地方并没有任何的偏移信息，即该表中所有链的数据都一股脑的接在了 **entrytable[0]**后面。我们的数据后面是要放进 **cp_nftable** 中的，因此，需要使用 **ip4tc_offset2index** 函数进行转换。

```
if (ip4tc_offset2index(&table->cpnftable_hook_entry[hook], entries,
                (struct ipt_entry *)((void *)entries->entrytable +
                        info.hook_entry[hook])) < 0)
    goto out;
```

图 **16**

这里，第二个参数是第二次 **getsockopt** 获得的 **entries**，第三个参数是当前表中各个链开始规则的偏移，通过转换，将偏移放到第一个参数中。

2.2.3 实际 match 内容的存储

从规则名，我们可以看出当前的 **match** 是哪个规则，**name** 的作用，**2.1.2** 中解释过。

```
/* UDP parameters */
if (!strcmp(m->u.user.name, "udp")) {
    struct ipt_udp *udpinfo = (struct ipt_udp *)m->data;

    rule->l3.type = CM_NF_L3_TYPE_UDP;
    rule->l3.data.udp.spts[0]  = htons(udpinfo->spts[0]);
    rule->l3.data.udp.spts[1]  = htons(udpinfo->spts[1]);
    rule->l3.data.udp.dpts[0]  = htons(udpinfo->dpts[0]);
    rule->l3.data.udp.dpts[1]  = htons(udpinfo->dpts[1]);
    rule->l3.data.udp.invflags = udpinfo->invflags;
    return 0;
}
```

图 **17**

确定了是 **UDP match** 后，将相应的数据写入到 **rule** 即可。本条 **ipt_entry** 中的 **ipt_entry_match** 中的一条 **match** 规则解析完毕，返回。未来我们增加**-m** 匹配即扩展此处的代码即可识别更多的类型。

2.2.4 target 规则的存储

```
/* REJECT target */
if (!strcmp(t->u.user.name, "REJECT")) {
    rule->target.type = CM_NF_TARGET_TYPE_REJECT;
    return 0;
}
```

图 **18**

如上图，实际和 **match** 差别不大，这里不赘述了。

2.2.5 向 FPM（FASTPATH）发送数据

目前已有的代码是 **CM** 获取了数据后发给 **FPM**，**FPM** 再发给 **fastpath** 代码。这意味着当前 **CM** 至 **FPM** 有一套接口和相应规则（**CP_NFRULE**），**FPM** 到 **fastpath** 又有一套相应的规则（**FP_NFRULE**）。以 **UDP** 协议的 **match** 为例：

CM 填写 **CP_NFRULE** 的代码，和相应数据结构为：

```
00222:      /* UDP parameters */
00223:      if (!strcmp(m->u.user.name, "udp")) {
00224:          struct ipt_udp *udpinfo = (struct ipt_udp *)m->data;
00225:
00226:          rule->l3.type = CM_NF_L3_TYPE_UDP;
00227:          rule->l3.data.udp.spts[0]  = htons(udpinfo->spts[0]);
00228:          rule->l3.data.udp.spts[1]  = htons(udpinfo->spts[1]);
00229:          rule->l3.data.udp.dpts[0]  = htons(udpinfo->dpts[0]);
00230:          rule->l3.data.udp.dpts[1]  = htons(udpinfo->dpts[1]);
00231:          rule->l3.data.udp.invflags = udpinfo->invflags;
00232:          return 0;
00233:      }
```

```
union {
    struct {
        u_int16_t spts[2];              /* Source port range. */
        u_int16_t dpts[2];              /* Destination port range. */
        u_int8_t invflags;              /* Inverse flags */
    } udp;
```

图 **19 20**

而 **nf_ip_match()**函数实际解析 **udp match** 的部分和相应数据结构为：

```
00398:     case FP_NF_L3_TYPE_UDP:
00399:         /* Must not be a fragment. */
00400:         if (ntohs(ip->ip_off) & FP_IP_OFFMASK)
00401:             return NF_IP_MATCH_NO;
00402:
00403:         if (!nf_check_len(m, ip->ip_hl * 4, sizeof(struct fp_udphdr)))
00404:             return NF_IP_MATCH_ERROR;
00405:         uh = m_off(m, ip->ip_hl * 4, struct fp_udphdr *);
00406:
00407:         if (!(nf_port_match(r->l3.data.udp.spts[0], r->l3.data.udp.spts[1],
00408:                     ntohs(uh->source),
00409:                     !!(r->l3.data.udp.invflags & FP_NF_IPT_UDP_INV_SRCPT))
00410:                 && nf_port_match(r->l3.data.udp.dpts[0], r->l3.data.udp.dpts[1],
00411:                     ntohs(uh->dest),
00412:                     !!(r->l3.data.udp.invflags & FP_NF_IPT_UDP_INV_DSTPT))))
00413:             return NF_IP_MATCH_NO;
00414:         break;
```

```
        union {
            struct {
                uint16_t spts[2];           /* Source port range. */
                uint16_t dpts[2];           /* Destination port range. */
#define FP_NF_IPT_UDP_INV_SRCPT     0x01    /* Invert the sense of source ports.
#define FP_NF_IPT_UDP_INV_DSTPT     0x02    /* Invert the sense of dest ports. */
#define FP_NF_IPT_UDP_INV_MASK      0x03    /* All possible flags. */
                uint8_t invflags;
            } udp;
```

图 **21 22**

其他类似于 **ICMP**，**TCP**，**LIMIT** 等基本都是一致的。现在不需要 **FPM** 了，那么按照项目需求，比对 **nf_ip_match()**代码，直接在 **CM** 获取数据后填成 **FP_NFRULE** 格式并发送给 **fastpath** 即可。

目前问题有两点：

1、当前 **fastpath** 已有的匹配规则和对应 **iptables** 结构体如下:

-m tcp **FP_NF_L3_TYPE_TCP** **ipt_tcp**

-m udp **FP_NF_L3_TYPE_UDP** **ipt_udp**

-m icmp **FP_NF_L3_TYPE_ICMP** **ipt_icmp**

-m multiport **xt_multiport_v1**

-m iprange **xt_iprange_mtinfo**

-m mark **markflag** **// xt_mark_target_info_v1**

-m mac **MCORE_NF_MAC** **xt_mac_info**

-m limit **FP_NF_I2OPT_RATELIMIT** **xt_rateinfo**

没有的:

-m time **xt_time_info**

-m physdev **xt_physdev_info**

-m state **xt_state_info**

-m connlimit **xt_connlimit_info**

暂缓:

-m hashlimit **xt_hashlimit_mtinfo1**

-m recent **xt_recent_mtinfo_v1**

-m socket **xt_socket_mtinfo1**

-m ndpi 未知

三、iptables 代码解释

iptables 主要功能其实比较清晰，即配置和回显。简单点讲，由以下几部分构成：
1、加载外部的规则库；
2、解释输入的命令行参数，并存放进相应的结构里；
3、输入配置的合法性检查，这一部分对读者并不是很重要；
4、使用 **getsockopt** 取出 **netfilter** 现在已有的规则，并进行整理和检查；
5、将新增的配置插入规则，并使用 **setsockopt** 配置进 **netfilter**。

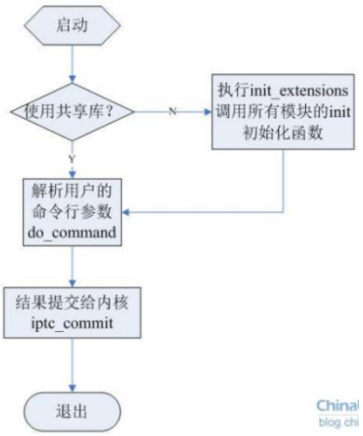

图 **23**

整个过程如上图所示，下面通过这几步进行讲解。

3.1 外部库的加载

前文说过，为了模式上的精简和可扩展性，**iptables** 自己本身并没有整合所有的 **match** 和 **target** 模块，而是靠在启动的时候，动态加载其他模块来实现功能的。每个模块自己实现了自己的一些操作函数，如 **libxt_time.c** 中加载的 **time** 模块，正是这样，让 **iptables** 变得比较灵活，可以自行添加自己需要的功能，也让编译出来的 **iptables** 占用空间较小，如下图例：

```
static struct xtables_match time_match = {
    .name          = "time",
    .family        = NFPROTO_UNSPEC,
    .version       = XTABLES_VERSION,
    .size          = XT_ALIGN(sizeof(struct xt_time_info)),
    .userspacesize = XT_ALIGN(sizeof(struct xt_time_info)),
    .help          = time_help,
    .init          = time_init,
    .print         = time_print,
    .save          = time_save,
    .x6_parse      = time_parse,
    .x6_fcheck     = time_check,
    .x6_options    = time_opts,
};
```

图 **24**

在 **do_command4** 之前，我们先调用了 **init_extensions()**函数来加载 **match** 模块，并初始化了 **xtables_pending_matches** 全局链表。在这里有已经注册**(**或者说已经加载**)**的所有 **match** 结构，以上面的 **time** 模块为例，这里会调用其**_init** 函数进行注册

```
void _init(void)
{
    xtables_register_match(&time_match);
}
```

图 **25**

注意以下的定义，在 **Xtables.c** 的 **192** 行。注册在前两个链表上的，都是未完全注册的，只进行了部分加载的模块。中间两个链表，都是完全注册的模块，后面的两个函数则用来从未完全注册链上取下相应模块，插入到完全注册链上。

```
/* Keep track of matches/targets pending full registration: linked lists. */
struct xtables_match *xtables_pending_matches;
struct xtables_target *xtables_pending_targets;

/* Keep track of fully registered external matches/targets: linked lists. */
struct xtables_match *xtables_matches;
struct xtables_target *xtables_targets;

/* Fully register a match/target which was previously partially registered. */
static void xtables_fully_register_pending_match(struct xtables_match *me);
static void xtables_fully_register_pending_target(struct xtables_target *me);
```

图 **26**

iptables 对于加载模块的理念是使用才加载，调用才分配，以此做到了代码的短小精悍。函数 **should_load_proto** 之前的注释解释到，**iptables** 支持隐式加载协议扩展，但是如果遇到一个还未能被解析的选项，那说明这个协议并未被完全加载。例如，**-p tcp** 很好解析，但是如果想解析**--dport** 时，就需要加载整个 **tcp** 模块了。后面在执行 **xtables_find_match()**函数时，会调用图 **26** 中的函数，触发“延迟的初始化”，实际就是 **partially register** 到 **fully register** 的过程。

```
/* Trigger delayed initialization */
for (dptr = &xtables_pending_matches; *dptr; ) {
    if (extension_cmp(name, (*dptr)->name, (*dptr)->family)) {
        ptr = *dptr;
        *dptr = (*dptr)->next;
        ptr->next = NULL;
        xtables_fully_register_pending_match(ptr);
    } else {
        dptr = &((*dptr)->next);
    }
}
```

图 **27**

具体情况可见 **3.2.2** 节，加载**--syn** 时的解释。

3.2 命令行参数的解释

3.2.1 getopt_long 函数的解释

iptables 配置一般都很长，需要跟很多五花八门的参数，为了解决这么复杂的参数问题，使用了 **getopt_long** 函数来解析这些命令。

getopt_long 的原型为：

getopt_long(int argc, char * const argv[], const char * opstring,
const struct option *longopts, int *longindex)
extern int optind, opterr, optopt;

这里 **argc** 和 **argv** 由 **main** 函数参数传递。

一般使用 **getopt_long** 的方法都是

```
while( (result = getopt_long(xxxx)) !=-1)
{
    switch (result):
    case A: xxxx
            break;
    case B:xxxx
            break;
    .....
    default:break;
}
```

这样的循环判断句式，每次成功执行，将返回一个正在解析的选项，设置一些全局变量，当返回**-1** 的时候，整个命令行就解析完成了。

其中第四个参数 **longopts** 指向如下的结构体：

```
struct option{
        const char *name;
        int has_arg;
        int *flag;
        int val;
};
```

name 是选项的名字，如**-help** 等

has_arg 的值为 **0**，**1**，**2** 分别代表无参数，有参数，参数可选，对应参数后的冒号数量

flag 如果为 **null**， 则函数返回 **val** 值，否则将返回值 **val** 写入 **flag** 指针指向的变量，并返回 **0**。

如下是一个选项初始化的例子：

```
char *const short_options = "ho:v"
const struct option long_options = {
   { "help",     0,   NULL,   'h' },
   { "output",   1,   NULL,   'o' },
   { "verbose",  0,   NULL,   'v' },
   { "input",    1,   NULL,   'i' }
};
```

不难理解，当输入**-help** 时，此次的函数调用返回为**'h',**下面的 **case 'h':**就可以做相应的处理了,而参数为**-o** 或者**-i** 时，则后面必须跟上参数才行。

然后解释一下设置的全局变量：

char *optarg 当前选项后面跟的参数字串（如果有的话，如**--sport 2000,**这里指向 **2000**）。

int optind **argv** 的当前索引值。即每次操作完成后，剩余的参数在 **argv[optind]**至 **argv[argc-1]**区间中。

举一个例子，当前执行 **./test.out -h -o eth1 -v -i eth2,** 那么 **optind** 分别就是 **1**，**2**，**4**，**5**，**7.** 为什么呢？**optind** 开始为 **1**，指向**-h**，但是**-h** 不需要参数，所以下次调用 **getopt_long** 后自动加 **1**，变为 **2** 即**-o**，但是**-o** 有参数，后面的肯定就是其后接参数值，于是在本次直接跳到了下一个选项 **4**，就指向了**-v.** 同理，解析完了**-v** 后，下一次循环到 **5** 时直接跳到了 **7** 即 **eth2** 之后一位。

这里顺便说明一下两种选项的区别：
长选项**(long options)**　包含两个连字号，如**--help --sport** 等
短选项**(short options)**　包含一个连字号，如**-s -d -n** 等

3.2.2 参数解释的过程

解析参数和执行是使用了 **do_command4** 函数来执行的**(ipv6** 则是 **do_command6,**目前都以 **ipv4** 讲解，后面不再解释**)**，其原形为 **int do_command(int argc, char *argv[], char **table, iptc_handle_t *handle)**；**tables** 即操作的表名，默认为 **filter** 表。**iptc_handlt_t** 这个结构保存了从内核返回的，由 **tables** 名指定的表的所有信息，是后续代码处理的主线。在 **2.1.2** 中讲到了 **match/target** 在用户空间和内核空间不同的结构体定义，那么实际上，**xtables_match** 结构就是 **iptables** 在加载这个 **match** 模块时所用到的结构体，如 **time** 模块等，自己如果想添加规则，就实例化这样一个对象并构造相应方法即可，见图 **24**。而真正用在我们所配置的 **iptables** 规则里的匹配条件是由下图这样的结构体表示的：

```
struct xtables_rule_match {
    struct xtables_rule_match *next;
    struct xtables_match *match;
    /* Multiple matches of the same type: the ones before
       the current one are completed from parsing point of view */
    bool completed;
};
```

图 **28**

可以看到在这里，所有的 **xtables_match** 被组织成了一个链表。这样的话，当一条规则里有多个 **match** 条件时，我们只需要将规则用到的 **match** 条件，用指针指向 **iptables** 加载的相应模块，后面就可以直接调用实现在相应模块的函数中了。再次说明，**iptables** 自己本身是不实现 **match** 或者 **target** 功能的。下面以一条配置为例，解释命令行的流程

iptables –A INPUT –i eth0 –t tcp --syn –s 172.16.22.75 -d 192.16.10.1 –j ACCEPT

3.2.2.1 -A INPUT

对于这种控制命令，都是使用 **add_command()**函数处理的。函数原形为:
static void add_command(unsigned int *cmd, const int newcmd, const int othercmds, int invert)该函数主要是将控输入的控制参数解析出来，根据不同的 **case**，填入不同的第二参数 **newcmd**，并将其写入到第一个参数 **cmd** 中。**cmd** 的每一位都设置为不同的动作，如下图所示。

```
#define CMD_NONE        0x0000U
#define CMD_INSERT      0x0001U
#define CMD_DELETE      0x0002U
#define CMD_DELETE_NUM      0x0004U
#define CMD_REPLACE     0x0008U
#define CMD_APPEND      0x0010U
#define CMD_LIST        0x0020U
#define CMD_FLUSH       0x0040U
#define CMD_ZERO        0x0080U
#define CMD_NEW_CHAIN       0x0100U
#define CMD_DELETE_CHAIN    0x0200U
#define CMD_SET_POLICY      0x0400U
#define CMD_RENAME_CHAIN    0x0800U
#define CMD_LIST_RULES      0x1000U
#define CMD_ZERO_NUM        0x2000U
#define CMD_CHECK       0x4000U
#define NUMBER_OF_CMD   16
```

图 **29**

只有在敲入 **iptables -Z chainname** 的时候才需要额外参数 **othercommand**，即输出清空后的链实际情况。**,**此时会附加一个 **CMD_LIST** 属性。

当解析到**-A** 的时候，**-A** 后面必然有其他参数，即**-A PREROUTING**，此时的 **optarg** 即指向这个 **PREROUTING**，印证了我们在 **3.1.1** 中对 **optarg** 的解释。

```
case 'A':
    add_command(&command, CMD_APPEND, CMD_NONE,
            cs.invert);
    chain = optarg;
    break;
```

图 **30**

而实际的 **add_command** 处理很简单，主要就是一句***cmd |= newcmd;** 注意，对于控制选项，是不允许取反的，即 **cs.invert** 必须为 **0.**

3.2.2.2 -i eth0

iptables –A INPUT –i eth0 –p tcp --syn –s 172.16.22.75 -d 192.16.10.1 –j ACCEPT

这时我们解析到了**-i eth0**，此时 **optarg** 是 **eth0**，而 **optind** 就是 **5(**指向 **eth0** 后一位**)**

其中 **set_option** 和上面的 **add_command** 做法类似， **cs_option** 也是一个位变量**.**参数 **OPT_VIANAMEIN** 是指当前处理入接口名称。

```
case 1: /* non option */
    if (optarg[0] == '!' && optarg[1] == '\0') {
        if (cs.invert)
            xtables_error(PARAMETER_PROBLEM,
                  "multiple consecutive ! not"
                  " allowed");
        cs.invert = TRUE;
        optarg[0] = '\0';
        continue;
    }
    fprintf(stderr, "Bad argument `%s'\n", optarg);
    exit_tryhelp(2);
```

图 **31**

注意我们使用！取反时后面是要跟空格的，此时根据 **longopt** 的跳转，在这里对 **cs.invert** 设置为 **TRUE**。

```
case 'i':
    if (*optarg == '\0')
        xtables_error(PARAMETER_PROBLEM,
            "Empty interface is likely to be "
            "undesired");
    set_option(&cs.options, OPT_VIANAMEIN, &cs.fw.ip.invflags,
           cs.invert);
    xtables_parse_interface(optarg,
            cs.fw.ip.iniface,
            cs.fw.ip.iniface_mask);
    break;
```

图 **32**

根据 **cs.invert** 来给第三个参数 **cs.fw.ip.invflags** 取相应值，其可能取值如下图

```
static const int inverse_for_options[NUMBER_OF_OPT] =
{
/* -n */ 0,
/* -s */ IPT_INV_SRCIP,
/* -d */ IPT_INV_DSTIP,
/* -p */ XT_INV_PROTO,
/* -j */ 0,
/* -v */ 0,
/* -x */ 0,
/* -i */ IPT_INV_VIA_IN,
/* -o */ IPT_INV_VIA_OUT,
/*--line*/ 0,
/* -c */ 0,
/* -f */ IPT_INV_FRAG,
};
```

图 **33**

接下来就是将接口名称和 **mask** 位赋给 **fw.ip.iniface.** 由于**-i** 选项有可能取非，即**-i ! eth0** 的情况，所以此处最好使用 **arv[optind-1]**指向接口名。

3.2.2.3 -p tcp

此时 **set_option** 的第三个参数为 **OPT_PROTOCOL**，表明配置协议，其宏定义如下

```
enum {
    OPT_NONE        = 0,
    OPT_NUMERIC     = 1 << 0,
    OPT_SOURCE      = 1 << 1,
    OPT_DESTINATION = 1 << 2,
    OPT_PROTOCOL    = 1 << 3,
    OPT_JUMP        = 1 << 4,
    OPT_VERBOSE     = 1 << 5,
    OPT_EXPANDED    = 1 << 6,
    OPT_VIANAMEIN   = 1 << 7,
    OPT_VIANAMEOUT  = 1 << 8,
    OPT_LINENUMBERS = 1 << 9,
    OPT_COUNTERS    = 1 << 10,
};
```

图 **34**

主要执行的 **xtables_parse_protocol** 函数并没有什么太多的行为，这里和 **3.1** 中对于**-p tcp** 的解释是对应的，单纯的解析**-p tcp** 不需要加载很多选项，这里就能完成了。只有在加载一些扩展的协议选项时，才需要完全加载模块本身，如下一个参数。

3.2.2.4 --syn

iptables –A INPUT –i eth0 –p tcp --syn –s 172.16.22.75 -d 192.16.10.1 –j ACCEPT

不难看出 **getopt_long** 的短选项中并没有 **case syn: (**当然也不可能用字符串作为 **case** 入口**)**，此时会落入 **switch** 语句的 **default** 选项中，执行 **command_default** 函数。

以下的过程就比较复杂了：解析到这个时候，还没有任何**-m/-j** 选项被命中，所以所有关于这两者的指针都还是空的，即目前还没有“**fully register**”的 **match** 或者 **target**，因此会走到下面的 **load_proto** 函数。该函数先将传进来的数字型的协议名，改为对应的字符串协议名，并使用 **xtables_find_match** 函数进行查找。此时就是在 **3.1** 中说明的 **pending** 链表中，查找名字和协议名相同的结构体**(**查找**"**非完全注册**"**的目标**)**。实际上这里并非单纯的查找，而是用来做一个**"partially register"** 到**"fully register"**的置换。

```
/* Trigger delayed initialization */
for (dptr = &xtables_pending_matches; *dptr; ) {
    if (extension_cmp(name, (*dptr)->name, (*dptr)->family)) {
        ptr = *dptr;
        *dptr = (*dptr)->next;
        ptr->next = NULL;
        xtables_fully_register_pending_match(ptr);
    } else {
        dptr = &((*dptr)->next);
    }
}

for (ptr = xtables_matches; ptr; ptr = ptr->next) {
    if (extension_cmp(name, ptr->name, ptr->family)) {
        struct xtables_match *clone;

        /* First match of this type: */
        if (ptr->m == NULL)
            break;
```

图 **35**

找到就删除，然后再将相应的名字传入 **xtables_fully_register_pending_match** 函数，此处会再

次调用 **xtables_find_match** 函数，这时不会在 **pending** 链上再找到了，于是，现在将这个 **match** 结构体插入到全局链表 **xtables_matches**（完全注册链）的结尾。执行到这里的时候，**xtables_fully_register_pending_match** 函数就完成了。

```
/* Append to list. */
for (i = &xtables_matches; *i; i = &(*i)->next);
me->next = NULL;
*i = me;
```

图 **36**

可以看到，这个过程实际也算是注册的第二部分过程，即 **iptables** 在启动的时候，维护了两个链表，一边使用，一边初始化，真正使用到该 **match** 的时候，才正式加载其 **match** 方法。继续向下，虽然这时候 **xtables_matches** 里已经有了相应的模块名，也可以找到，但是由于是第一次使用，**ptr->m** 即内部的 **xt_entry_match** 还是为空，所以还是跳出。

```
for (ptr = xtables_matches; ptr; ptr = ptr->next) {
    if (extension_cmp(name, ptr->name, ptr->family)) {
        struct xtables_match *clone;

        /* First match of this type: */
        if (ptr->m == NULL)
            break;
```

图 **37**

往下执行，这时模块还未试用过，因此 **ptr->loaded** 标志为空，将 **ptr** 置空.

```
if (ptr && !ptr->loaded) {
    if (tryload != XTF_DONT_LOAD)
        ptr->loaded = 1;
    else
        ptr = NULL;
}
```

图 **38**

这时，**should_load_proto** 返回了，在 **load_proto** 返回之前再次执行了 **find_proto** 函数：

```
struct xtables_match *load_proto(struct iptables_command_state *cs)
{
    if (!should_load_proto(cs))
        return NULL;
    return find_proto(cs->protocol, XTF_TRY_LOAD,
              cs->options & OPT_NUMERIC, &cs->matches);
}
```

图 **39**

但是这次已经和上次完全不一样了，这次不但是 **XTF_TRY_LOAD** 方式，且将 **cs->matches** 传入了。因此这次将 **ptr->load** 置为 **1**， 再继续执行，此时相应的初始化和注册都已完成，将刚才加载的 **tcp** 模块的 **match** 结构加载进 **cs(matches)**中。

```
if (ptr && matches) {
    struct xtables_rule_match **i;
    struct xtables_rule_match *newentry;

    newentry = xtables_malloc(sizeof(struct xtables_rule_match));

    for (i = matches; *i; i = &(*i)->next) {
        if (extension_cmp(name, (*i)->match->name,
                  (*i)->match->family))
            (*i)->completed = true;
    }
    newentry->match = ptr;
    newentry->completed = false;
    newentry->next = NULL;
    *i = newentry;
}

return ptr;
```

图 **40**

这样，整个 **find_proto** 就结束了,此处返回的 **ptr** 即相应的 **match**。

```
m = load_proto(cs);
if (m != NULL) {
    size_t size;

    cs->proto_used = 1;

    size = XT_ALIGN(sizeof(struct xt_entry_match)) + m->size;

    m->m = xtables_calloc(1, size);
    m->m->u.match_size = size;
    strcpy(m->m->u.user.name, m->name);
    m->m->u.user.revision = m->revision;
    xs_init_match(m);
```

图 **41**

返回到 **command_default** 继续执行，将 **cs->proto_used** 置为 **1**，表示模块已经被加载，也即将被使用了。填充 **xt_entry_match** 变量，并使用 **xs_init_match** 函数初始化相应 **m** 变量的初始值，注意下面的 **optind--**,需要再次返回到**--syn** 的参数中，这样，**command_default** 函数就返回了。

继续进入 **while** 循环，取出**--syn**，再次进入 **command_default** 函数，此时 **cs-matches** 不再为空了，调用 **xtables_option_mpcall**，此时便是真正调用相应模块加载时的 **parse** 函数了：

```
void xtables_option_mpcall(unsigned int c, char **argv, bool invert,
                struct xtables_match *m, void *fw)
{
    struct xt_option_call cb;

    if (m->x6_parse == NULL) {
        if (m->parse != NULL)
            m->parse(c - m->option_offset, argv, invert,
                  &m->mflags, fw, &m->m);
        return;
    }
```

图 **42**

函数原型即在 **libxt_tcp.c** 中的 **tcp_parse** 函数。返回解析结果，这样，整个**--syn** 就解析完成了。

回头可以看到，**-p tcp --syn** 这是两个匹配条件。第一个仅仅是协议匹配，是在基本匹配内的，不需要加载任何的外置模块。而**--syn** 实际是隶属于 **tcp** 协议的一个匹配，因此在解析的过程中先加载了 **tcp** 协议模块。在解析中需要额外加载第三方**(**可以这么说，我们自己也可以实现自己的第三方 **match)**模块的资源进行匹配，那么这就是一个扩展匹配。

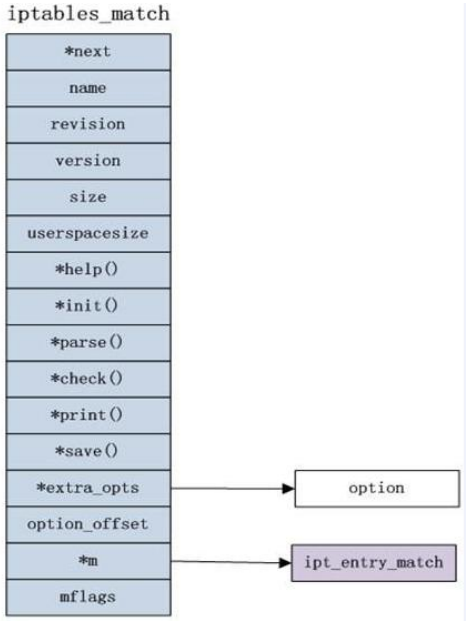

图 **43**

如图 **43**，是一个扩展匹配模块的结构。

```
static struct xtables_match tcp_match = {
    .family      = NFPROTO_UNSPEC,
    .name        = "tcp",
    .version     = XTABLES_VERSION,
    .size        = XT_ALIGN(sizeof(struct xt_tcp)),
    .userspacesize  = XT_ALIGN(sizeof(struct xt_tcp)),
    .help        = tcp_help,
    .init        = tcp_init,
    .parse       = tcp_parse,
    .print       = tcp_print,
    .save        = tcp_save,
    .extra_opts  = tcp_opts,
};
```

图 **44**

上图是在 **libxt_tcp.c** 中对 **tcp** 协议扩展模块初始化注册使用的结构体。

3.2.2.5 -s 172.16.22.75 -d 192.16.10.1

iptables –A INPUT –i eth0 –p tcp --syn –s 172.16.22.75 -d 192.16.10.1 –j ACCEPT

这里相对就比较简单了，暂存给 **shostnetworkmask**

```
case 's':
    set_option(&cs.options, OPT_SOURCE, &cs.fw.ip.invflags,
           cs.invert);
    shostnetworkmask = optarg;
    break;
```

图 **45**

注意这里的 **optarg**，赋值内容是一个字符串常量，这里将其赋值给 **shostnetworkmask**，并不影响在下一次的 **getopt_long** 里面再赋值给 **dhostnetworkmask**。即这时候的=号，代表的是将一个字符串常量赋值给一个指针，使得这个指针也指向了一个字符串常量，而不是简单的将指针指向了字符串常量所在的地址。因此在下一次的 **getopt_long** 时可以直接再暂存目的地址的字符串。

3.2.2.6 -j ACCEPT

iptables –A INPUT –i eth0 –p tcp --syn –s 172.16.22.75 -d 192.16.10.1 -j ACCEPT

程序走入分支**'j'**,首先还是配置相应的选项并进行合法性检查，

```
tic void command_jump(struct iptables_command_state *cs)

 size_t size;

 set_option(&cs->options, OPT_JUMP, &cs->fw.ip.invflags, cs->invert);
 cs->jumpto = parse_target(optarg);
 /* TRY_LOAD (may be chain name) */
 cs->target = xtables_find_target(cs->jumpto, XTF_TRY_LOAD);
```

图 **46**

在 **xtables_find_target** 里，做的事情其实和分析**--syn** 差不多，一样是从 **pending** 链上查找并删除，再挂到 **fully register** 的链上，最后成功加载相应的 **target**，这里就不赘述了。这样，一条命令就解析完成了。

到这个时候，控制参数，如**-A**，存放在 **command** 中；规则参数，如**-p**，存放在 **option** 中；源和目的地址分别保存在相应的指针里；也完成了对 **fw.ip** 即 **ipt_ip** 的初始化。再往下

就是一些合法性的检查，这些代码不详细介绍了。

3.3 从内核取出已有规则

到了这里，我们的命令行已经解释完成，需要开始进行实际操作了。在这里，**iptables** 并不是下发单独的一条 **ipt_entry** 给内核，而是采用了按表分类，将所有内容都同步上来，重新解析**(**用户态对于数据结构的配置和内核态是不一样的**)**，插入我们新生成的内容，再将其统一下发下去的做法。

这里我有一个疑问，为什么不直接单独下发一条增量改变就行了，为何大费周折的把所有的数据都同步上来，且全部配置下去？此处的操作存在三个硬伤：

第一，规则数量巨大的时候，同步上来，转换格式，插入新规则，转换格式，再配置下去，耗时较长，在上千条配置的情况下可能会出现几分钟等待时间的极端情况**(**即便只是一条 **show)**。

第二，规则的统计信息连续性是靠同步时附带当前规则统计信息的方式实现的。那么举个简单的例子，在 **10：00：00** 秒时某规则命中 **1000** 个包，此时用户进行了操作，**5** 分钟后操作完成，规则和这 **1000** 个包的信息又被配置进内核。表面看来同步信息还在延续，实际上这 **5** 分钟的统计信息是丢失了的。

第三，大量配置在内核 **replace** 时，势必要加锁操作。而且此操作时间单位可能是分钟，这段事件内，**netfilter** 过滤规则是会失效的。

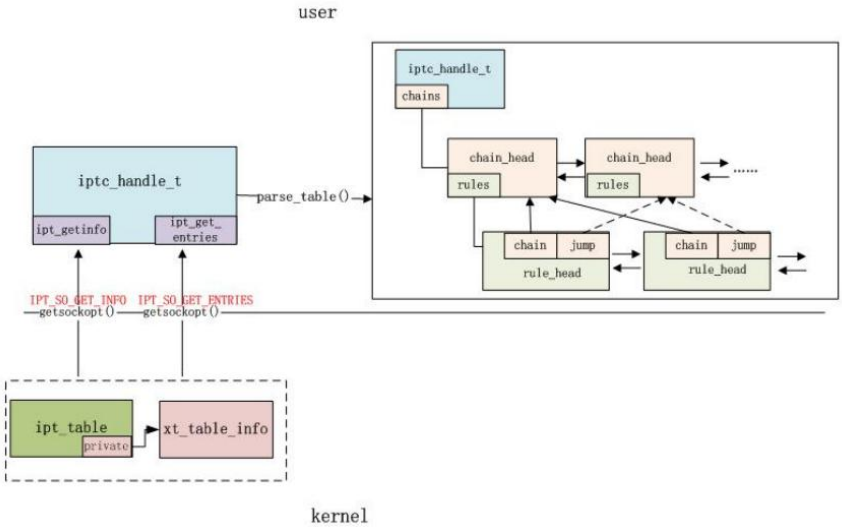

图 **47**

图有点老，不过大概意思是一样的。内核态是 **xt_table** 结构，其 **private** 指针指向的 **xt_table_info** 结构保存了相应结构。而用户态主要是 **xtc_handle** 结构。在这个层面是大差不差的，都是一种使用柔性数组，连续内存管理的数据结构，使用 **underflow/hookentry** 指示规则的分界**.**

继续往下走，这里需要先取出相应表中，**netfilter** 里所有的内容，注意此时 **handle** 还是空的，进入 **iptc_init** 函数，也即 **TC_INIT** 函数。这个函数作用就是取出**-**组织数据。

```
if (!*handle)
    *handle = iptc_init(*table);
```

图 **48**

这时取出的内容，就是像上文说的，类似 **1.2** 节与 **2.2.2** 节里的描述，使用两次 **getsockopt** 来获取相应的数据，并放置在 **handle** 结构体最后的柔性数组中。

从内核获取数据后，可以看到 **TC_INIT** 函数最后调用了一个叫做 **parse_table** 的函数，就是通过这个函数，规则完成了图 **47** 中的转换。

这里内核态的数据结构不赘述了，可以在 **2.1** 中看得到。着重讲一下用户态的。首先看一下 **xtc_handle** 的结构：

```
struct xtc_handle {
  int sockfd;                    /*不解释了，本次 socket 操作的 fd*/
  int changed;                   /*  并非每次的 iptables 配置都会修改数据的，这是个标志  */

  struct list_head chains;

  struct chain_head *chain_iterator_cur;
  struct rule_head *rule_iterator_cur;

  unsigned int num_chains;              /*自定链的数目  */
  ..........
  /*后面这俩也不重复了，前面都解释的很清楚了*/
  STRUCT_GETINFO info;
  STRUCT_GET_ENTRIES *entries;
};
```

其关系如下图所示：

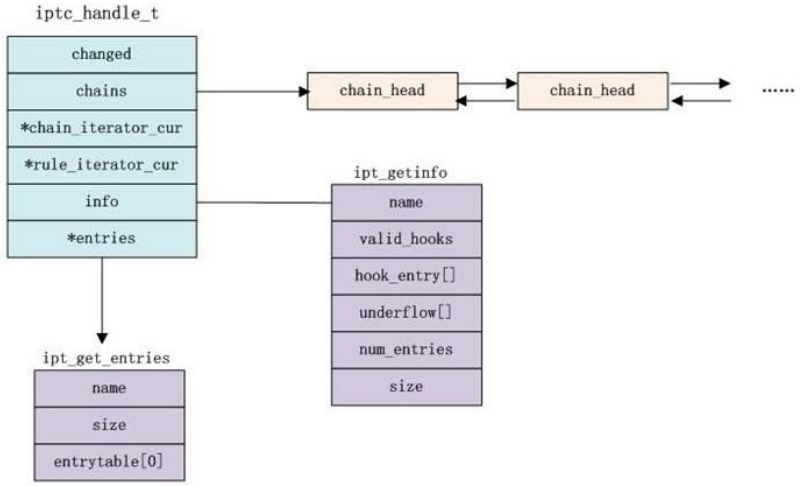

图 **49**

当在用户态中对取出的表内容进行处理后**(parse_table** 函数**)**，会把规则组织到链中保存。这样，**chains** 后面接了跟多 **chain_head**，这就是该表管理的所有链了，每个链都由这样的一个链头来保存。其中 **chain_iteratio_cur** 和 **rule_iterator_cur** 这两个指针，指向了当前遍历到的链和规则。虽然又增加了很多数据结构，但是只要保存的是 **iptables** 的规则，那么说到最后，它和内核中的 **ipt_tables** 里面的数据内容就完全是一样的，只是结构、位置不大一样。为什么取到用户态要进行这样的更改，其实不难理解，是因为连续内存形式存放的数据，基本是没法进行增删改变的。那么具体再看到链里：

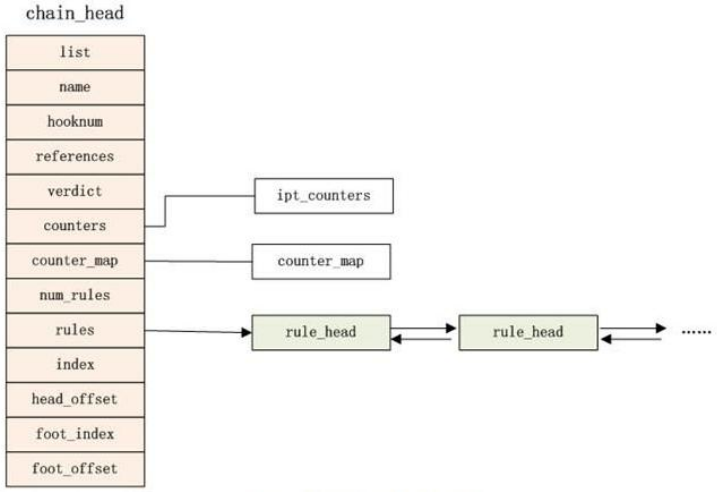

图 **50**

比较重要的就是 **counters** 记录了该链的统计值，**counter_map** 表示对该链统计值的操作，用于后面 **setsockopt** 设置相应值的时候起作用。**rules** 指向的是该链管理的所有规则，规则也是以链的形式保存。一个 **rule_head** 结构保存一条实际规则。

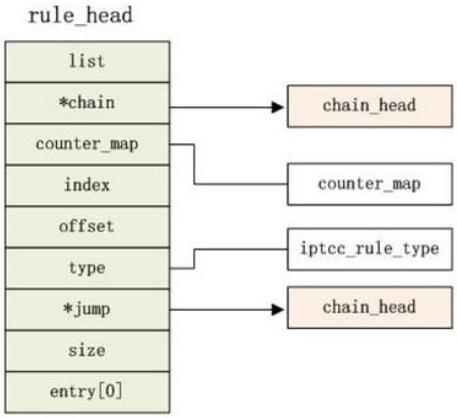

图 **51**

再往下可以看到 **rule_head** 的具体情况。**chain** 又回指该规则所在链**(**图 **47** 中可以看到相应的指向**)**，**jump** 指向的是可能跳转的其他链**(**未必会存在**)**。**type** 表示其所属类型，如标准 **TARGET** 和扩展操作，或者跳转等。可以看到，这里不同的 **verdict** 决定了不同的行为，注意 **FALLTHROUGH** 意为全放行。**entry[0]**后面就是我们的 **ipt_entry** 了。

后面对表的操作，可能需要实现对表中规则进行遍历。主要功能函数是 **iptc_first_chain**，**iptc_next_chain**，**iptc_first_rule()**，**iptc_next_rule()**等。

```
if (t->verdict < 0) {
    DEBUGP_C("standard, verdict=%d\n", t->verdict);
    r->type = IPTCC_R_STANDARD;
} else if (t->verdict == r->offset+e->next_offset) {
    DEBUGP_C("fallthrough\n");
    r->type = IPTCC_R_FALLTHROUGH;
} else {
    DEBUGP_C("jump, target=%u\n", t->verdict);
    r->type = IPTCC_R_JUMP;
    /* Jump target fixup has to be deferred
     * until second pass, since we migh not
     * yet have parsed the target */
}
```

图 **52**

最后的 **parse_table** 函数，其作用就是将使用 **getsockopt** 得到的连续 **buffer**，按照跳转目的关系，顺序关系，分拆成图 **47** 这样的规则链结构，并加上了引用计数，索引等。这样，**iptc_init** 函数就结束了，返回充满了规则的，用户态的 **xtc_handle.**

3.4 加入新配置的规则到规则集中

再次回到 **do_command4** 函数，经历了一些 **check** 函数后，开始下一部分工作，即将刚才经过解释后的参数，插入到 **iptc_init** 获取到的 **iptables** 规则集中，生成最新的规则集。

首先使用 **generate_entry** 函数，将刚才分析得到的结果**(match** 和 **target,**分别是后两个参数**)**，组合成一个 **ipt_entry** 结构，这个函数里的代码也可以很好的解释 **2.1.1** 中的数据结构。

```
        e = generate_entry(&cs.fw, cs.matches, cs.target->t);
        free(cs.target->t);
    }
}

switch (command) {
case CMD_APPEND:
    ret = append_entry(chain, e,
                nsaddrs, saddrs, smasks,
                ndaddrs, daddrs, dmasks,
                cs.options&OPT_VERBOSE,
                *handle);
    break;
```

图 **53**

由于之前我们执行的是**-A**，所以这里走到 **APPEND** 的选项中，可以从 **TC_APPEND_ENTRY** 开始的注释中很明确的看出来，这里是将之前填充的 **fw** 结构体**(ipt_entry)**，插入到指定的 **chain** 中。

```
iptc_fn = TC_APPEND_ENTRY;
if (!(c = iptcc_find_label(chain, handle))) {
    DEBUGP("unable to find chain `%s'\n", chain);
    errno = ENOENT;
    return 0;
}

if (!(r = iptcc_alloc_rule(c, e->next_offset))) {
    DEBUGP("unable to allocate rule for chain `%s'\n", chain);
    errno = ENOMEM;
    return 0;
}
```

图 **54**

接下来的操作较好理解，这里首先找到了相应的链**(chain)**，然后在增加了一条规则**(rule)**并分配内存**(**同时回指到相应的链上**)**，最后将之前组织好的 **e(ipt_entry)**复制到这个规则里，再将这个规则插入到链的最后，增加链上规则的计数，同时，走到这里说明我们这次的操作已经对整个规则集做了改变，将 **changed** 位修改。这里对数据结构如果不明白，看看之前的图 **49**，**50**，**51** 中 **xtc_handle, chain, rule** 的上下关系就明白了。

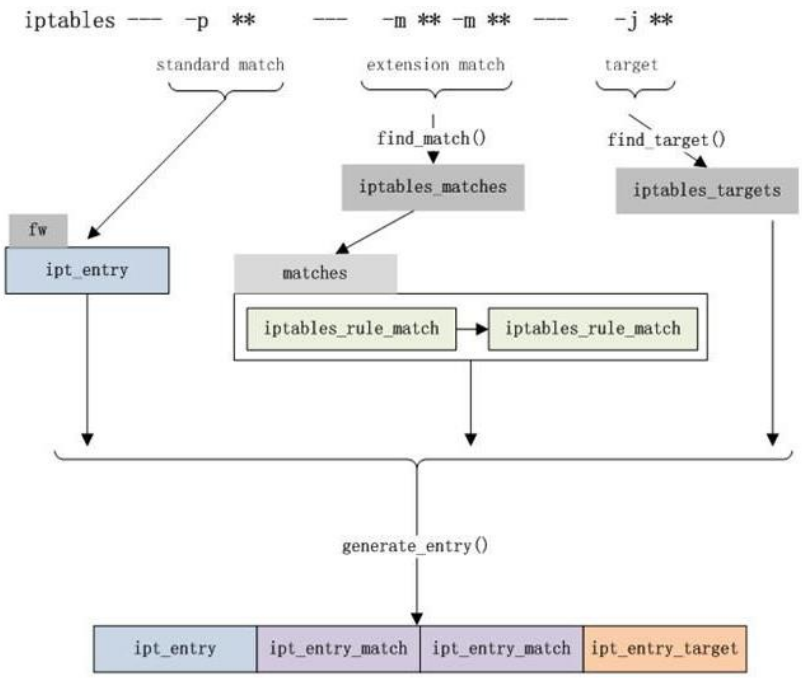

图 **55**

规则的分析，组配，大概就是由上图描述的这么一个流程。**extension** 的 **match** 在每次的分析中，逐步完全加载，插入到 **iptables_matches** 上的链表中，如下图所示，最后将该链上所有的 **match** 扩展结构一个个的填入到 **entry** 后面。

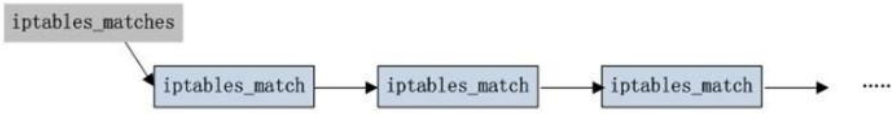

图 **56**

注意这里还有一个函数 **iptcc_map_target**，是用来设置当前 **rules** 的 **target**，回头看图 **51** 及相应字段的解释，这里就是要设置这些字段了：如果没设定 **target**，那这里的规则类型为全放行，如果是标准 **target**，那么设置相应的 **verdict**，并将 **target name** 置为 **STANDARD_TARGET**。

```
else {
  /* Maybe it's an existing chain name. */
  struct chain_head *c;
  DEBUGP("trying to find chain `%s': ", t->u.user.name);

  c = iptcc_find_label(t->u.user.name, handle);
  if (c) {
      DEBUGP_C("found!\n");
      r->type = IPTCC_R_JUMP;
      r->jump = c;
      c->references++;
      return 1;
  }
  DEBUGP_C("not found :(\n");
```

图 **57**

如果是一条自建链，那就设置相应的引用计数，跳转链和类型。

这时我们新增的规则已经被配置到以 **xtc_handle** 形式管理的一个规则集中了，释放一些中间变量，这部分的工作就算是完成了。

3.5 将最新的规则集下发到 netfilter

再回到图 **46**，执行 **iptc_commit** 即 **TC_COMMIT** 函数，这里可以看到之前 **change** 位的作用：

```c
/* Don't commit if nothing changed. */
if (!handle->changed)
    goto finished;

new_number = iptcc_compile_table_prep(handle, &new_size);
if (new_number < 0) {
    errno = ENOMEM;
    goto out_zero;
}
```

图 **58**

还记得 **3.3** 节中讲了内核和用户态数据结构不同的事情么，这里将要下发，就必须得再转换一次了。就是使用 **iptcc_compile_table_prep** 函数计算偏移**(**初始化 **head** 的各种 **offset**，各条 **rule** 的 **index** 等**)**，为 **repl** 指针申请空间**(**依然也是尾接柔性数组的结构**)**，填写 **valid hook** 等数据，并使用 **iptcc_compile_table** 添加相应数据到 **ipt_replace** 中

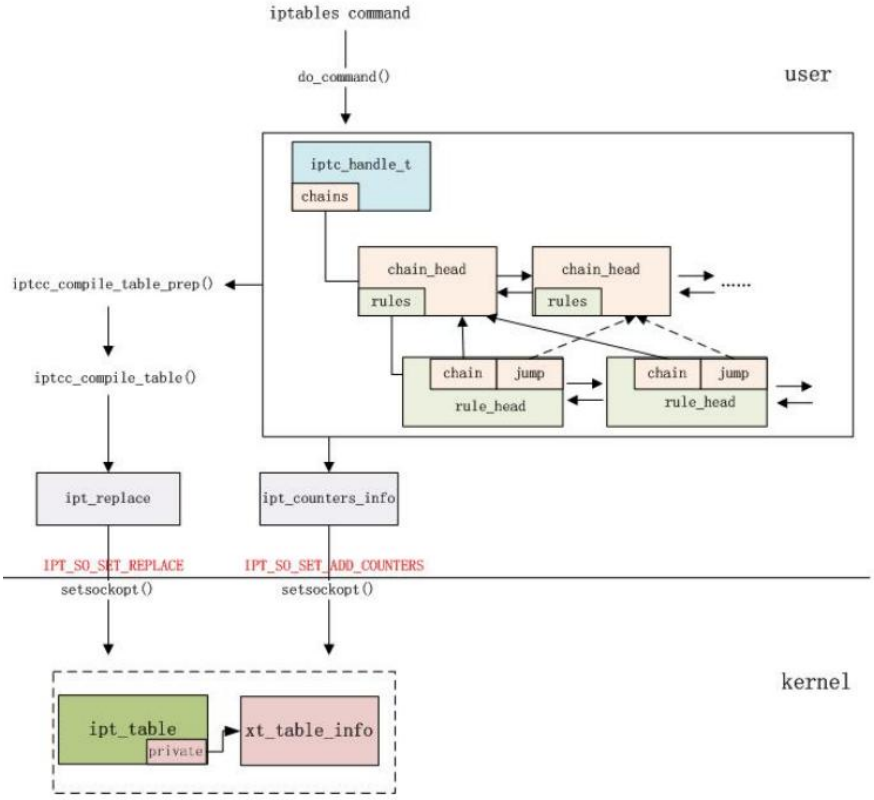

图 **59**

注意这里 **setsockopt** 使用的宏是 **SO_SET_REPLACE,**是用现有写进 **repl** 的数据整体替换 **netfilter** 里正在生效的数据，后面再对 **count** 计数进行设置，整个 **iptables** 规则配置就算是彻底完成了。

3.6 总结

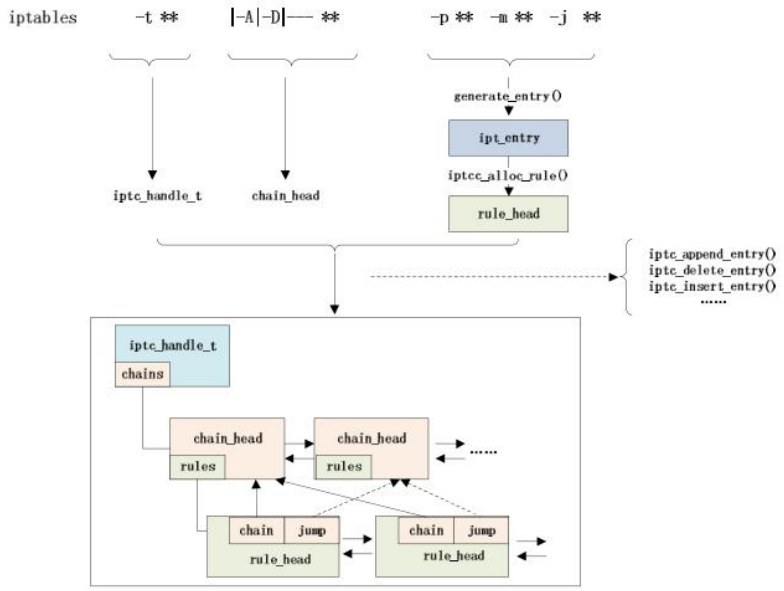

图 **60**

总的来说，**iptables** 规则解析并存放，及相应的数据结构，就是以上这样一个过程。 到此为止，**iptables** 从配置到读取，整个过程已经讲完了，可以多看看相关的结构体，如果对一些指针、链表的遍历不熟的话，可以看看代码里构建**/**解析的那些循环。**iptables** 大量使用了 **TLV** 结构，基本对数据的操作都是使用偏移进行的，有空多看看实际的代码偏移规则，就会比较好理解。

四、其他补充内容和需要改进的部分

内核每一个 **match** 匹配都是一个 **xt_action_param** 结构

hotdrop 参数允许模块将数据包直接丢弃。但是一般这不是 **match** 模块该做的事，只有部分 **match** 模块才会有。