Writing your own netfilter match

by Nicolas Bouliane
<nib(at)cookinglinux!org>

About the author:
Nicolas is a young warrior in the free software community. He's a gnu/linux addict since the day he installed it on his computer in 1998. He spend his time studying the linux networking stack, writing free softwares and attending at linux related conference like the OLS. When he's not in front of his computer, he likes watching sci−fi movies, playing chess and listening Richard Stallman's talk.

Abstract:

The iptables/netfilter framework gives us the possibility to add features. To do so, we write kernel modules that registers against this framework. Also, depending on the feature's category, we write an iptables module. By writing your new extension, you can match, mangle, give faith and track a given packet. In fact, you can do almost everything you want in this filtering world. Beware that a little error in a kernel module can severly crash the computer.

For the sake of simplicity, I will explain a skeleton match that I wrote. This way, I hope to make the interactions with the framework a little easier to understand. Here, I'll assume you already know a bit about iptables and that you know C programming.

This example will show you how to match a packet according to the source and/or destination address ip.

Description

The general steps around creating an iptables/netfilter's match module are:

- You want to match a specific situation.
- Write the user−space part which will handle arguments.
- Write the kernel−space part which will analyze packets and says to match or not.

1.0 The iptables module

The purpose of an iptables library is basically to interact with the user. It will handle the arguments the user want the kernel−part to take in consideration.

1.1 Structures and Functions available

At first, some basic structures. <*iptables/include/iptables.h*>
We will see later in this text what's the purpose of each field.

```
/* Include file for additions: new matches and targets. */
struct iptables_match
{
   struct iptables_match *next;

   ipt_chainlabel name;

   const char *version;

   /* Size of match data. */
   size_t size;

   /* Size of match data relevent for userspace comparison purposes */
   size_t userspacesize;

   /* Function which prints out usage message. */
   void (*help)(void);

   /* Initialize the match. */
   void (*init)(struct ipt_entry_match *m, unsigned int *nfcache);

   /* Function which parses command options; returns true if it
           ate an option */
   int (*parse)(int c, char **argv, int invert, unsigned int *flags,
           const struct ipt_entry *entry,
           unsigned int *nfcache,
           struct ipt_entry_match **match);

   /* Final check; exit if not ok. */
   void (*final_check)(unsigned int flags);

   /* Prints out the match iff non-NULL: put space at end */
   void (*print)(const struct ipt_ip *ip,
            const struct ipt_entry_match *match, int numeric);

   /* Saves the match info in parsable form to stdout. */
   void (*save)(const struct ipt_ip *ip,
           const struct ipt_entry_match *match);

   /* Pointer to list of extra command-line options */
   const struct option *extra_opts;

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

1.2 Inside the skeleton

1.2.1 Initialization

We initialize the common fields in the 'iptables_match' structure.

```
static struct iptables_match ipaddr
= {
```

'Name' is the file name string of your library (ie: libipt_ipaddr).
You can't provide another name, it's used for auto−loading your library.

```
    .name           = "ipaddr",
```

The next field, 'version', is the iptables's version. Both next are used to keep corelation between the user−space and kernel−space shared structure's size.

```
    .version        = IPTABLES_VERSION,
    .size           = IPT_ALIGN(sizeof(struct ipt_ipaddr_info)),
    .userspacesize  = IPT_ALIGN(sizeof(struct ipt_ipaddr_info)),
```

'Help' is called when a user enter 'iptables −m module −h'. 'Parse' is called when you enter a new rule, its duty is to validate the arguments. In the case of 'print', its called by 'iptables −L' to show previously enterd rules.

```
    .help           = &help,
    .init           = &init,
    .parse          = &parse,
    .final_check    = &final_check,
    .print          = &print,
    .save           = &save,
    .extra_opts     = opts
};
```

The iptables infrastructure can support multiple shared libraries. Each library must register to iptables by calling 'register_match()', defined into <*iptables/iptables.c*>. This function is called when the module is loaded by iptables. For more information about it: 'man dlopen'.

```
void _init(void)
{
   register_match(&ipaddr);
}
```

1.2.2 save function

If we have a ruleset that we want to save, iptables provide the tool 'iptables−save' which dumps all your rules. It obviously needs your extension's help to dump proper rules. This is done by calling this function.

```
static void save(const struct ipt_ip *ip, const struct ipt_entry_match *match)
{
   const struct ipt_ipaddr_info *info = (const struct ipt_ipaddr_info *)match->data;
```

We print out the source−addresse if it's part of the rule.

```
   if (info->flags & IPADDR_SRC) {
      if (info->flags & IPADDR_SRC_INV)
         printf("! ");
      printf("--ipsrc ");
```

```
      print_ipaddr((u_int32_t *)&info->ipaddr.src);
   }
```

We print out the destination−adresse if it's part of the rule.

```
   if (info->flags & IPADDR_DST) {
      if (info->flags & IPADDR_DST_INV)
         printf("! ");
      printf("--ipdst ");
      print_ipaddr((u_int32_t *)&info->ipaddr.dst);
   }
}
```

1.2.3 print function

In the same philosophy of the previous one, this function aims to print information about the rule. It's called by 'iptables −L'. We will see later in this text what's the purpose of 'ipt_entry_match *match', but you certainly already have a little idea about it.

```
static void print(const struct ipt_ip *ip,
                  const struct ipt_entry_match *match,
                  int numeric)
{
   const struct ipt_ipaddr_info *info = (const struct ipt_ipaddr_info *)match->data;

   if (info->flags & IPADDR_SRC) {
         printf("src IP ");
      if (info->flags & IPADDR_SRC_INV)
         printf("! ");
      print_ipaddr((u_int32_t *)&info->ipaddr.src);
   }

   if (info->flags & IPADDR_DST) {
      printf("dst IP ");
      if (info->flags & IPADDR_DST_INV)
         printf("! ");
      print_ipaddr((u_int32_t *)&info->ipaddr.dst);
   }
}
```

1.2.4 final check function

This function is a kind of last chance for sanity check. It's called when the user enter a new rule, right after arguments parsing is done.

```
static void final_check(unsigned int flags)
{
   if (!flags)
      exit_error(PARAMETER_PROBLEM, "ipt_ipaddr: Invalid parameters.");
}
```

1.2.5 parse function

This is the most important function because it's here that we verify if arguments are used correctly and set informations we will share with the kernel−part. It is called each time an argument is found, so if the user provides two arguments, it will be called twice with the argument code into the var 'c'.

```
static int parse(int c, char **argv, int invert, unsigned int *flags,
                 const struct ipt_entry *entry,
                 unsigned int *nfcache,
                 struct ipt_entry_match **match)
{
```

We use this special structure to keep informations we will share with the kernel−part. 'Match' pointer is passed to a couple of functions so we can work on the same data structure. Once the rule is loaded, this pointer is copied to the kernel−part. This way, the kernel module knows what the user asks to analyze (and that's the point, no?).

```
   struct ipt_ipaddr_info *info = (struct ipt_ipaddr_info *)(*match)->data;
```

Each arguments correspond to an single value, so we can do specific actions according to the inputed arguments. We will see later in this text how we map arguments to values.

```
   switch(c) {
```

First, we check if the argument has been used more than once. If it appears to be the case, we call 'exit_error()' defined in <*iptables/iptables.c*>, which will exit immediatly with the status flag 'PARAMETER_PROBLEM' defined in <*iptables/include/iptables_common.h*>. Else, we set 'flags' and 'info−>flags' to the 'IPADDR_SRC' value defined in our header's file. We will see this header's file later.

Although both var flags seems to have the same purpose, they really don't. The scope of 'flags' is only this function, and 'info−>flags' is a field part of our structure which will be shared with the kernel−part.

```
      case '1':
         if (*flags & IPADDR_SRC)
            exit_error(PARAMETER_PROBLEM, "ipt_ipaddr: Only use --ipsrc once!");
         *flags |= IPADDR_SRC;
         info->flags |= IPADDR_SRC;
```

We verify if the invert flag, '!', has been inputed and then set appropriate information into the 'info−>flags'. Next, we call 'parse_ipaddr', an internal function written for this skeleton, to convert the ip address string to a 32bits value.

```
         if (invert)
            info->flags |= IPADDR_SRC_INV;

         parse_ipaddr(argv[optind-1], &info->ipaddr.src);
         break;
```

In the same thought, we check for multiple use and set appropriate flags.

```
      case '2':
         if (*flags & IPADDR_DST)
            exit_error(PARAMETER_PROBLEM, "ipt_ipaddr: Only use --ipdst once!");
         *flags |= IPADDR_DST;
         info->flags |= IPADDR_DST;
         if (invert)
            info->flags |= IPADDR_DST_INV;
```

```
            parse_ipaddr(argv[optind-1], &info->ipaddr.dst);
            break;

         default:
            return 0;
      }

      return 1;
}
```

1.2.6 options structure

We have discussed earlier that every arguments are mapped to a single value. The 'struct option' is the better way to achieve it. For more information about this structure, I strongly suggest you read 'man 3 getopt'.

```
static struct option opts[] = {
   { .name = "ipsrc",   .has_arg = 1,   .flag = 0,   .val = '1' },
   { .name = "ipdst",   .has_arg = 1,   .flag = 0,   .val = '2' },
   { .name = 0 }
};
```

1.2.7 init function

This init function is used to set some specific stuff like the netfilter cache system. It's not very important to know how exactly it works for now.

```
static void init(struct ipt_entry_match *m, unsigned int *nfcache)
{
   /* Can't cache this */
   *nfcache |= NFC_UNKNOWN;
}
```

1.2.7 help function

This function is called by 'iptables –m match_name –h' to show available arguments.

```
static void help(void)
{
   printf (
             "IPADDR v%s options:\n"
             "[!] --ipsrc <ip>\t\t The incoming ip addr matches.\n"
             "[!] --ipdst <ip>\t\t The outgoing ip addr matches.\n"
             "\n", IPTABLES_VERSION
           );
}
```

1.2.8 the header's file 'ipt_ipaddr.h'

It's in this file that we define our stuff that we need.

```
#ifndef _IPT_IPADDR_H
#define _IPT_IPADDR_H
```

We have seen earlier that we set flags to some specific values.

```
#define IPADDR_SRC   0x01     /* Match source IP addr */
#define IPADDR_DST   0x02     /* Match destination IP addr */

#define IPADDR_SRC_INV  0x10  /* Negate the condition */
#define IPADDR_DST_INV  0x20  /* Negate the condition */
```

The structure 'ipt_ipaddr_info' is the one who will be copied to the kernel–part.

```
struct ipt_ipaddr {
   u_int32_t src, dst;
};

struct ipt_ipaddr_info {

   struct ipt_ipaddr ipaddr;

   /* Flags from above */
   u_int8_t flags;

};

#endif
```

1.3 Summary chapter 1

In the first part, we discussed the purpose of the iptables library. We covered the internals of each function and how the main structure 'ipt_ipaddr_info' is used to keep information that will be copied to the kernel–part for further consideration. We also look at the iptables structure and how to register our new library.
You should keep in mind that this is only a skeleton example to help me to show you how the framework is working. Furthermore, 'ipt_ipaddr_info' and things like that are not part of the iptables/netfilter but part of this example.

2.0 The netfilter module

The duty of a match module is to inspect each packet received and to decide if it matches or not according to our criteria. The module has the following means to do that:

- Receive each packet hitting the table related with the match module
- Tell netfilter if our module match the packet

2.1 Structures and Functions available

At first some basic structures. This structure is defined in <*linux/netfilter_ipv4/ip_tables.h*>.
If you are interested in learning more about this structure and the previous one presented for iptables, you should look at the netfilter hacking howto written by Rusty Russell and Harald Welte.

```
struct ipt_match
{
   struct list_head list;

   const char name[IPT_FUNCTION_MAXNAMELEN];

   /* Return true or false: return FALSE and set *hotdrop = 1 to
           force immediate packet drop. */
   /* Arguments changed since 2.4, as this must now handle
           non-linear skbs, using skb_copy_bits and
           skb_ip_make_writable. */
   int (*match)(const struct sk_buff *skb,
           const struct net_device *in,
           const struct net_device *out,
           const void *matchinfo,
           int offset,
           int *hotdrop);

   /* Called when user tries to insert an entry of this type. */
   /* Should return true or false. */
   int (*checkentry)(const char *tablename,
           const struct ipt_ip *ip,
           void *matchinfo,
           unsigned int matchinfosize,
           unsigned int hook_mask);

   /* Called when entry of this type deleted. */
   void (*destroy)(void *matchinfo, unsigned int matchinfosize);

   /* Set this to THIS_MODULE. */
   struct module *me;
};
```

2.2 Inside the skeleton

2.2.1 Initialization

We initialize the common fields in the 'ipt_match' structure.

```
static struct ipt_match ipaddr_match
= {
```

'Name' is the file name string of your module (ie: ipt_ipaddr).

```
        .name       = "ipaddr",
```

Next fields are callbacks that the framework will use. 'Match' is called when a packet is passed to your

module.

```
        .match      = match,
        .checkentry = checkentry,
        .me         = THIS_MODULE,
};
```

Your kernel module's init function needs to call 'ipt_register_match()' with a pointer to a 'struct ipt_match' to register against the netfilter framework. This function is called on module loading.

```
static int __init init(void)
{
        printk(KERN_INFO "ipt_ipaddr: init!\n");
        return ipt_register_match(&ipaddr_match);
}
```

When unloading the module this function is called. It's here that we unregister our match.

```
static void __exit fini(void)
{
        printk(KERN_INFO "ipt_ipaddr: exit!\n");
        ipt_unregister_match(&ipaddr_match);
}
```

We hand them functions that will be called at the loading and unloading of the module.

```
module_init(init);
module_exit(fini);
```

2.2.2 match function

The linux tcp/ip stack is sprink of five netfilter's hooks. Thus when a packet walks in, the stack passes the packet to the appropriate hook which iterates through each table which iterates through each rules. When it's the time to your module to have the packet, it can finally do its job.

```
static int match(const struct sk_buff *skb,
                 const struct net_device *in,
                 const struct net_device *out,
                 const void *matchinfo,
                 int offset,
                 const void *hdr,
                 u_int16_t datalen,
                 int *hotdrop)
{
```

Hope you remember what we did in the user−part ! :). Now we map the user−space's copied structure into our own one.

```
        const struct ipt_skeleton_info *info = matchinfo;
```

The 'skb' contains the packet we want to look at. For more information about this powerfull structure used everywhere in the linux tcp/ip stack, Harald Welte wrote an excellent article (ftp://ftp.gnumonks.org/pub/doc/skb−doc.html) about it.

```
   struct iphdr *iph = skb->nh.iph;
```

Here, we are just printing some funny stuff to see what they look like. The macro 'NIPQUAD' is used to display an ip address in readable format, defined in <*linux/include/linux/kernel.h*>.

```
   printk(KERN_INFO "ipt_ipaddr: IN=%s OUT=%s TOS=0x%02X "
                    "TTL=%x SRC=%u.%u.%u.%u DST=%u.%u.%u.%u "
                    "ID=%u IPSRC=%u.%u.%u.%u IPDST=%u.%u.%u.%u\n",

                    in ? (char *)in : "", out ? (char *)out : "", iph->tos,
                    iph->ttl, NIPQUAD(iph->saddr), NIPQUAD(iph->daddr),
                    ntohs(iph->id), NIPQUAD(info->ipaddr.src), NIPQUAD(info->ipaddr.dst)
          );
```

If the '−−ipsrc' argument has been inputed we look if the source address match with the one specified in the rule. We don't forget to take in consideration the invert flag: '!'. If we don't match: we return the verdict; 0.

```
   if (info->flags & IPADDR_SRC) {
      if ( (ntohl(iph->saddr) != ntohl(info->ipaddr.src)) ^ !!(info->flags & IPADDR_SRC_INV) ) {

         printk(KERN_NOTICE "src IP %u.%u.%u.%u is not matching %s.\n",
                            NIPQUAD(info->ipaddr.src),
                            info->flags & IPADDR_SRC_INV ? " (INV)" : "");
         return 0;
      }
   }
```

Here, we do the same, except that we look for the destination address if '−−ipdst' has been inputed.

```
   if (info->flags & IPADDR_DST) {
      if ( (ntohl(iph->daddr) != ntohl(info->ipaddr.dst)) ^ !!(info->flags & IPADDR_DST_INV) )  {

         printk(KERN_NOTICE "dst IP %u.%u.%u.%u is not matching%s.\n",
                            NIPQUAD(info->ipaddr.dst),
                            info->flags & IPADDR_DST_INV ? " (INV)" : "");
         return 0;
      }
   }
```

If both failed, we return the verdict 1, which means we matched the packet.

```
   return 1;
}
```

2.2.3 checkentry function

Checkentry is most of the time used as a last chance for sanity check. It's a bit hard to understand when it's called. For explanation, see this post (http://www.mail−archive.com/netfilter−devel@lists.samba.org/msg00625.html). This is also explained in the netfilter hacking howto.

```
static int checkentry(const char *tablename,
                            const struct ipt_ip *ip,
                            void *matchinfo,
                            unsigned int matchsize,
                            unsigned int hook_mask)
{
```

```
   const struct ipt_skeleton_info *info = matchinfo;

   if (matchsize != IPT_ALIGN(sizeof(struct ipt_skeleton_info))) {
      printk(KERN_ERR "ipt_skeleton: matchsize differ, you may have forgotten to recompile me.\n"
      return 0;
   }

   printk(KERN_INFO "ipt_skeleton: Registered in the %s table, hook=%x, proto=%u\n",
                    tablename, hook_mask, ip->proto);

   return 1;
}
```

2.3 Summary chapter 2

In this second part, we covered the netfilter's module and how to register it by using a specific structure. In addition, we discussed how to match a specific situation according to criteria provided by the user−space part.

3.0 Playing with iptables/netfilter

We have seen how to write a new iptables/netfilter's match module. Now, we would like to add it in our kernel to play with it. Here, I assume that you know how to build/compile a kernel. First, get the skeletons's match files from the download page for this article.

3.1 iptables

Now, if you don't have the source of iptables you can download it ftp://ftp.netfilter.org/pub/iptables/. Then you have to copy 'libipt_ipaddr.c' into <*iptables/extensions/*>.

This is a line from <*iptables/extensions/Makefile*> in which you have to add 'ipaddr'.

```
PF_EXT_SLIB:=ah addrtype comment connlimit connmark conntrack dscp ecn
esp hashlimit helper icmp iprange length limit ipaddr mac mark
multiport owner physdev pkttype realm rpc sctp standard state tcp tcpmss
tos ttl udp unclean CLASSIFY CONNMARK DNAT DSCP ECN LOG MARK MASQUERADE
MIRROR NETMAP NOTRACK REDIRECT REJECT SAME SNAT TARPIT TCPMSS TOS TRACE
TTL ULOG
```

3.2 kernel

First, you have to copy 'ipt_ipaddr.c' in <*linux/net/ipv4/netfilter/*> and 'ipt_ipaddr.h' into <*linux/include/linux/netfilter_ipv4/*>. Some of you are still using linux 2.4, so I'll present both 2.4 and 2.6 files to edit.

For 2.4, edit <*linux/net/ipv4/netfilter/Config.in*> and add the bold line.

```
# The simple matches.
  dep_tristate '  limit match support' CONFIG_IP_NF_MATCH_LIMIT $CONFIG_IP_NF_IPTABLES
  dep_tristate '  ipaddr match support' CONFIG_IP_NF_MATCH_IPADDR $CONFIG_IP_NF_IPTABLES
```

Then, edit <*linux/Documentation/Configure.help*> and add the text in bold. I copied some text to help you find where you add yours.

```
limit match support
CONFIG_IP_NF_MATCH_LIMIT
  limit matching allows you to control the rate at which a rule can be
  ...
ipaddr match support
CONFIG_IP_NF_MATCH_IPADDR
  ipaddr matching. etc etc.
```

Finally, you have to add this bold line into <*linux/net/ipv4/netfilter/Makefile*>.

```
# matches
obj-$(CONFIG_IP_NF_MATCH_HELPER) += ipt_helper.o
obj-$(CONFIG_IP_NF_MATCH_LIMIT) += ipt_limit.o
obj-$(CONFIG_IP_NF_MATCH_IPADDR) += ipt_ipaddr.o
```

Now for 2.6, files to edit are <*linux/net/ipv4/netfilter/Kconfig*> and <*linux/net/ipv4/netfilter/Makefile*>.

Conclusion

It just remains you to recompile and add what I forgot to tell you.
Happy hacking!!
Thanks to Samuel Jean.

Webpages maintained by the LinuxFocus Editor team © Nicolas Bouliane "some rights reserved" see linuxfocus.org/license/ http://www.LinuxFocus.org	Translation information: en −−> −− : Nicolas Bouliane <nib(at)cookinglinux!org>

2005−07−22, generated by lfparser_pdf version 2.51