Ecris tes propres règles pour NetFilter

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

L´auteur:
Nicolas est un jeune guerrier dans la communauté du logiciel libre. Il est accro à GNU/Linux depuis le jour où il l'a installé sur son ordinateur en 1998. Il passe son temps à étudier la couche réseau de linux, à écrire des logiciels libres et à assister à des conférences sur linux, comme la OLS. Quant il n'est pas devant son ordinateur, il aime regarder des films de science−fiction, jouer aux échecs et écouter les discours de Richard Stallman.

Résumé:

L'interface du couple iptables/netfilter nous permet d'ajouter des fonctionnalités. Pour ce faire, nous écrivons des modules du noyau qui s'enregistrent par dessus cette interface. Donc, suivant la catégorie de la fonctionnalité, nous écrivons un module pour iptables. En écrivant notre nouveau module, nous pouvons détecter, transformer, accepter et suivre un paquet donné. En fait, tu peux faire presque tout ce que tu veux, dans ce monde du filtrage. Attention, une petite erreur dans un module du noyau peu endommager sévèrement ton ordinateur.

Pour rester simple, je vais expliquer un squelette de règle que j'ai écrit. Comme ça, j'espère qu'il sera plus simple de comprendre le lien entre l'extension et l'interface. Bon, je suppose que tu connais déjà un peu iptables et que tu sais programmer en langage C.

Cet exemple va te montre comment détecter un paquet en fonction de son adresse IP source et/ou de celle de destination.

Description

Les étapes générales pour créer un module de filtrage iptables/netfilter sont:

- Tu veux détecter une situation particulière.
- Écrire la partie dans l'espace utilisateur, qui va traiter les arguments.
- Écrire la partie dans l'espace noyau, qui va analyser les paquets et dire s'il y a concordance ou pas.

1.0 Le module iptables

Le but premier d'une bibliothèque iptables est d'interagir avec l'utilisateur. Elle va traiter les arguments que l'utilisateur veut faire passer à la partie noyau.

1.1 Structures et fonctions disponibles

Tout d'abord, quelques structures de base. *<iptables/include/iptables.h>*
Nous verrons plus tard dans cet article quel est la signification de chaque champ.

```
/* 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 À l'intérieur du squelette

1.2.1 Initialisation

Nous initialisons les champs communs dans la structure 'iptables_match'.

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

'Name' est le nom de fichier de ta bibliothèque (ie: libipt_ipaddr).
Tu peux donner un autre nom, il est utilisé pour l'auto−chargement de la bibliothèque.

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

Le champ suivant, 'version', est la version d'iptables. Les deux suivants sont utilisés pour la corrélation entre la taille de la structure partagée entre l'espace utilisateur et l'espace noyau.

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

'Help' est appelée quand l'utilisateur tape 'iptables −m module −h'. 'Parse' est appelée quand tu entres une nouvelle règle, son rôle est de valider les arguments. Quant à 'print', elle est appelée lorsque la commande 'iptables −L' est tapée pour afficher les règles précédemment définies.

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

L'infrastructure iptables peut supporteer de multiples bibliothèques partagées. Chacune doit s'enregistrer auprès d'iptables en appelant 'register_match()', qui est définie dans <*iptables/iptables.c*>. Cette fonction est appelée lorsque le module est chargé par iptables. Pour obtenir plus d'informations: 'man dlopen'.

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

1.2.2 Fonction save

Si nous avons un ensemble de règles que nous voulons sauvegarder, iptables fourni la commande 'iptables−save' qui décharge toutes tes règles. Il a évidemment besoin de l'aide de ton module pour décharger tes propres règles. C'est fait en appelant cette fonction.

```
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;
```

Nous affichons l'adresse source si elle fait partie de la règle.

```
   if (info->flags & IPADDR_SRC) {
```

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

Nous affichons l'adresse de destination si elle fait partie de la règle.

```
   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 Fonction print

C'est la même philosophie que la précédente, le but de cette fonction est d'afficher des informations sur les règles. Elle est appelée par 'iptables –L'. Nous verrons plus tard dans cet article le rôle de 'ipt_entry_match *match', mais tu as sans doute déjà une petite idée là–dessus.

```
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 Fonction final check

Cette fonction est une sorte de dernière chance pour le test de validation. Elle est appelée lorsque l'utilisateur entre une nouvelle règle, juste après l'analyse des arguments.

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

1.2.5 Fonction parse

C'est la fonction la plus importante, car c'est ici que nous vérifions si les arguments sont utilisés correctement et que nous stockons les informations partagées avec la partie noyau. Elle est appelée chaque fois qu'un argument est trouvé, donc si l'utilisateur fourni deux arguments, elle sera appelée deux fois avec le code de l'argument dans la variable '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)
{
```

Nous utilisons cette structure spéciale pour stocker les informations que nous voulons partager avec la partie noyau. Le pointeur 'Match' est passé à plusieurs fonctions donc nous travaillons sur la même structure de données. Une fois que la règle est chargée, ce pointeur est copié dans la partie noyau. Comme ça, le module du noyau sait ce que l'utilisateur demande à analyser (et c'est le but, non?).

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

Chaque arguments correspond à une simple valeur, donc nous pouvons faire des actions spécifiques suivant les arguments passés. Nous verrons plus tard dans cet article comment faire correspondre des valeurs aux arguments.

```
   switch(c) {
```

Tout d'abord, nous testons si l'argument a été utilisé plus d'une fois. Si c'est le cas, nous appelons 'exit_error()' définie dans <*iptables/iptables.c*>, qui sort immédiatement avec le code de retour 'PARAMETER_PROBLEM' défini dans <*iptables/include/iptables_common.h*> Autrement, nous initialisons 'flags' et 'info−>flags' à la valeur 'IPADDR_SRC' définie dans notre fichier d'en−tête. Nous verrons ce fichier d'en−tête plus tard.

Bien que les deux variables drapeaux semblent avoir le même dessein, ce n'est pas du tout le cas. La portée de 'flags' est seulement cette fonction, et 'info−>flags' est un champ qui fait partie de la structure qui sera partagée avec la partie noyau.

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

Nous vérifions si le drapeau inverseur, '!', a été introduit, et nous mettons l'information adéquate dans 'info−>flags'.
Ensuite, nous appelons 'parse_ipaddr', une fonction interne écrite pour ce squelette, pour convertir la chaine de l'adresse IP en une valeur sur 32bits.

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

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

Dans la même idée, nous vérifions une utilisation multiple et mettons les drapeaux idoines.

```
      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 Structure des options

Nous avons dit plus tôt que chacun des arguments est relié à une valeur simple. La structure 'struct option' est le meilleur moyen pour faire cela. Pour plus d'informations sur cette structure, je te recommande fortement de lire '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 Fonction init

Cette fonction init est utilisée pour définir des choses spécifiques, comme le système de cache de netfilter. Ce n'est pas très important de savoir précisemment comment elle fonctionne pour le moment.

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

1.2.7 Fonction help

Cette fonction est appelée par 'iptables –m match_name –h' pour afficher les arguments utilisables.

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

```
            "\n", IPTABLES_VERSION
         );
}
```

1.2.8 Le fichier d'en–tête 'ipt_ipaddr.h'

C'est dans ce fichier que nous définissons les macros dont nous avons besoin.

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

Nous avons vu plutôt que nous définissons des drapeaux pour certaines valeurs.

```
#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 */
```

La structure 'ipt_ipaddr_info' est celle qui sera copiée dans la partie noyau.

```
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 Sommaire du chapitre 1

Dans la première partie, nous avons discuté du but de la bibliothèque iptables. Nous avons regardé l'intérieur de chaque fonction et comment la structure 'ipt_ipaddr_info' est utilisée pour garder les informations qui seront copiées dans la partie noyau pour une utilisation ultérieure. Nous avons regardé la structure iptables et comment enregistrer notre nouvelle bibliothèque.
Tu dois garder à l'esprit, que ce n'est qu'un squelette d'exemple pour m'aider à te montrer comment fonctionne l'interface. Du reste 'ipt_ipaddr_info' et les choses comme ça ne font pas partie d'iptables/netfilter mais de cet exemple.

2.0 Le module netfilter

L'objet d'un module de détection est d'inspecter chaque paquet reçu et de décider s'il correspond ou non à nos critères. Le module fait ça de cette manière:

- Recevoir chaque paquet touchant la table relative au module de détermination de correspondance
- Dire à netfilter si le paquet correspond aux règles de notre module ou pas

2.1 Structures et fonctions disponibles

Tout d'abord quelques structures de base. Celle–ci est définie dans <*linux/netfilter_ipv4/ip_tables.h*>.
Si tu as envie d'en apprendre plus sur cette structure et celle précédemment présentée pour iptables, tu dois lire netfilter hacking howto écrit par Rusty Russell et 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 A l'intérieur du squelette

2.2.1 Initialisation

Nous initialisons les champs communs de la structure 'ipt_match'.

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

'Name' est la chaîne du nom de fichier de ton module (ie: ipt_ipaddr).

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

Les champs suivant sont les fonction de rappel que l'interface utilisera. 'Match' est appelé lorsque le paquet est passé à ton module.

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

La fonction de votre module noyau init doit appeler 'ipt_register_match()' avec un pointeur sur une structure 'struct ipt_match' pour s'enregistrer auprès de l'interface de netfilter. Cette fonction est appelée au chargement du module.

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

La fonction suivante est appelée au déchargement du module. C'est ici que l'on efface nos règles.

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

Nous passons les fonctions qui seront appelées au chargemenr et au déchargement du module.

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

2.2.2 Fonction match

La couche tcp/ip de linux dispose de 5 hooks netfilter. Lorsqu'un paquet arrive, la couche le transmet au hook approprié, qui le fait passer dans chaque table qui le compare à chaque règle. Lorsque c'est au tour de ton module d'avoir le paquet, il peut faire son travail.

```
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)
{
```

J'espère que tu te rappelles que nous avons fait la partie utilisateur ! :) Maintenant nous associons la structure copiée dans l'espace utilisateur à la notre.

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

'skb' contient le paquet que nous voulons examiner. Pour plus d'information sur cette structure puissante utilisée partout dans la couche tcp/ip de linux, Harald Welte a écrit un excellent article (ftp://ftp.gnumonks.org/pub/doc/skb−doc.html) dessus.

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

Ici, nous affichons juste des choses amusantes pour voir à quoi elles ressemblent. La macro 'NIPQUAD', utilisée pour afficher une adresse IP dans un format lisible, est définie dans <*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)
          );
```

Si l'argument '−−ipsrc' a été fourni nous regardons si l'adresse source correspond avec celle spécifiée dans la règle. Nous n'oublions pas de prendre en considération le drapeau inverseur: '!'. Si ça ne correspond pas, nous retournons le 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;
      }
   }
```

Ici, nous faisons de même, sauf que nous regardons l'adresse de destination si l'argument '−−ipdst' a été fourni.

```
   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;
      }
   }
```

Si les deux échouent, nous retournons le verdict 1, qui signifie que le paquet correspond.

```
   return 1;
}
```

2.2.3 Fonction checkentry

Checkentry est utilisée la plupart du temps comme une dernière chance de correspondance. C'est un peu difficile de comprendre quand a−t−elle été appelée, regarde ça post (http://www.mail−archive.com/netfilter−devel@lists.samba.org/msg00625.html). C'est aussi expliqué dans le

```
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 Sommaire du chapitre 2

Dans ce second chapitre, nous avons couvert le module netfilter et la manière de l'enregistrer en utilisant une structure spécifique. Ensuite, nous avons discuté de la manière de détecter une certaine situation suivant les critères fournis par la partie utilisateur.

3.0 Jouer avec iptables/netfilter

Nous avons vu comment écrire un nouveau module de règles iptables/netfilter. Maintenant, nous voulons l'ajouter dans notre noyau pour jouer avec. Ici, je suppose que tu sais comment construire/compiler une noyau. Premièrement, récupérons les sources du squelette depuis La page de téléchargement pour cet article.

3.1 iptables

Maintenant, si tu n'as pas les sources d'iptables, tu peux les télécharger ftp://ftp.netfilter.org/pub/iptables/. Ensuite, tu dois copier 'libipt_ipaddr.c' dans <*iptables/extensions/*>.

C'est une ligne de <*iptables/extensions/Makefile*> dans laquelle tu dois ajouter '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 Noyau

Tout d'abord, tu dois copier 'ipt_ipaddr.c' dans <*linux/net/ipv4/netfilter/*> et 'ipt_ipaddr.h' dans <*linux/include/linux/netfilter_ipv4/*>. Certain d'entre vous utilisent encore linux 2.4, donc je vais présenter les fichiers à éditer des deux versions.

Pour 2.4, édites <*linux/net/ipv4/netfilter/Config.in*> et ajoutes la ligne en gras.

```
# 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
```

Ensuite, édites <*linux/Documentation/Configure.help*> et ajoutes le texte en gras. J'ai copié plus de texte afin de te permettre de trouver où ajouter le tien.

```
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.
```

Finalement, tu dois ajouter cette ligne en gras dans <*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
```

Maintenant pour 2.6, les fichiers à éditer sont <*linux/net/ipv4/netfilter/Kconfig*> et <*linux/net/ipv4/netfilter/Makefile*>.

Conclusion

Il ne te reste plus qu'à recompiler et ajouter ce que j'ai oublié de te dire.
Joyeux hacking!!
Merci à Samuel Jean.
