Focus

Extensions personnalisées pour IPTables

Jarosław Sajko

Degré de difficulté

Il n'est pas toujours facile de traduire une stratégie de défense dans une configuration d'un pare-feu réseau. Souvent, nous avons besoin d'une fonctionnalité qui n'est pas disponible dans notre pare-feu. Si ce pare-feu est basé sur IPTables, nous pouvons implémenter une telle fonctionnalité en tant que module d'extension. De plus, nous pouvons être étonné par sa simplicité.

Tout le monde connaissant un peu Internet et les ordinateurs a certainement entendu parler des services de type *pare-feu*. Toute personne qui s'occupe de la sécurité informatique a configuré plusieurs fois des services de ce type. Les systèmes de pare-feux disponibles sur le marché diffèrent sur plusieurs points. Du point de vue technique, ce sont les différences dans les fonctionnalités offertes qui sont les plus importantes. Les fournisseurs de solutions commerciales assurent que leurs programmes possèdent une fonctionnalité unique et avancée non disponible dans d'autres produits, les possibilités infinies et promettent que les employés souriants du support technique seront à notre entière disposition.

Ce dernier est parfois inévitable, mais nous serions plus contents d'obtenir un produit opérationnel au lieu d'échanger des emails avec le support technique. Nous voudrions aussi savoir et comprendre pourquoi certaines choses ne fonctionnent pas. Nous découvrons que les performances du programme sont non seulement finis, mais souvent insuffisantes. Une seule bonne nouvelle est que nous pouvons nous-mêmes construire une fonctionnalité unique et avancée à partir des solutions gratuites, grâce à notre temps libre et à l'aide – j'espère – de cet article. Bien sûr, cela ne veut pas dire que je considère les solutions commerciales comme inutiles.

Le paquet *IPTables* peut être téléchargé à l'adresse *www.netfilter.org*. Il est aussi disponible dans Linux en standard avec les noyaux 2.4 et ultérieurs. La version abrégrée du projet se trouve sur la page mentionnée. Le projet existe depuis la fin des années 90, alors assez longtemps pour le considérer comme produit de confiance. À mon avis,

Cet article explique...

- comment écrire notre propre extension pour *IPTables*.

Ce qu'il faut savoir...

- les notions de base du protocole IP,
- les notions de base du fonctionnement des systèmes d'exploitation,
- le langage C.

Chargement de l'extension – structures

En fonction du type d'extension chargée (match ou target), on utilise la fonction appropriée, respectivement `ipt_register_match` ou `ipt_register_target`. Chacune d'elles, en tant que paramètre d'entrée obtient la structure convenable elle-même (mais similaire). Il y a cinq champs à remplir, à savoir :

- `name` – le nom du module d'extension, il est recommandé qu'il soit identique à la partie du nom du fichier du module (les fichiers des modules d'extension s'appellent généralement *ipt_NOM.o*),
- `me` – dans ce champ, il faut saisir le texte littéral `THIS_MODULE`, ce qui signifie le pointeur sur soi-même et est important pour le compteur de renvois au module, et de cela, pour la fonction `cleanup_module`,
- `checkentry` – ici, nous entrons le pointeur sur la fonction qui est appelée au moment de l'ajout d'une règle exploitant ce module. Cette fonction doit, avant tout, vérifier la validité de cette règle,
- `destroy` - ce champ peut contenir le pointeur vers la fonction qui est appelée au moment de la suppression de la notation de ce type,
- `match` ou `target` (en fonction du type d'extension) – le plus importants, le pointeur vers la fonction qui décide de l'association du paquet ou exécute les opérations définies pour celui-ci.

La structure pour le match s'appelle `struct ipt_match`, et pour le target `struct ipt_target`.

l'une des qualités les plus intéressantes du paquet *IPTables* est la possibilité de construire pour lui des extensions personnalisées.

Netfilter est à vrai dire, un cadre d'application (en anglais *framework*) permettant de filtrer et de modifier des paquets. Il se compose de trois éléments essentiels :

- *les accroches* (*hooks* en anglais) sont les endroits déterminés sur la pile de protocole du système d'exploitation partir desquels est appelé le code *Netfilter* pour chaque paquet traversant la pile de protocoles,
- l'association aux *points d'accrochage* concrets des fonctions que *Netfilter* appellera pour les paquets traversant la pile de protocoles,
- le pilote *ip_queue* permettant la mise en file d'attente des paquets dans l'espace utilisateur pour leur traitement ultérieur ; ce transfert se fait de façon asynchrone.

Outre des éléments, un élément moins fonctionnel du cadre d'application est utile : il s'agit des commentaires disponibles dans le code source.

IPTables est, avant tout, un jeu de modules utilisant la fonctionnalité *Netfilter*, qui définit les tables de règles, ainsi que les critères d'association d'un paquet au modèle et des actions déterminées entreprises pour les paquets associés. Ainsi, nous pouvons manipuler la fonctionnalité *Netfilter* à partir du niveau d'abstraction plus élevé, de manière plus claire et plus commode. Le nom *IPTables* provient du fait que les listes des règles sont présentées sous forme de tableaux et elles sont stockées dans la mémoire en tant que tableaux à une nom donné.

En général, *IPTables* peut être divisé en deux parties : une partie liée aux services de translation des adresses et des ports (PNAT – *port and network address translation*) et l'autre partie relative aux services de filtrage. Les deux parties sont extensibles. Hormis les modules d'extension, il y a encore des outils de l'espace utilisateur, servant principalement à saisir les règles sous une forme plus lisible ou plus commode.

Modules d'extension

Le module d'extension est un module standard du noyau. Il doit implémenter les fonctions définies pour le module d'extension en utilisant pour cela certaines structures standards. De plus, encore une exigence est nécessaire pour un tel module. Son code doit être réentrant car il se peut que lors de la gestion d'un paquet, arrive une commande demandant l'interruption et la gestion d'un autre. Dans les systèmes SMP contenant plusieurs processeurs, la probabilité de cet évènement augmente de façon importante. Voici la liste des fonctions de base exigées par ce module :

- `init_module(...)` – le point d'entrée du module, sa tâche principale consiste à charger le module dans le cadre d'application et retourner 0 ou une valeur négative, si le chargement échoue,
- `cleanup_module(...)` – le point de sortie du module, le code de cette fonction doit décharger le module du cadre *Netfilter,*
- `ipt_register_match(...)` – prend en paramètre la structure `struct ipt_match` et sert à enregistrer les extensions des matches (associations),
- `ipt_register_target(...)` – prend en paramètre la structure `struct ipt_target` et sert à enregistrer les extensions des targets (cibles),
- `ipt_unregister_match(...)` – décharge l'extension du match,
- `ipt_unregister_target(...)` – décharge l'extension du target.

D'habitude, ce module exécute l'une de ces fonctions, de match ou de target, alors du jeu ci-dessus, nous choisissons respectivement quatre fonctions. En réalité, pour nous faciliter la tâche, nous implémentons les fonctions qui s'appellent un peu différemment et nous nous servons des macros standards. Les structures des fonctions `ipt_register_match` et `ipt_register_target` ont été présentées dans l'encadré.

Outre les fonctions décrites, nous devons aussi implémenter celles qui résultent du type d'extension et auxquels nous avons saisi les pointeurs dans la structure lors du chargement. Après cette opération, nous avons notre module tout prêt. Je le montrerai sur un exemple concret.

Implémentation d'un exemple d'une extension

Revenons pour quelques instants aux solutions commerciales. Plusieurs d'entre elles offrent les fonctions regroupées en paquets et disponibles via une interface graphiques à l'aide de quelques clics. Dans une certaine solution commerciale, il existe un ensemble de quelques fonctions simples appelé *Fingerprint Scrambling*. Sa tâche consiste à rendre plus difficile la reconnaissance distante du nom et de la version du système d'exploitation (cf. l'encadré *OS fingerprinting*). Dans notre cas, ce seront les extensions ipt_TTL, ipt_IPID et ipt_ISN. L'extension modifiante TTL d'un datagramme IP est disponible par défaut, et quant aux autres, nous allons les écrire, au moins en partie. Je dois bien laisser aux lecteurs un devoir à la maison.

Au début, nous nous occuperons d'ipt_IPID. Vu que le nom n'explique pas tout, décrivons d'abord le but de cette extension. L'un des facteurs qui aident à la détection d'un système d'exploitation distant est la possibilité de classifier correctement l'algorithme déterminant le champ ID du datagramme IP dans le système distant. Pour en savoir plus, référez-vous aux informations dans l'encadré. Notre tâche consistera donc à modifier le champ ID dans les datagrammes IP pour que leur classification soit incorrecte.

ipid_checkentry

D'après ce que j'ai écris ci-dessus, le module d'extension est chargé dans *IPTables* au moyen de la structure dans laquelle il faut donner quelques pointeurs sur les fonctions. Commençons par le champ `checkentry` de cette structure.

La fonction à laquelle nous entrons le pointeur dans ce champ est appelée pour chaque règle saisie utilisant l'extension donnée. En tant que paramètres d'entrée, elle obtient :

- le nom du tableau à laquelle la règle est ajoutée,
- la notation ajoutée sous forme de structure `ipt _ entry`,

OS fingerprinting

L'une des premières phases de l'attaque est la collecte des informations sur l'objectif. En cas d'attaque dans le monde informatique, les informations suivantes sont importantes : le type et la version du système d'exploitation et différentes versions des applications du système qui sera attaqué.

Sans l'accès local à l'ordinateur, nous utilisons la méthode de détection distante d'un système au moyen du « relevé d'une empreinte digitale » (en anglais *fingerprinting*) des paquets des protocoles réseau que nous obtenons de la part du système. Nous effectuons ce qu'on appelle *OS fingerprinting*.

Fingerprinting peut être passif ou actif. Le fingerprinting passif consiste à écouter seulement les paquets envoyés par le système scanné. Par contre, dans le fingerprinting actif, nous imposons la distribution des paquets en envoyant les requêtes, les tentatives d'établissement d'une session TCP, etc.

Les paquets reçus sont analysés d'une manière similaire. On prend en compte le mode d'implémentation obligatoires et facultatives des fonctions des protocoles. À partir de ces informations, il est possible d'estimer le type et la version du système.

Champ Identification du datagramme IP

L'en-tête de chaque datagramme IP contient le champ ID dont le but est d'aider au réassemblage des datagrammes fragmentés. Les fragments appartenant au même datagramme ont le même ID unique. C'est un mot d'une longueur de 16 bits, alors théoriquement, il permet de défragmenter simultanément 65536 paquets sur un noeud. Lors de la transmission sans défragmentation, il n'est pas important. Pourtant, différents systèmes d'exploitation emploient différents algorithmes pour déterminer la valeur de ce champ. Certains augmentent sa valeur pour chaque datagramme envoyé d'une unité fixe, d'autres augmentent d'une valeur aléatoire prédéfinie et encore d'autres, pour chaque datagramme tirent au sort un numéro arbitraire.

En fonction du système, il y a encore des nuances supplémentaires liées à la gestion de cette valeur. Le trait caractéristique des versions plus anciennes de certains systèmes d'exploitation est le fait que pour les datagrammes avec le bit DF (*Don't Fragment*) configuré, la valeur de ce champ est toujours égal à 0.

Dans les versions plus récentes de Linux on peut observer que pour les segments SYN/ACK des connexions TCP, cette valeur est configurée toujours à 0. Il existe plus de traits caractéristiques de ce type pour chaque implémentation.

Étant donné toutes ces différences entre les systèmes d'exploitation, ce champ est très important pour la reconnaissance distante de la version d'un système d'exploitation. La valeur de ce champ est utilisée, par exemple par *nmap* (scanneur actif) ou *p0f* (scanneur passif).

Au moyen du scanneur *nmap,* il est possible de vérifier quel algorithme est utilisé par un système donné (il suffit de le lancer avec l'option *–v* et *–O*). La prévisibilité des numéros d'identification successifs des datagrammes IP est aussi importante pour les raisons de sécurité.

Ces noeuds du réseau dont les datagrammes IP ont une valeur ID facile à prévoir, peuvent être exploitées pendant l'analyse des réseaux, parfois ceux qui ne sont pas normalement disponibles (ce qu'on apelle. *Idlescan*, pouvant être effectués aussi à l'aide de *nmap*, avec l'option *–sI*).

Listing 1. *Le code de la fonction ipid_checkentry*

```
static int ipid_checkentry(const char *tablename,
                const struct ipt_entry *e, void *targinfo,
                unsigned int targinfosize, unsigned int hook_mask) {

        if(strncmp(tablename, "mangle", 6) != 0) {
                printk(KERN_WARNING "IPID: Can only be called from the \
                       "mangle\" table");
                return 0;
        }
        if(targinfosize != IPT_ALIGN(sizeof(struct ipt_ipid_target_info))) {
                printk(KERN_WARNING "IPID: targinfosize %u != %Zu\n",
                                targinfosize, IPT_ALIGN(sizeof(struct ipt_
                       ipid_target_info)));
                return 0;
        }
        return 1;
}
```

- les options spécifiques pour l'extension,
- le masque des points d'accrochage (*hooks)* à partir desquels cette règle peut être appelée.

Le paramètre de base est la valeur 0, si la règle ne peut pas être acceptée. Dans le cas contraire, la valeur 1 est retournée.

Alors, nous pouvons vérifier si la règle est placée dans l'endroit approprié du tableau. Ces règles qui modifient le paquet doivent être ajoutées au tableau *mangle*. C'est notre cas, alors nous allons vérifier si le nom du tableau est correct.

On peut aussi vérifier si les options spécifiques pour le module sont bien configurées, par exemple si leur valeur est dans la plage valide. Si une règle n'est prévue que pour un protocole concret, par exemple UDP, nous pouvons le vérifier ici.

Les informations sur ce sujet se trouvent dans la structure `ipt_entry` transférée. Il est aussi recommandé de vérifier si la structure avec les paramètres spécifiques pour l'extension est bien ajustée du point de vue de l'espace mémoire occupée. Pour cela, nous disposons de la macro `IPT_ALIGN`.

Vu que notre module est assez simple, nous ne vérifions que l'ajustement de la mémoire et le nom du tableau auquel la règle est ajoutée. Le code de la fonction est présenté dans le Listing 1.

ipid_destroy

La fonction dans le champ `destroy` est appelée, quand la règle utilisant cette extension est supprimée de la mémoire. Cela permet d'allouer de l'espace pour les données de la règle dans la fonction `checkentry` et de les supprimer ici. Dans notre cas, cette fonction est vide, alors je la présenterai en entier :

```
static void ipid_destroy
(void *targinfo, unsigned
int targinfosize) {}
```

ipid_target

Enfin, la fonction réalisant directement la tâche. D'après la description préalable de la structure, cette fonction peut être `match` ou `target`. Nous modifions le paquet, et nous laissons l'association aux autres extensions, alors ce sera `target`. La fonction de type target reçoit quelques paramètres d'entrée, y compris le pointeur vers le tampon `skb`, le nom de l'interface d'entrée et de sortie pour le paquet (l'une peut rester vide) et les données spécifiques à la règle. Ces données proviennent de l'espace utilisateur et ont été préparées au moment de la préparation de la règle. Il peut y avoir des options spécifiques pour l'extension, ainsi que les données temporaires liées à la règle. Les options et le stockage des données seront décrites dans la suite de l'article, alors je néglige maintenant cette structure.

La structure `skb`, alors le tampon du socket sera présentée avec plus de détails dans l'encadré, mais ici je voudrais seulement mentionner que c'est une structure universelle du noyau du système Linux servant à faciliter les opérations sur les paquets aux couches spécifiques de la pile de protocoles. Elle permet le stockage dans un seul endroit de toutes les informations relatives au paquet, ce qui nous sera certainement utile.

Le but de cette fonction consiste (outre les opérations sur le paquet) à rendre le verdict, pour le cadre *IPTables,* sur ce que l'on veut faire ensuite avec le paquet donné. En cas de cibles simples, tels que ACCEPT ou DROP, le verdict est unique. Par contre, en cas de fonction de type `match`, le verdict se limite généralement à constater si le paquet a été associé ou pas (il se peut, dans les situations exceptionnelles, que le paquet soit refusé).

Dans notre cas, le paquet sera modifié, le verdict concernera alors le traitement ultérieur du paquet par *IPTables*. De plus, dans les situations exceptionnelles, telles que par exemple le manque de mémoire pour le traitement du paquet, nous le refuserons. La liste des verdicts pouvant être déterminés en paramètre de départ de la fonction dans les fonctions de type `target` n'est pas longue, mais suffisante :

- `NF_DROP` – stoppe le traitement du paquet (refuse le paquet) ,
- `NF_ACCEPT` – continue la traversée de paquet (accepte le paquet) ,
- `NF_STOLEN` – l'information pour *Netfilter* que le paquet avec `sk_buff` entier a été intercepté par le module,
- `NF_QUEUE` – ce verdict est utilisé, par exemple par le module ip_queue du paquet *Netfilter* afin de transférer les paquets pour

Listing 2. *L'exemple de l'implémentation ipid_target*

```
static unsigned int ipid_target(struct sk_buff **pskb,
                const struct net_device *in, const struct net_device *out,
                unsigned int hooknum, const void *targinfo, void *userinfo) {

        struct iphdr * iph;
        u_int16_t ipid_diffs[2];

        if (!skb_ip_make_writable(pskb, (*pskb)->len))
                return NF_DROP;

        iph = (*pskb)->nh.iph;
        ipid_diffs[0] = (iph->id)^0xffff;
        ipid_diffs[1] = iph->id = htons(counter++);
        iph->check = csum_fold(csum_partial((char *)ipid_diffs,
                                sizeof(ipid_diffs), iph->check^0xffff));

        (*pskb)->nfcache |= NFC_ALTERED;
        return IPT_CONTINUE;
}
```

leur traitement dans l'espace utilisateur,

- `NF_REPEAT` – redirige le paquet à repasser par les fonctions enregistrées pour le point d'accrochage donné.

Ce sont tous les verdicts de *Netfilter* que nous pouvons utiliser, mais puisque nous travaillons sur le niveau supérieur, c'est-à-dire *IPTables*, nous nous servirons donc du verdict `IPT_CONTINUE`, qui signifie la continuation du traitement ultérieur et utilisé par les extensions *IPTables*.

Comme nous en savons déjà un peu sur les paramètres de la fonction, nous pouvons passer à l'implémentation de son corps. Conformément aux principes adoptés, nous allons modifier la valeur du champ ID de l'en-tête du protocole IP. Au début, nous le ferons d'une manière très simple – à l'aide d'un compteur intérieur, post-incrémenté. Indépendamment de la méthode choisie, nous modifierons l'en-tête IP, et de cela, le tampon du socket (`struct sk_buff`). Nous devons informer le système de notre intention. Dans les noyaux 2.6, cette opération est faite à l'aide d'une fonction :

```
if (!skb_ip_make_writable
(pskb, (*pskb)->len))
 return NF_DROP;
```

Nous appelons cette fonction en lui transférant le tampon et sa longueur. C'est l'exemple d'une situation où nous pouvons décider d'imposer le refus du paquet. Nous ne sommes pas capables de réaliser le but admis, alors pour les raisons de sécurité, nous refusons le paquet. Dans les noyaux de la ligne 2.4, nous informons le système sur la modification projetée en copiant le tampon :

```
struct sk_buff *nskb =
skb_copy(*pskb, GFP_ATOMIC);
```

C'est ce que nous allons maintenant faire avec les données contenues dans le tampon et celles des en-têtes des protocoles dépend en particulier de la destination de l'extension et de notre invention. Dans la plupart des cas, ces opérations sont simples. Si rien n'est modifié, il suffit parfois de déterminer le verdict à l'aide d'une comparaison.

Quant à nous, nous effectuons une modification, ce qui entraîne la nécessité de mettre à jour les sommes de contrôle. Pour ce faire, le plus simple est d'utiliser les fonctions standard `sum_fold` et `csum_partial`. L'exemple approprié est présenté dans le Listing 2.

La conséquence suivante de la modification du paquet est la nécessité d'en informer le cadre d'application *Netfilter*. Nous pouvons le faire à l'aide du drapeau configuré dans le champ du tampon du socket :

```
(*pskb)->nfcache |= NFC_ALTERED;
```

Lors du codage de l'extension, le module doit souvent saisir des données supplémentaires dans le journal. Il serait bien que cette opération n'occupe pas toute la puissance de calcul de l'ordinateur. Le nombre de messages envoyés dans ces situations peut être limité à l'aide

Socket Buffer

Dans les paquets réseau, outre les données utilisateur, les en-têtes des protocoles sont envoyés. Chacune des couches, en commençant par la couche transport et descendant vers le bas, ajoute son en-tête.

Chaque couche de la pile de protocoles et chaque protocole sont gérés par fonctions différentes. Alors, pour ne pas copier inutilement les données, une seule grande structure (`struct sk_buff`), a été créée ; elle stocke les informations sur les en-têtes de tous les protocoles. Cette structure peut comprendre les données suivantes :

- le temps de réception du paquet pour les paquets arrivant,
- l'interface réseau via laquelle le paquet nous est parvenu,
- les sommes de contrôle
- le socket réseau, si le paquet est lié à un socket local
- autres données utiles lors du traitement du paquet par chaque couche de la pile de protocoles

Cette structure est aussi employée par *Netfilter* et transférée à la fonction `match` et `target`. La structure est liée aux fonctions servant à copier ou celles permettant son remplacement. La description plus détaillée des champs de la structure `sk_buff` est disponible dans le fichier d'en-tête *skbuff.h*.

OFFRES DE COUPLAGE

12 N°s + 6 N°s

147 €

99 €

12 N°s + 6 N°s

12 N°s + 6 N°s

12 N°s + 6 N°s

	OUI, je m'abonne et désire profiter des offres spéciales de couplage			
	Référence de l'offre :	**Prix :**	**Qté :**	**À partir du numéro :**
☐	12 N°s Linux+ DVD + 6 N°s PHP	99 €		
☐	12 N°s Linux+ DVD + 6 N°s Hakin9 comment se défendre ?	99 €		
☐	12 N°s Linux+ DVD + 6 N°s SDJ Extra	99 €		
☐	12 N°s Linux+ DVD + 6 N°s Programmation sous Linux	99 €		

Abonnez – vous !!!

- par courrier en nous renvoyant le bon ci-dessous
- par le web, sur notre site : *http://www.shop.software.com.pl/fr*
- par téléphone entre 10-17 au 00 48 22 887 13 44
- par fax au 00 48 22 887 10 11

Commande

Merci de remplir ce bon de commande et de nous le retourner par fax : 0048 22 887 10 10 ou par courrier : Software-Wydawnictwo Sp. z o. o., Piaskowa 3, 01-067 Varsovie, Pologne ; E-mail : *abonnement@software.com.pl*

Prénom Nom ..... **Entreprise** .....

Adresse .....

Code postal ..... **Ville** .....

Téléphone ..... **Fax** .....

E-mail .....

Je règle par :

☐ **Chèque**

☐ **Carte bancaire n° CB** |_|_|_|_| |_|_|_|_| |_|_|_|_| |_|_|_|_| **expire le** |_|_|_|_| **date et signature obligatoires**

☐ **type de carte** ..... **code CVV/CVC** |_|_|_|

☐ **Virement bancaire :**

Nom banque : Société Générale Chasse/Rhône

banque	guichet	numéro de compte	clé Rib
30003	01353	00028010183	90

IBAN : FR76 30003 01353 00028010183 90

Adresse Swift (Code BIC) : SOGEFRPP

de la fonction `net_ratelimit`. La journalisation des messages doit se présenter ainsi :

```
if(net_ratelimit())
printk("message...\n");
```

Au début, ces informations doivent être suffisantes. Un exemple d'une telle implémentation est présentée dans le Listing 2.

Aux fonctions que nous venons d'implémenter, il faut aussi ajouter une structure remplie, l'enregistrer et ajouter au début quelques directives des activations et l'extension est prête. Le fichier entier est disponible sur le CD joint au magazine.

Outil de l'espace utilisateur

Une fois l'extension préparée, il est nécessaire qu'il soit possible d'ajouter les règles qui l'utilisent. Les règles seront ajoutées au moyen d'outil standard *iptables*. Cet outil a aussi une structure modulaire et il suffit de préparer une bibliothèque appropriée avec la gestion de notre module.

Une telle bibliothèque doit contenir, avant tout, la fonction `_init` à partir de laquelle la fonction `register_match` ou `register_target` sera appelée, en fonction du module géré. C'est la même situation que pendant le chargement du module d'extension. Dans notre cas, ce sera `register_target`. C'est la structure qui lui est passée en argument. Les champs qui nécessitent d'être commentés seront expliqués sur l'exemple de notre module :

- `next` – utilisé pour la création de la liste des targets, par exemple lors du listing des règles. La valeur initiale doit être NULL,
- `name` – le nom doit être conforme au nom de la bibliothèque, comme par exemple IPID pour libipt_IPID.so,
- `version` – la version de l'outil *IPTables*,
- `help` – le pointeur vers la fonction affichant la description de la syntaxe pour l'extension,

```
initialisation done
> iptables -t mangle -A FORWARD -s 192.168.0.2 -d 192.168.1.2 -j IPID
> gen_ip IF=eth0 192.168.0.2 192.168.1.2 0 TCP 1060 80 SYN
rcv:eth0
hook:NF_IP_PRE_ROUTING ip_conntrack NF_ACCEPT {IPv4 192.168.0.2 192.168.1.2 0 6 1060 80 SYN}
hook:NF_IP_PRE_ROUTING iptable_raw NF_ACCEPT {IPv4 192.168.0.2 192.168.1.2 0 6 1060 80 SYN}
hook:NF_IP_PRE_ROUTING ip_conntrack NF_ACCEPT {IPv4 192.168.0.2 192.168.1.2 0 6 1060 80 SYN}
hook:NF_IP_PRE_ROUTING iptable_mangle NF_ACCEPT {IPv4 192.168.0.2 192.168.1.2 0 6 1060 80 SYN}
hook:NF_IP_PRE_ROUTING iptable_nat NF_ACCEPT {IPv4 192.168.0.2 192.168.1.2 0 6 1060 80 SYN}
INFO:IPID: Id changed 0 -> 0
hook:NF_IP_FORWARD iptable_mangle NF_ACCEPT {IPv4 192.168.0.2 192.168.1.2 0 6 1060 80 SYN}
hook:NF_IP_FORWARD iptable_filter NF_ACCEPT {IPv4 192.168.0.2 192.168.1.2 0 6 1060 80 SYN}
hook:NF_IP_POST_ROUTING iptable_mangle NF_ACCEPT {IPv4 192.168.0.2 192.168.1.2 0 6 1060 80 SYN}
hook:NF_IP_POST_ROUTING iptable_nat NF_ACCEPT {IPv4 192.168.0.2 192.168.1.2 0 6 1060 80 SYN}
hook:NF_IP_POST_ROUTING ip_conntrack NF_ACCEPT {IPv4 192.168.0.2 192.168.1.2 0 6 1060 80 SYN}
send:eth1 {IPv4 192.168.0.2 192.168.1.2 0 6 1060 80 SYN}
> gen_ip IF=eth0 192.168.0.2 192.168.1.2 0 TCP 1060 80 SYN
rcv:eth0
hook:NF_IP_PRE_ROUTING ip_conntrack NF_ACCEPT {IPv4 192.168.0.2 192.168.1.2 0 6 1060 80 SYN}
hook:NF_IP_PRE_ROUTING iptable_raw NF_ACCEPT {IPv4 192.168.0.2 192.168.1.2 0 6 1060 80 SYN}
hook:NF_IP_PRE_ROUTING ip_conntrack NF_ACCEPT {IPv4 192.168.0.2 192.168.1.2 0 6 1060 80 SYN}
hook:NF_IP_PRE_ROUTING iptable_mangle NF_ACCEPT {IPv4 192.168.0.2 192.168.1.2 0 6 1060 80 SYN}
hook:NF_IP_PRE_ROUTING iptable_nat NF_ACCEPT {IPv4 192.168.0.2 192.168.1.2 0 6 1060 80 SYN}
INFO:IPID: Id changed 0 -> 1
hook:NF_IP_FORWARD iptable_mangle NF_ACCEPT {IPv4 192.168.0.2 192.168.1.2 0 6 1060 80 SYN}
hook:NF_IP_FORWARD iptable_filter NF_ACCEPT {IPv4 192.168.0.2 192.168.1.2 0 6 1060 80 SYN}
hook:NF_IP_POST_ROUTING iptable_mangle NF_ACCEPT {IPv4 192.168.0.2 192.168.1.2 0 6 1060 80 SYN}
hook:NF_IP_POST_ROUTING iptable_nat NF_ACCEPT {IPv4 192.168.0.2 192.168.1.2 0 6 1060 80 SYN}
hook:NF_IP_POST_ROUTING ip_conntrack NF_ACCEPT {IPv4 192.168.0.2 192.168.1.2 0 6 1060 80 SYN}
send:eth1 {IPv4 192.168.0.2 192.168.1.2 0 6 1060 80 SYN}
>
```

Figure 1. *L'exemple d'une session avec l'émulateur nfsim*

- `init` – ici, il est possible de mettre le pointeur vers la fonction exécutant les opérations d'initialisation auxiliaires. Cette fonction sera appelée avant l'appel de `parse`,
- `parse` – comme son nom l'indique, cette fonction est appelée pour la gestion des paramètres non distingués par *IPTables*. Si ce sont réellement les options attendues par l'extension, la fonction doit retourner une valeur non nulle. Si l'un des paramètres d'entrée de la fonction est la variable. L'un des paramètres d'entrée de la fonction est la variable `invert`, configuré à *true*, si avant la spécification de l'option, un *!* a été présent,
- `final_check` – la fonction appelée après l'analyse de l'option, permet d'appeler par exemple `exit_error()`, si les options s'excluent mutuellement ou aucune des options obligatoires n'a été donnée,
- `print` – la fonction utilisée pendant l'affichage des règles, doit imprimer les informations non standard pour les règles. Utilisée pendant l'impression des règles à l'aide de la commande *iptables –L*,
- `save` – le pointeur vers la fonction utilisée pour reproduire les options utilisées pour créer la règle,
- `extra_opts` – c'est le pointeur vers la tables de structures – la liste des options supplémentaires acceptées par l'extension doit être terminée par la structure remplie de NULL. Elle est unie avec la liste des arguments standard et transférée à `getopt_long`.

Pour les extensions de type *match*, la structure est similaire. Pour que notre module soit correctement géré et testé, il suffit de remplir au début cette structure. Les fonctions déclarées seront laissées pratiquement sans corps, et dans la liste des options, nous définirons un seul enregistrement avec les valeurs NULL. Le fichier complet de la bibliothèque est disponible sur le CD joint au magazine. Nous mettons la bibliothèque compilée dans l'endroit accessible par l'outil *IPTables* et nous pouvons passer aux tests de notre module.

Les Tests

La stabilité du module peut influencer la stabilité du noyau dans lequel nous le chargerons, alors il sera mieux d'effectuer les tests dans un environnement séparé. Cette possibilité est offerte par *nfsim*. Cet outil peut être aussi téléchargé à partir du site *www.netfilter.org*. Comme son nom l'indique, c'est un simulateur de *Netfilter*. Il permet de tester les extensions.

Après le démarrage, nous disposons d'une console dans laquelle nous pouvons entrer les règles

```
14:52:44.943074 150.254.173.130 > 212.77.100.101: icmp: echo request (ttl 128, id 14288, len 60)
14:52:44.943097 150.254.173.130 > 212.77.100.101: icmp: echo request (ttl 127, id 29575, len 60)
14:52:45.955631 150.254.173.130 > 212.77.100.101: icmp: echo request (ttl 128, id 14313, len 60)
14:52:45.955648 150.254.173.130 > 212.77.100.101: icmp: echo request (ttl 127, id 29576, len 60)
14:52:46.963816 150.254.173.130 > 212.77.100.101: icmp: echo request (ttl 128, id 14338, len 60)
14:52:46.963832 150.254.173.130 > 212.77.100.101: icmp: echo request (ttl 127, id 29577, len 60)
14:52:47.972001 150.254.173.130 > 212.77.100.101: icmp: echo request (ttl 128, id 14363, len 60)
14:52:47.972018 150.254.173.130 > 212.77.100.101: icmp: echo request (ttl 127, id 29578, len 60)
```

Figure 2. *La valeur ID pour les paquets modifiés par le pare-feu*

à l'aide d'*IPTables,* générer les paquets et les sessions TCP. À la suite de ces opérations, les informations sur la trace du paquet à travers la pile de protocole sont affichées à l'écran. Nous pouvons voir aussi nos propres messages qui peuvent être affichés sur la console à l'aide de la fonction `printk`.

Nous devons placer notre module dans le répertoire *netfilter/ipv4*. Grâce à cela, il sera chargé automatiquement lors du démarrage de l'environnement de l'émulateur. Une courte session avec l'émulateur permet de repérer très vite les erreurs critiques, de savoir si notre module fonctionne correctement ou les sommes de contrôle sont correctes, etc. L'aide disponible dans l'outil (accessible après un clic sur la commande `help`) est suffisante pour apprendre la gestion de l'émulateur, il est donc inutile de la répéter ici.

L'exemple d'une session avec l'émulateur a été présenté sur la Figure 1. On voit que le module s'est chargé sans échec, la règle a été ajoutée. Ensuite, deux paquets ont été envoyés. Dans le premier, IPID a été changé de 0 en 0, et dans le deuxième de 0 en 1. Cela est dû au fait que dans le paquet généré, IPID est toujours mis à 0, par contre contre le module est doté d'un compteur interne qui augmente IPID de 1 à chaque appel. Il faut aussi faire attention à l'endroit dans lequel notre module est appelé. Nous avons ajouté notre règle à la table *mangle* de la chaîne FORWARD, c'est pourquoi le message apparaît avant la fin du traitement de cette chaîne.

Après les tests de l'extension dans l'émulateur, nous pouvons tenter de charger le module dans la mémoire du noyau. J'ai chargé la version présentée sur le CD fourni avec la revue, sur un pare-feu protégeant un réseau avec quelques machines. Tout fonctionne parfaitement, on peut donc considérer que la version alpha a été testée au moyen d'une méthode métier.

La Figure 2 affiche quatre paquets *icmp echo-request* consultés à l'aide de *tcpdump*. Chacun d'eux est vu deux fois : avant et après le passage à travers le pare-feu. On voit que le TTL diminue de 1 (ce qui est naturel) et que l'ID change, ce qui est dû au fonctionnement de notre module. L'ID original change un peu plus vite, par contre celui remplacé seulement de 1.

Possibilité de choisir

Dans certains cas, mais pas tous, le champ ID ainsi changé peut s'avérer suffisant pour dissimuler notre système. De plus, si nous voulons qu'il soit changé autrement, il faut modifier le code du module et le recompiler.

Et si dans une règle la modification est effectuée d'une autre manière comme dans une autre ? Dans ce cas, ce sont les options transmises de l'espace utilisateur au module à l'aide de l'outil *IPTables* qui viennent à notre aide. Il faut programmer cela, ce qui est très facile.

Nous saisissons les options pour l'extension de la bibliothèque pour l'outil *IPTables* sous forme d'une liste de structures (de manière standard pour *getopt*) dans le fichier contenant le fichier avec le code source gérant notre extension. Par exemple :

```
static struct option opts[] = {
{"random", 0, 0, 'r'},
{"incremental", 1, 0, 'i'},{0}
};
```

Chacun qui utilisait la fonction `getopt _ long`, connaît sans doute la signification des champs de cette structure. Si vous ne l'avez jamais fait, les explications sont disponibles sur la page `man` de cette fonction.

Les options ainsi ajoutées doivent être maintenant gérées dans le corps de la fonction `parse` de la bibliothèque de l'extension. Nous allons le faire de manière standard pour `getopt _ long`.

Listing 3. *L'exemple de l'implémentation de la gestion des options de l'extension dans la bibliothèque de l'outil IPTables*

```
struct ipt_ipid_target_info {
        u_int32_t mode;
        u_int32_t step;
};


static int parse(int c, char **argv, int invert, unsigned int *flags,
                const struct ipt_entry *entry, struct ipt_entry_target
                        **target) {
        struct ipt_ipid_target_info * ipid_info = (struct ipt_ipid_target_
                        info *)(*target)->data;

        ...

        *flags = ipid_info->mode;
        return 1;
}

static void final_check(unsigned int flags) {
        if(!(flags&IPID_MODE_RANDOM) && !(flags&IPID_MODE_INCREMENTAL))
                exit_error(PARAMETER_PROBLEM, "You have to chose an
                        algorithm\n");
        ...
}
```

Listing 4. *L'exemple de l'implémentation de la gestion des options dans le code du module du noyau*

```
static unsigned int ipid_target(struct sk_buff **pskb,
        ...
        struct ipt_ipid_target_info * ipid_info = (struct ipt_ipid_target_
                info *)targinfo;
        ...
        if(ipid_info->mode&IPID_MODE_INCREMENTAL) {
                ipid_diffs[1] = iph->id = htons(counter);
                counter += ipid_info->step;
        }
        else if(ipid_info->mode&IPID_MODE_RANDOM) {
                get_random_bytes(&(ipid_diffs[1]), sizeof(u_int16_t));
                ipid_diffs[1] = iph->id = htons(ipid_diffs[1]);
        }
        ...
```

Il reste encore la question de la structure qui sera transférée au noyau. Alors, c'est la même structure dont la validité de la taille nous avons vérifié dans la fonction `ipid_checkentry`. Pour chaque extension, nous définissons la structure à l'aide de laquelle nous transférerons les options et qui existera en relation avec la règle. Chaque fois que la fonction `match` ou `target` d'une extension donnée sera appelée, cette même structure y sera transférée.

Le cycle de vie de cette structure commence justement ici : dans l'espace utilisateur, au moment de l'ajout de la règle. Pour accéder au niveau de la fonction `parse`, il faut l'extraire de la structure `ipt_entry_target` de la manière présentée dans le Listing 3. La définition de la structure est aussi disponible dans ce listing.

Le paramètre d'entrée suivant de la fonction `parse` qui nous intéresse est la variable `flags`. Elle permet de transférer les informations parmi les appels successifs de `parse`, ainsi que de transférer les informations à `final_check`. Quant à nous, nous exploiterons cette variable à transmettre à `final_check` les informations sur les paramètres transférés par l'utilisateur. Nous voulons que un seul algorithme de changement du champ ID du datagramme IP soit choisi. L'exemple est présenté dans le Listing 3. De plus, il vaut la peine de programmer aussi les autres fonctions : `help`, `print`, `save`, de façon à pouvoir profiter pleinement de l'option, mais c'est au lecteur de décider.

Une bonne pratique est de vérifier aussi dans la fonction `checkentry` du module, si les options saisies sont correctes. Mais nous n'allons pas nous en occuper ici.

Si nous voulons utiliser l'option au niveau du module dans le noyau, il suffit de se référer à la structure transférée en tant que paramètre d'entrée à la fonction `checkentry` et `target` ou `match`. Nous pouvons le faire de la manière similaire à celle dans le cas de l'espace utilisateur.

L'exemple est présenté dans le Listing 4 contenant les fragments modifiés de la fonction `ipid_target`. Après la recompilation du module et de la bibliothèque, nous aurons la possibilité de spécifier la façon de changer le champ ID au niveau de la ligne de commande.

Stockage des données

Dans notre cas, le compteur par rapport auquel nous modifions la valeur du champ ID, est stocké globalement. L'un des défauts de cette solution est le fait que la valeur ainsi stockée est commune à toutes les règles exploitant ce module. Mais il se peut que nous voulions que la valeur du compteur change séparément et indépendamment pour chaque règle, par exemple pour un réseau, nous souhaitons des changements aléatoires, et pour l'autre – des changements incrémentiels. Comment faire ?

D'après le point précédent, à chaque appel de la fonction `ipid_target`, la structure `targinfo` dont la définition dépend de l'utilisateur est transférée à cette fonction. Dans cette structure, nous pouvons ajouter un champ successif qui représentera notre compteur. Une telle structure est créée pour chaque règle séparément, alors chaque règle aura un compteur à part. La mise à jour modifiée du compteur dans le code de la fonction `ipid_target` pourrait se présenter comme suit :

```
if(ipid_info->
mode&IPID_MODE_INCREMENTAL) {
ipid_diffs[1] = iph->id =
htons(ipid_info->lastval);
ipid_info->lastval += ipid_info->step;
}
```

Mais en résolvant ce problème, nous nous heurtons à un autre. Vu que dans les systèmes SMP, pour chaque processeur une copie séparée de la table est maintenue, nous pouvons avoir le problème avec la présence de deux copies du compteur. Il peut donc arriver que la même valeur soit présente plusieurs fois. Il faut donc prendre soin à ce qu'il n'existe qu'une seule copie du compteur, indépendamment du nombre de processeurs.

L'une des façons plus simples d'atteindre ce but consiste à ajouter un champ supplémentaire dans la structure `targinfo`. Ce sera le pointeur à la copie principale de cette structure. Ce pointeur nous permettra de nous référer aux champs contenant les valeurs modifiées. Pour introduire ces concepts dans le code, seules de petites modifications sont nécessaires : outre l'ajout d'un champ à la structure `targinfo`, il suffit de saisir une inscription appropriée dans la fonction `ipid_checkentry` :

```
ipid_info->master = ipid_info;
```

et partout où dans la fonction `ipid_target` nous nous référons à un champ avec la valeur modifiée, de changer :

```
ipid_info->lastval
```

devient:

```
ipid_info->master->lastval
```

Outre la copie des tables, il nous reste encore un problème à résoudre. Le code d'un tel module doit être réentrant. Il peut arriver que lors de la gestion d'un paquet, une interruption demandant la gestion simultanée d'un autre paquet, peut avoir lieu. Toujours quand un accès simultané aux données a lieu, il faut gérer cet accès de façon à garder la cohérence des données. Il est facile de nous imaginer la situation que pour deux paquets, nous déterminons une nouvelle valeur du champ ID du datagramme IP et nous incrémentons la valeur du compteur. Cela peut mener à l'affectation de la même valeur à deux paquets ou à d'autres incohérences.

Une solution simple à ce problème est d'utiliser les blocages de type `spinlock _ t`. C'est un mécanisme du noyau du système d'exploitation supportant la gestion de la colatéralité. Pour cela, il suffit de déclarer l'utilisation de ce verrou, par exemple:

```
static spinlock_t ipid_lock =
SPIN_LOCK_UNLOCKED;
```

Ensuite, dans tous les endroits où nous nous référons à une valeur partagée, nous pouvons mettre une demande de verrouillage d'accès, à l'aide d'une des fonctions spécialement conçues à cet effet :

```
spin_lock_bh(&ipid_lock);
```

Et une demande de déverrouillage après l'exécution de l'opération :

```
spin_unlock_bh(&ipid_lock);
```

Il est aussi important d'éviter la situation quand l'un de threads demande le verrouillage tandis qu'il ne sera pas possible d'ôter le verrouillage fait par un autre thread. Cette situation pourra mener au plantage du système.

Il paraît que pour nos besoins, le problème de stockage des données a été résolu. Pourtant, il n'est pas facile de s'imaginer que cela ne suffit pas. Si nous avions voulu, par exemple, stocker les compteurs à part pour chaque flux IP défini comme paire (source, cible), nous aurions dû créer des structures plus sophistiquées. À moins que vous ayez décidé d'introduire une règle séparée pour chaque paire. Mais cette solution ne paraît pas optimale.

Dans ces situations, nous pouvons envisager la création de propres cache d'objets, le maintien pour les besoins du module des structures plus avancées (tables de hashage, arborescences, listes), mais c'est le sujet d'un autre article.

Ce n'est que le début

Les informations présentées et un peu de temps suffisent pour écrire un module simple tout à fait opérationnel. Cela doit vous encourager à concevoir des modules plus complexes.

Le paquet *IPTables* comprend plusieurs modules non présentés dans l'article, qui utilisent les fonctionnalités du cadre plus avancées, telles que la possibilité de suivre les connexions, d'écrire les extensions supportant le filtrage des protocoles utilisant plusieurs connexions parallèles ou l'outil avancé NAT.

Le module ipt_IPID, qui nous venons de construire, peut servir d'un élément de pare-feu dissimulant la vraie identité des systèmes d'exploitation protégés. Certainement, il ne suffit pas pour réaliser complètement la tâche, de même que le groupe de fonctions *Fingerprint Scrambling* d'une solution pare-feu commerciale. Mais il peut être efficace pour empêcher l'analyse des autres noeuds du réseau à l'aide de nos machines.

Mais il ne faut pas oublier que le champ ID est important pour les datagrammes IP fragmentés, alors il n'est pas possible de changer facilement sa valeur, si le paquet est un fragment d'un datagramme IP. Il suffit de s'assurer que la règle ne soit pas être appliquée aux paquets fragmentés. On peut le réaliser de plusieurs manière, par exemple en écrivant une autre extension, cette fois-ci de type *match*.

Considérez la création d'ipt_ISN comme une aventure intellectuelle et tentez de résoudre les problèmes supplémentaires qui en résultent, par exemple comment maintenir l'information sur l'état de la connexion TCP, comment assurer le changement bilatéral des numéros de séquence (dans un sens, on change SEQ, de l'autre ACK). Mais ces problèmes sont certainement à résoudre.

Je peux vous avouer que j'ai conçu un tel module pendant l'écriture de cet article. ●

Sur Internet

- *http://www.netfilter.org/documentation/HOWTO//netfilter-hacking-HOWTO.txt* – les informations sur l'implémentation interne du paquet *Netfilter*,
- *ftp://ftp.rfc-editor.org/in-notes/rfc791.txt* – le document RFC décrivant le protocole IP,
- *http://www.insecure.org/nmap/nmap-fingerprinting-article.html* – l'article sur la détection distante du système d'exploitation au moyen de l'analyse des protocoles réseau.

À propos de l'auteur

L'auteur est employé à l'Équipe de Sécurité du Centre des Super-ordinateurs et des Réseaux de Poznan. Il s'occupe des questions de sécurité informatique, participe aux tests de pénétration et aux audits effectués par l'Équipe de Sécurité. Pour plus d'informations sur les travaux de l'Équipe, référez-vous aux pages : *http://security.psnc.pl/*