Linux netfilter Hacking HOWTO

Rusty Russell et Harald Welte, mailing list `netfilter@lists.samba.org`
Traduit par Fabrice MARIE*fabrice@netfilter.org*
v1.0.1 Samedi 1er Juillet 18:24:41 EST 2000, traduction du vendredi 13 avril 2001

Ce document décrit l'architecture de netfilter pour Linux, comment développer/modifier des modules. Ce document décrit aussi les modules principaux sur lesquels netfilter s'appuie le plus comme les modules "packet filtering" (filtrage de paquets), "connection tracking" (suivi de connexions), et "Network Address Translation".

Table des matières

1 Introduction

Salut,

Ce document est un périple ; quelques parties vous aiderons bien, alors que dans d'autres vous serez presque seuls. Le meilleur conseil que je puisse vous donner est d'attraper une grande bonne tasse de café, ou de chocolat chaud, de vous prendre une chaise confortable, et de lire le contenu de ce document avant de vous aventurer tout seul dans le dangereux monde du "network hacking".

Pour comprendre mieux l'usage de l'infrastructure au dessus de netfilter, je vous recommande la lecture du "Packet Filtering HOWTO" et du "NAT HOWTO". Pour plus d'informations sur la programmation kernel, je vous suggère le "Rusty's Unreliable Guide to Kernel Hacking" et le "Rusty's Ureliable Guide to Kernel Locking".

1.1 Qu'est ce qu'est Netfilter ?

Premièrement, Netfilter est un canevas pour modifier les paquets réseaux, en dehors de l'interface de sockets Berkeley. Il est composé de 4 parties. D'abord, chaque protocole définit des "hooks" (accroches) (IPv4 en définit 5) qui sont des points bien déterminés dans le trajet du paquet dans la pile de protocole. A chacun de ces points, le protocole va appeler le canevas de Netfilter et lui passer le paquet et le numéro de "hook".

Deuxièmement, des parties du kernel peuvent s'enregistrer pour écouter à différents "hooks" pour chacun des protocoles. Donc quand un paquet est passé au canevas Netfilter, ce dernier va vérifier si quelqu'un s'est enregistré pour ce protocole et ce "hook". Si c'est le cas, ils vont tous avoir une chance d'examiner (et aussi modifier) le paquet dans l'ordre, et ensuite de : se débarrasser du paquet (`NF_DROP`), de l'autoriser à passer (`NF_ACCEPT`), de dire à netfilter d'oublier le paquet (`NF_STOLEN`), ou de dire à Netfilter de queuter le paquet en userspace (`NF_QUEUE`).

Troisièmement, les paquets qui ont été queues, sont récupérés (par le driver ip_queue) pour les envoyer en userspace. Ces paquets sont gérés asynchronement.

La partie finale consiste en commentaires cools dans le code et de la documentation. C'est très important dans tous projets expérimentaux. Le moto de netfilter est (volé sans aucune honte de Cort Dougan) :

```
``So... how is this better than KDE?''  (``Alors ... en quoi c'est mieux que KDE ?'')
```

(Ce moto a presque réussi sont chemin : 'Whip me, beat me, make me use ipchains' ('Fouettez moi, battez moi, faites moi utiliser ipchains'))

Additionnellement à ce canevas nu, des modules divers ont été écrits, pour fournir les mêmes fonctionnalités que les anciens (pre-netfilter) kernels, et en particulier un système NAT extensible, et un système de filtrage de paquets extensible lui aussi (iptables).

1.2 Qu'est ce qui n'allait pas avec ce qu'on avait dans les 2.0 et les 2.2 ?

1. Pas d'infrastructure établie pour passer des paquets au userspace :
   - La programmation kernel est difficile.
   - La programmation kernel doit être faite en C/C++
   - Les polices de filtrage dynamiques n'ont pas de place justifiée dans le kernel.
   - 2.2 a introduit la possibilité de copier des paquets vers le userspace via netlink, mais re-injecter des paquets était lent, et sujet aux vérifications de 'sanité'. ex : la re-injection d'un paquet se disant provenir d'une interface existante n'était pas possible.
2. Le proxy transparent était bordellique :
   - On vérifie **tous** les paquets pour voir si il y une socket attachée à cette adresse.
   - root a le droit d'attacher une socket à une adresse étrangère.
   - Impossible de rediriger les paquets générés localement.
   - REDIRECT ne gère pas les réponses UDP : rediriger les paquets UDP de named vers le port 1153 ne marche pas parce que quelques clients n'aiment pas que la réponse arrive avec un port source diffèrent de 53.
   - REDIRECT ne se synchronise pas avec l'allocation de port TCP/UDP : un utilisateur peut avoir un port masqué par une règle REDIRECT.

  - A été cassé au moins 2 fois pendant la série des 2.1.
  - Le code est extrêmement intrusif. Considérez les stats sur le nombre de '#ifdef CONFIG_IP_TRANSPARENT_PROXY' dans le 2.2.1 : 34 occurrences dans 11 fichiers. Comparez cela à 'CONFIG_IP_FIREWALL', qui a 10 occurrences dans 5 fichiers.
3. Créer une règle de filtrage de paquets indépendamment de l'adresse de l'interface n'est pas possible :
   - Obligation de savoir l'adresse de l'interface locale pour distinguer les paquets générés localement ou les paquet terminant en local, des paquets qui traversent seulement.
   - Même cela n'est pas suffisant dans les cas de redirection ou de masquerading.
   - La chaîne FORWARD seule a les informations sur le paquet qui sortent, ce qui veut dire que vous avez à deviner d'où le paquet est venu, ou vous avez besoin d'une bonne connaissance de la topologie du réseau que vous utilisez.
4. Le masquerading est cloué sur le filtrage de paquets :

   Les interactions entre le masquerading et le filtrage de paquets rendent le firewalling complexe.
   - Lors du filtrage dans INPUT, les paquets semblent être destinés à la machine elle même.
   - Lors du filtrage dans FORWARD, les paquets démasqueradés ne sont pas vu du tout.
   - Lors du filtrage dans OUTPUT, les paquets semblent venir de la machine locale
5. La manipulation du TOS, redirection, "ICMP unreachable" et le marquage de paquets (qui peu affecter le port-forwarding, le routage, et le QoS) sont cloués sur le code de filtrage de paquets aussi.
6. Le code d'ipchains n'est ni modulaire, ni extensible (par exemple : le filtrage basé sur la adresse MAC, le filtrage des options, etc).
7. Le manque d'infrastructure suffisante a mené à la profusion de différentes techniques :
   - Masquerading, plus des modules pour certains protocoles.
   - La NAT rapide par le code de routage (n'a pas de module qui gère certains protocoles).
   - Le port-forwarding, la redirection, et le auto-forwarding
8. Incompatibilité entre CONFIG_NET_FASTROUTE et le filtrage de paquets :
   - Les paquets qui ne font que nous traverser doivent passer par trois chaînes de toutes façons.
   - Impossible de dire si ces chaînes peuvent être évitées.
9. L'inspection de paquets détruits est impossible à cause d'une protection de routage (par exemple : la vérification d'adresse source).
10. Impossible de lire automatiquement les compteurs sur les règle de filtrage de paquets.
11. CONFIG_IP_ALWAYS_DEFRAG est une option à la compilation seulement, rendant la vie difficile pour les distributions qui veulent un kernel général.

1.3 Qui êtes vous ?

Je suis le seul à être assez fou pour faire ça. En tant que co-auteur de ipchains et maintenant mainteneur du Linux Kernel IP Firewall, je vois beaucoup des problèmes que les gens ont avec le système actuel, en même temps d'avoir des informations sur ce qu'ils tentent de faire.

1.4 Pourquoi ça plante ?

Ah bon ?! Bah fallait voir comment c'était la semaine **dernière** !

Parce que je ne suis pas un aussi bon programmeur que ce que tout le monde espère, et je n'ai certainement pas testé tous les scénarios, parce que je manque de temps/équipement et/ou d'inspiration. J'ai effectivement une "test-suite", que je vous encourage à améliorer.

2 Ou puis-je trouver la dernière version ?

Il y a un serveur CVS sur netfilter.org qui contient les derniers HOWTOs, les outils userspace, et la “test suite”. Pour naviguer tranquillement, vous pouvez utiliser l’ *l’interface web* `http://cvs.netfilter.org/` .

Pour downloader les derniers sources, vous pouvez faire ce qui suit :

1. Loguez vous sur le serveur CVS de netfilter en anonymous :

   ```
   # cvs -d :pserver:cvs@pserver.netfilter.org:/cvspublic login
   ```

2. Quand il vous demande un password, entrez ‘cvs’.
3. Pompez (“check-out”) le code en utilisant :

   ```
   # cvs -d :pserver:cvs@pserver.netfilter.org:/cvspublic co netfilter/userspace
   ```

4. Pour mettre à jour vers la dernière version, utilisez

   ```
   cvs update -d -P
   ```

3 Architecture de Netfilter

Netfilter est pratiquement une série de “hooks” (accroches) à quelques endroits dans la pile de protocoles (pour le moment, IPv4, IPv6, DECnet). Le shema de traversé ressemble (idéalement) à cela :

```
Un paquet qui traverse le système Netfilter:

   --->[1]--->[ROUTE]--->[3]--->[4]--->
                 |            ^
                 |            |
                 |         [ROUTE]
                 v            |
                [2]          [5]
                 |            ^
                 |            |
                 v            |
```

Sur la gauche, c’est la ou les paquets arrivent : en ayant passé un simple test de sanité (par exemple : pas coupé, IP checksum est OK, et que ce n’est pas un paquet reçu en promiscuous), ils sont passés au “hook” NF_IP_PRE_ROUTING [1] du canevas de netfilter.

Ensuite, il entrent dans le code de routage, qui décide si oui ou non le paquet est destiné à une autre interface, ou pour un processus en local. Le code de routage peut éventuellement détruire le paquet si il n’est pas routable.

Si il est destine pour la machine elle même, le canevas netfilter est appelé pour le hook NF_IP_LOCAL_IN [2], avant d’être passé au processus, si il y en a un.

Si à la place le paquet est destiné pour une autre interface, le canevas netfilter est appelé pour le hook NF_IP_FORWARD [3].

Le paquet passe ensuite le dernier hook de netfilter, le NF_POST_ROUTING [4], avant d’être envoyé sur le cable.

Le hook NF_IP_LOCAL_OUT [5] est appelé pour les paquets qui sont générés localement. Ici vous pouvez voir que le routage ce fait après que ce hook est appelé : en fait, le code de routage est appelé avant (pour trouver l’adresse source IP, et quelques options IP) : si vous voulez modifier le routage, vous devez modifier le champs ‘skb->dst’ par vous même, comme dans le code de NAT.

3.1 La base de Netfilter

Maintenant nous avons un exemple de netfilter pour IPv4, vous pouvez voir quand chacun des hooks est activé. C'est l'essence même de netfilter.

Les modules kernel peuvent s'enregistrer pour écouter à chacun de ces hooks. Un module qui enregistre une fonction doit spécifier la priorité de cette fonctions à l'intérieur du hook. De telle sorte que quand ce hook est appelé par netfilter, le code réseaux central appellera les fonctions du hook dans l'ordre des priorités. Le module peut dire à netfilter de faire l'une de ces 5 choses :

1. NF_ACCEPT : le paquet peut continuer sa traversé comme normal.
2. NF_DROP : détruit le paquet, la traversé est arrêtée net.
3. NF_STOLEN : J'ai pris possession du paquet, arrête la traversée.
4. NF_QUEUE : queute le paquet (habituellement pour manipulation future en userspace).
5. NF_REPEAT : Appelle ce hook encore une fois.

Les autres parties de Netfilter (gérer les paquets queutés, et les commentaires cools) seront couverts dans la section kernel un peu plus tard.

Basé sur cette fondation, nous pouvons construire des manipulations de paquets relativement complexes, comme démontré dans les deux sections suivantes.

3.2 Sélection de Paquets

Un système de sélection de paquets appelé 'iptables' a été construit par dessus le canevas netfilter. C'est le descendant direct de 'ipchains' (qui descendait de 'ipfwadm', qui venait lui même de ipfw de BSD) mais amélioré car plus extensible. Les modules kernels peuvent demander à enregistrer une table nouvelle, et demander à ce qu'un paquet traverse un table donnée. Cette méthode de sélection de paquets est utilisée pour filtrer les paquets (la table 'filter'), pour NAT (la table 'nat') et pour modifier des paquets avant routage (la table 'mangle').

Les hooks qui sont enregistrés avec netfilter sont comme suit (avec les fonctions de chaque hooks dans l'ordre ou ils sont appelles) :

```
--->PRE------>[ROUTE]--->FWD---------->POST------>
    Conntrack    |       Mangle   ^    Mangle
    Mangle       |       Filter   |    NAT (Src)
    NAT (Dst)    |                |    Conntrack
    (QDisc)      |             [ROUTE]
                 v                |
                 IN Filter       OUT Conntrack
                 |  Conntrack     ^  Mangle
                 |  Mangle        |  NAT (Dst)
                 v                |  Filter
```

3.2.1 Filtrage de paquets

Cette table, 'filter', ne devrait jamais modifier les paquets : seulement les filtrer.

Un des avantages des filtres de iptables comparés à ceux de ipchains est que c'est petit et rapide, et que ça s'accroche dans netfilter aux points NF_IP_LOCAL_IN, NF_IP_FORWARD, et NF_IP_LOCAL_OUT. Ça veut dire que pour un paquet donné, il n'y a qu'une et un seule place pour le filtrer. Cela rend les choses bien plus simples pour l'utilisateur que ipchains ne l'était. Aussi, le fait que le canevas netfilter fournit l'interface input et output pour les hooks NF_IP_FORWARD signifie que beaucoup de types de filtrage sont rendus plus simples.

Note : J'ai porté les parties kernel de ipchains et de ipfwadm en modules par dessus le canevas netfilter. Cela permet l'usage des vieux utilitaires en userspace ipfwadm et ipchains, sans avoir à mettre à jour.

3.2.2 NAT

Voici la réaction de la table 'nat', à qui l'on passe des paquets par deux hooks netfilter : pour les paquets qui ne sont pas générés localement, les hooks NF_IP_PRE_ROUTING et NF_IP_POST_ROUTING sont une place parfaite pour modifier respectivement la destination et la source d'un paquet. Si CONFIG_IP_NF_NAT_LOCAL est définie, le hook NF_IP_LOCAL_IN est utilisé pour modifier la destination des paquets générés localement.

Cette table est légèrement différente de la table 'filter', effectivement, seulement le premier paquet d'une nouvelle connexion va traverser la table : le résultat de cette traversé est ensuite appliquée à tous les futurs paquets qui font partie de la même connexion.

Masquerading, Port Forwarding, et Proxy transparent Je divise NAT en Source NAT (ou le premier paquet voit sa source être modifiée) et Destination NAT (ou le premier paquet voit sa destination être modifiée).

Le Masquerading est une forme spéciale de Source NAT, port-forwarding et transparent proxying sont des formes spéciales de Destination NAT. Elles sont toutes maintenant effectuées à l'aide du canevas NAT, plutôt que d'être des entités indépendantes.

3.2.3 Modification de Paquets

La table de modification de paquets (la table 'mangle') est utilisée pour des changements concrets des informations d'un paquet. Les exemples sont la target TOS et la target TCPMSS. La table mangle s'accroche aux 3 hooks de netfilter (veuillez noter que ceci a changé à partir du noyau 2.4.18. Les noyaux précédents n'avaient pas la table mangle accrochée à tout les hooks).

3.3 Suivi de connexions

Le suivi des connexions est fondamental pour NAT, mais il est implémenté dans un module à part entière. Cela permet une extension au code de filtrage de paquets pour simplement et proprement implémenter le suivi de connexions (le module 'state').

3.4 Autres additions

Cette nouvelle flexibilité fournit une opportunité de faire des choses vraiment cools, et de manière plus importante permet aux gens d'écrire des améliorations ou des remplacement complets.

4 Informations pour les programmeurs

Je vais vous donner un secret : c'est mon hamster qui fait tout le coding. Je suis juste un moyen pour mon hamster, une façade pour le reste du monde. Et donc, si il y a des bugs, ne vous plaignez pas à moi. Mettez ça sur le dos du poilu à la place.

4.1 Comprendre ip_tables

iptables fournit simplement un tableau de règles en mémoire (d'où le nom iptables), et des informations comme quel paquet à partir de quel hook va où ? Après qu'une table est enregistrée, le userspace peut lire et remplacer sont contenu en utilisant les fonctions getsockopt() et setsockopt().

iptables ne s'enregistre avec aucun des hooks de netfilter, il s'appuie sur d'autres modules pour faire ça, et passe les paquets de manière appropriée. Un module doit enregistrer le hook netfilter et ip_tables séparément, et fournir un mécanisme pour appeler ip_tables quand le hook est atteint.

4.1.1 Structures de données d'ip_tables

Pour la convenance, la même structure de données est utilisée pour représenter une règle en userspace et à l'intérieur du kernel (bien que quelques champs seulement sont utilisés dans le kernel).

Chaque règle contient les parties suivantes :

1. Une 'struct ipt_entry'.
2. 0 ou plus 'struct ipt_entry_match', chacune avec une taille variable (0 ou plus octets) de données accrochées à elles.
3. Une 'struct ipt_entry_target', avec une taille variable de données (0 ou plus octets) accrochées à elles.

La nature variable des règles donne beaucoup de flexibilité pour les extensions, comme nous allons le voir, spécialement comme chaque "target" ou chaque "match" peut porter une taille arbitraire de données. Cependant, ceci a quelques désavantages : on a besoin de faire attention à l'alignement. On fait ça en s'assurant que les structures 'ipt_entry', 'ipt_entry_match' et 'ipt_entry_target' sont de taille conventionnelle, et que les tailles de toutes les données sont arrondies vers le haut jusqu'à l'alignement maximum de la machine, en utilisant la macro IPT_ALIGN().

La 'struct ipt_entry' a les champs suivants :

1. Une 'struct ipt_ip', qui contient les spécifications pour le header IP qui est à matcher.
2. Un tableau de bits 'nf_cache' qui montre quel partie du paquet cette règle a examiné.
3. Un champs 'target_offset' indiquant l'offset à partir du début de la règle, où la structure ipt_entry_target commence. Ça devrait être toujours aligné correctement à l'aide de la macro IPT_ALLIGN().
4. Un champs 'next_offset' indiquant la taille totale la règle, en incluant les matches, et la target. Ça doit être aussi toujours aligné correctement, à l'aide de la macro IPT_ALLIGN().
5. Un champs 'comefrom' utilisé par le kernel pour suivre la trace des paquets dans leur traversé.
6. Un champs 'struct ipt_counters' contenant le paquet et les compteurs d'octets pour les paquets qui on matché cette règle.

'struct ipt_entry_match' et 'struct ipt_entry_target' sont très similaires, du fait qu'elles contiennent toutes les deux un champs (IPT_ALIGNé) représentant la longueur totale (respectivement 'match_size' et 'target_size') et une 'union' contenant le nom du match ou de la target (pour le userspace), et un pointeur (pour le kernel).

A cause de la nature légèrement ... "bidouille" de la structure de données des règles, des fonctions d'aide sont fournies :

ipt_get_target()
: Cette fonction 'inline' retourne un pointeur vers la target de la règle.

IPT_MATCH_ITERATE()
: Cette macro appelle la fonction donnée pour chaque match de la règle donnée. Le premier argument de la fonction est 'struct ipt_match_entry' et les autres arguments (s'il y en a) sont ceux fournis à

la macro IPT_MATCH_ITERATE(). La fonction doit soit retourner 0 pour que l'itération continue, soir une valeur non nulle pour arrêter l'itération.

IPT_ENTRY_ITERATE()

Cette fonction prend un pointeur vers une 'entry' la taille totale de la table des 'entry's, et une fonction à appeler à chaque fois. Le premier argument de la fonction doit être la 'struct ipt_entry', et les autres arguments (s'il y en a) sont ceux fournis à la macro IPT_ENTRY_ITERATE(). La fonction doit retourner zéro pour que l'itération continue, ou une valeur non nulle pour arrêter l'itération.

4.1.2 ip_tables à partir du userspace

Le userspace a 4 opérations : il peut lire la table courante, lire les infos (les positions des hooks et taille des tables), remplacer la table (et prendre les anciens compteurs), et ajouter de nouveaux compteurs.

Cela permet de simuler n'importe quel opération atomique à partir du userspace : cela est fait par la librairie 'libiptc' qui fournit les fonctions utiles "add/delete/replace" pour les programmes.

Parce que ces tables sont transférées dans le kernelspace, l'alignement peut devenir un problème quand les règles d'alignement du userspace et du kernel sont différentes (par exemple : sparc64 avec un userspace 32 bits). Ces cas sont gérés en écrivant par dessus la définition de 'IPT_ALIGN' pour ces plate-formes dans libiptc.h

4.1.3 Utilisation d'ip_tables et la Traversé.

Le kernel commence à traverser à partir de la location indiquée par le hook. Cette règle est examinée, si l'élément 'struct ipt_ip' match, chaque 'struct ipt_entry_match' est vérifie à son tour (la fonction associée avec chaque match est appelée). Si la fonction match retourne 0, l'itération stoppe sur cette règle. Si la fonction met le paramètre 'hotdrop' à 1, le paquet va être aussi immédiatement détruit (c'est utilisé pour quelque paquets louches, comme dans la fonction tcp match).

Si l'itération continue jusqu'à la fin, les compteurs sont incrementés, la 'struct ipt_entry_target' est examinée : si c'est une target standard, le champs 'verdict' est lu (une valeur négative signifie un verdict du paquet, sinon, une valeur positive signifie un offset de déplacement). Si la réponse est positive et que l'offset n'est pas celui de la règle suivante, le drapeau 'back' est mis, et la dernière valeur de 'back' est reportée dans le champs 'comefrom' de la règle.

Pour les targets pas standards, la fonction target est appelée : elle retourne un verdict (les targets pas standards ne peuvent sauter ("jump"), comme cela casserait le code qui détecte les paquets qui bouclent sans fin). Le verdict peut être IPT_CONTINUE pour continuer à la règle suivante.

4.2 Étendre iptables

Parce que je suis fénéant, `iptables` est assez extensible. C'est typiquement une combine pour refiler le boulot à d'autres gens, ce pour quoi est fait l'Open Source de toutes façons (cf. Free Software, qui, comme dirait Richard Stalleman, est à propos de la liberté, et j'ai assisté à un de ses discours quand j'ai écris ça).

Étendre `iptables` comprend généralement 2 parties : Étendre le kernel, en écrivant un nouveau module, et aussi optionnellement étendre le programme `iptables` en userspace, en écrivant une nouvelle librairie partagée.

4.2.1 Le Kernel

Écrire un module pour le kernel en soit même est relativement simple, comme vous pouvez le voir dans les exemples. Une chose qu'il faut savoir cependant, c'est que votre code doit être re-entrant : il peut y avoir un paquet qui arrive du userspace pendant qu'un autre arrive à cause d'une interruption. En fait, avec SMP, il peut même y avoir un paquet venant d'une interruption par CPU dans les 2.3.4 et au dessus.

Les fonctions à connaître sont :

init_module()
: C'est le point d'entrée du module. Il retourne un nombre négatif représentant le code d'erreur ou 0 si il s'est enregistré avec succès avec netfilter.

cleanup_module()
: C'est le point de sortie du module. Il devrait se dé-enregistrer de netfilter.

ipt_register_match()
: C'est utilisé pour enregistrer un nouveau type de match. Vous lui donnez une 'struct ipt_match', qui est habituellement déclarée comme une variable 'static' (pour le fichier entier).

ipt_register_target()
: C'est utilise pour enregistrer un nouveau type de target. Vous lui donnez une 'struct ipt_target', qui est habituellement déclarée comme une variable 'static' (pour le fichier entier).

ipt_unregister_target()
: Utilisé pour dé-enregistrer votre target.

ipt_unregister_match()
: Utilisé pour dé-enregistrer votre match.

Un avertissement avant de fournir des trucs louches (comme fournir des compteurs) dans la place restante de votre nouveau match ou target. Sur les machines SMP, la table entière est dupliquée en utilisant 'memcpy()' pour chaque CPU : si vous voulez vraiment garder l'information centrale, vous devez aller voir la méthode qu'utilise le match 'limit'.

Les nouvelles fonctions de match. Les nouvelles fonctions de match son généralement écrites comme des modules à part entière. Il est ainsi possible d'avoir ces modules extensible à leur tour, mais ça n'est pas nécessaire habituellement. Un autre moyen serait d'utiliser la fonction 'nf_register_sockopt' du canevas netfilter pour permettre à l'utilisateur de parler directement à votre module. Un autre moyen finalement est d'exporter les symboles pour que d'autre modules puissent s'enregistrer eux même, de la même manière que netfilter et ip_tables le font.

Le centre de votre nouvelle fonction de match est la structure 'ipt_match' qui est fournie à 'ipt_register_match()'. Cette structure à les champs suivants :

list
: Ce champs est mis à n'importe quoi, par exemple disons '{NULL, NULL}'.

name
: Ce champs est le nom de la fonction de match, comme appelée dans le userspace. Le nom doit être le même que le nom de fichier du module (par exemple si le nom est "mac", le nom de du module doit être "ipt_mac.o") pour que le chargement automatique des modules fonctionne.

match
: Ce champs est un pointeur sur la fonction de match, qui prend le skb, les pointeurs sur les interfaces d'entrée "in" et de sortie "out" (un desquels peut être NULL, selon le hook), un pointeur vers les données de match de la règle en cours (la structure qui à été préparée en userspace), l'offset IP (une

valeur non nulle signifie qu'il s'agit d'un fragment (pas le premier)), un pointeur sur le header du protocole, la longueur des données (la taille du paquet moins la taille du header IP) et finalement, un pointeur vers une variable 'hotdrop'. La fonction doit retourner une valeur non nulle si le paquet match, et peut mettre le 'hotdrop' à 1 si elle retourne 0, pour indiquer que le paquet doit être détruit immédiatement.

checkentry

Ce champs est un pointeur vers une fonction qui vérifie les spécifications d'une règle. Si la fonction retourne 0, alors la règle ne sera pas acceptée. Par exemple, le match "tcp" ne va accepter seulement que des paquets TCP, et donc, si la 'struct ipt_ip' qui fait partie de la règle ne spécifié pas explicitement que les paquets doivent être de type tcp, 0 est retourné. L'argument 'tablename' vous permet de controler dans quelle table votre match peut être utilisé, et le 'hook_mask' est un masque de bit de hooks qui définit de quel hooks cette règle peut être appelé : si votre match n'a aucun sens dans certain hooks de netfilter, vous pouvez éviter ça ici.

destroy

Ce champs est un pointeur vers une fonction qui est appelée quand une entrée utilisant ce match a été effacée. Cela vous permet d'allouer dynamiquement des ressources dans 'checkentry' et de les nettoyer ici.

me

Ce champs est mis à 'THIS_MODULE', ce qui donne un pointeur vers votre module. Cela cause l'incrémentation ou la décrémentation du compteur d'utilisation, chaque fois qu'une règle utilisant votre module est ajoutée ou effacée. Cela permet d'éviter à un utilisateur d'enlever explicitement le module (et par conséquent d'appeler la fonction cleanup_module()) si une règle utilise encore votre module.

Nouvelles Targets Si votre target modifie un paquet (par exemple le header ou le corps du paquet), elle doit appeler la fonction skb_unshare() pour copier le paquet au cas où le paquet est cloné. Sinon n'importe quelle raw socket qui a un clone du skbuff verra les modifications (les gens verront des trucs bizarre arriver avec tcpdump).

Les nouvelles targets sont habituellement écrites comme des modules à part entière. La discussion à ce propos trouvée dans la section ci-dessus 'Nouvelles Fonctions de match' s'applique ici aussi.

Le centre de votre nouvelle target est la 'struct ipt_target' qui est passée en paramètre à la fonction ipt_register_target(). Cette structure a les champs suivants :

list

Ce champs est mis à n'importe quelle valeur, disons '{NULL, NULL}'.

name

Ce champs est le nom de la fonction de target, comme appelée dans le userspace. Le nom doit correspondre à celui du module (par exemple, si le nom est "REJECT", le module doit s'appeler "ipt_REJECT.o") pour que le chargement automatique des modules fonctionne.

target

Ceci est un pointeur vers la fonction de target, qui prend en paramètre le skbuff, le numéro de hook, 2 pointeurs sur les périphériques d'entrée et de sortie (l'un ou l'autre pouvant être NULL), un pointeur sur des données de target, et la position de la règle dans la table. Cette fonction de target peut soit retourner IPT_CONTINUE (-1) si la traversé doit continuer, ou un verdict netfilter (NF_DROP, NF_ACCEPT, NF_STOLEN, etc...).

checkentry

Ce champs est un pointeur vers une fonction qui vérifie les spécifications d'une règle. Si elle retourne 0, alors la règle ne sera pas acceptée.

destroy

Ce champs est un pointeur vers une fonction qui est appelée quand une entrée utilisant cette target est effacée. Cela vous permet d'allouer dynamiquement des ressources dans 'checkentry' et de les libérer ici.

me

Ce champs est mis à 'THIS_MODULE', ce qui donne un pointeur vers votre module. Ceci cause l'incrémentation ou la décrémentation du compteur d'utilisation quand une règle utilisant votre target est ajoutée ou effacée. ça évite qu'un utilisateur puisse enlever votre module si une règle s'en sert encore.

Nouvelles Tables Vous pouvez créer une nouvelle table pour vos propres besoins si vous le souhaitez. Pour faire cela, vous devez appeler la fonction 'ipt_register_table()', avec une 'struct ipt_table' qui a les champs suivants :

list

Ce champs est mis à n'importe quelle valeur, disons '{NULL, NULL}'.

name

Ce champs est le nom de la fonction de table, comme appelée dans le userspace. Le nom doit correspondre à celui du module (par exemple, si le nom est "nat", le module doit s'appeler "iptable_nat.o") pour que le chargement automatique fonctionne.

table

Ceci est une 'struct ipt_replace' déjà remplie, comme utilisée dans le userspace pour remplacer une table. Le pointeur 'counters' doit être mis à NULL. Cette structure de données peut être déclarée comme '__initdata' pour être oubliée après le boot.

valid_hooks

Ceci est un masque de bits pour définir les hooks IPv4 de netfilter que la table va utiliser. Ceci est utilisé pour vérifier que les points d'entrée sont valides, et pour calculer les hooks possibles pour les fonctions checkentry() utilisées dans ipt_match et ipt_target.

lock

Ceci est un 'spinlock' en lecture-écriture pour la table entière. Initilisez le à RW_LOCK_UNLOCKED.

private

Ceci est utilisé en interne par le code d'ip_tables.

4.2.2 Outils userspace

Vous avez écrit votre superbe module kernel dont vous voulez contrôler les options à partir du userspace. Plutôt que d'avoir une branche différente d'`iptables` pour chaque extension, j'utilise la dernière technologie des années 90 : les librairies partagées.

Les nouvelles tables n'ont pas besoin d'extension à `iptables` généralement : l'utilisateur utilise simplement l'option '-t' pour utiliser la nouvelle table.

Les librairies partagées devraient avoir une fonction '_init()', qui sera appellée automatiquement après leur chargement : c'est un peu équivalent à la fonction 'init_module()' qu'on utilise dans les modules kernel. Cette fonction devrait appeler 'register_match()' ou 'register_target()' respectivement si votre librairie partagée fournit un nouveau match ou une nouvelle target.

Vous devez fournir une librairie partagée : elle peut être utilisée pour initialiser une partie de la structure, ou fournir des options additionnelles. J'insiste maintenant sur une librairie partagée, même si celle-ci ne fait rien, pour limiter les problèmes.

Il y a plusieurs fonctions utiles décrites dans le header ‘iptables.h’, spécialement :

check_inverse()
Vérifie qu’un argument est en fait un ‘ !’, et si c’est le cas, met le flag ‘invert’ s’il n’est pas déjà mis. Si ça renvoie vrai, vous devriez incrémenter ‘optind’, comme fait dans les exemples.

string_to_number()
convertit une chaîne de caractères en un nombre, dans l’interval demandé, retournant -1 si la chaîne est malformée ou si le nombre sort de l’interval.

exit_error()
Doit être appelée si une erreur a été trouvée. Habituellement, le premier argument est ‘PARAMETER_PROBLEM’, ce qui signifie que l’utilisateur n’a pas utilisé la ligne de commande correctement.

Nouvelles fonctions de match La fonction ‘_init()’ de votre librairie partagée donne à ‘register_match()’ un pointeur sur une ‘struct iptables_match’ static, qui a les champs suivants :

next
Ce pointeur est utilisé pour faire une liste chaînée de matchs (comme pour lister les règles). Il doit être mis à ‘NULL’ au début.

name
Le nom de la fonction de match. Ceci doit correspondre au nom de la librairie (par exemple "tcp" pour "libipt_tcp.so").

version
Habituellement, on met IPTABLES_VERSION (macro) : c’est fait pour être sûr que le programme `iptables` ne charge pas la mauvaise version d’une librairie partagée par erreur.

size
La taille des données de match pour ce match. Vous devriez utilisez la macro IPT_ALIGN() pour vous assurer que c’est correctement aligné.

userspacesize
Pour quelques matches, le kernel change quelques champs en interne (la target ‘limit’ en est un bon exemple). Cela signifie que un simple ‘memcmp()’ n’est pas suffisant pour comparer deux règles (nécessaire pour la fonction effacer-la-règle-correspondante). Si c’est le cas, placez tous les champs qui ne changent pas au début de la structure, et mettez la taille des champs inchangés ici. Bien sur, la plupart du temps cette valeur sera égale à celle du champs ‘size’.

help
Une fonction qui imprime le message d’aide pour l’option.

init
Ceci peut être utilisé pour initialiser de l’espace supplémentaire (si besoin est) dans la structure ipt_entry_match, et initialiser les bits de ‘nfcache’. Si vous examinez quelque chose qui n’est pas exprimable à l’aide du contenu de ‘linux/include/netfilter_ipv4.h’, alors affectez simplement ce champs en faisant un OR avec NFC_UNKNOWN. Cette fonction sera appelée avant ‘parse()’.

parse
Cette fonction est appelée quand une option non reconnue est vue sur la ligne de commande : elle devrait retourner une valeur non nulle si effectivement cette option vous été destinée. ‘invert’ est vrai si un ‘ !’ a été déjà vu. Le pointeur ‘flags’ est pour l’utilisation exclusive de votre librairie de match, et il est généralement utilisé pour garder un masque de bits contenant les options déjà spécifiées. Faites bien attention à ajuster les bits de ‘nfcache’. Vous pouvez étendre la taille de la structure ‘ipt_entry_match’ en ré-allouant de l’espace si nécessaire, mais alors il faut vous assurer que la taille est passée en utilisant la macro IPT_ALIGN().

final_check

Cette fonction est appellée après que la ligne de commande a été examinée, est reçois en paramètre la variable entière 'flags' réservée à votre librairie. Cela vous donne une dernière chance de vérifier que n'importe quel option obligatoire a bien été spécifiée, par exemple appelez la fonction 'exit_error()' s'il en manque une.

print

Cette fonction est utilisée par le code de listage de règles pour imprimer sur la sortie standard les informations spécifiques à votre match (s'il y en a) pour une règle. le flag 'numeric' est mis à vrai si l'utilisateur a spécifié l'option '-n' sur la ligne de commande.

save

Cette fonction est l'inverse de 'parse' : elle est utilisée par 'iptables-save' pour reproduire les options utilisées pour créer la règle.

extra_opts

C'est une liste 'NULL-terminated' (terminée par un NULL) d'options supplémentaires que votre librairie offre. Cette liste est mélangée aux autres options d'`iptables` qui sont gérées par 'getopt_long()'. Allez voir la man-page de 'getopt_long' pour plus de détails. L'argument retourné par getopt_long devient le premier argument ('c') pour votre fonction 'parse()'.

Il y a des éléments additionnels à la fin de cette structure réservés pour usage interne à `iptables`, vous n'avez pas besoin de les utiliser.

Nouvelles fonctions de target. La fonction '_init()' de votre librairie partagée donne à 'register_target()' un pointeur vers une 'struct iptables_target' static, qui a des champs similaires aux champs de la 'struct iptables_match' décrite au-dessus.

4.2.3 Utiliser 'libiptc'

`libiptc` est la librairie de contrôle de iptables, étudiée pour lister et manipuler les règles dans le module kernel iptables. Bien qu'elle est actuellement principalement utilisée par le programme iptables, cela rend beaucoup plus facile l'écriture de nouveaux utilitaires. Vous devez être 'root' pour utiliser ces fonctions.

Les tables kernel elles même sont une table de règles, et un jeu de nombre représentant les points d'entrée. Les noms de chaînes ("INPUT", etc...) sont fournies en tant qu'abstraction par la librairie. Les chaînes crées par l'utilisateur sont étiquetées en insérant un noeud d'erreur avant la tête de la chaîne définie par l'utilisateur, qui contient le nom de la chaîne dans la section de données supplémentaires de la target (la position des chaînes par défaut sont définies par les trois points d'entrée de la table).

Les targets standard suivantes sont supportées : ACCEPT, DROP, QUEUE (qui sont traduites respectivement en NF_ACCEPT, NF_DROP, ND_QUEUE), RETURN (qui est traduit en spécialement en IPT_RETURN qui sera géré par iptables) et JUMP (traduit à partir du nom de la chaîne courante vers un offset dans la table).

Quand 'iptc_init()' est appelé, la table, incluant les compteurs sont lus. Cette table est manipulée par les fonctions 'iptc_insert_entry()', 'ipt_replace_entry', 'iptc_replace_entry()', 'iptc_append_entry()', 'iptc_delete_entry()', 'iptc_delete_num_entry()', 'iptc_flush_entries()', 'iptc_zéro_entries()', 'iptc_create_chain()' 'iptc_delete_chain()', et 'iptc_set_policy()'.

Les changements portés sur la table ne sont pas ne sont pas effectués jusqu'à ce que vous appeliez la fonction 'iptc_commit()'. Ce qui signifie que deux utilisateurs manipulant la même chaîne en même temps vont rencontrer des problèmes. 'locking' (pour la synchronisation) est nécessaire pour éviter cela, mais ça n'est pas encore implémenté.

Cependant, il n'y a pas de condition de course avec les compteurs. Les compteurs sont re-ajoutés au kernel de telle sorte que l'incrémentation d'un compteur entre la lecture et l'écriture de la table se voit toujours.

Il y a quelques fonctions utiles :

iptc_first_chain()
: Cette fonction retourne le nom de la première chaîne de la table.

iptc_next_chain()
: Cette fonction retourne le nom de la chaîne suivante dans la table, NULL signifie qu'il n'y a plus de chaînes.

iptc_builtin()
: Retourne vrai si le nom de la chaîne suivante correspond à une chaîne qui est pré-définie.

iptc_first_rule()
: Ceci retourne un pointeur vers la première règle de la chaîne dont on a donné le nom, NULL pour une chaîne vide.

iptc_next_rule()
: Cette fonction retourne une pointeur vers la prochaine règle dans la chaîne dont a donné le nom, NULL pour la fin de la chaîne.

iptc_get_target()
: Cette fonction retourne la target d'une règle donnée. Si c'est une target étendue, le nom de la target est retourné. Si c'est un saut vers une autre chaîne, le nom de la chaîne est retourné. Si c'est un verdict (par exemple DROP), ce nom est retourné. Si la règle n'a pas de target (règle de comptage par exemple) alors une chaîne nulle est retournée.

  Notez que cette fonction devrait être utilisée à la place d'utiliser directement la valeur qui est dans le champs 'verdict' de la structure 'ipt_entry', comme elle contient les interprétations que l'on a expliqué plus haut.

iptc_get_policy()
: Cette fonction retourne la politique d'une chaîne pré-définie, et rempli l'argument 'counters' avec les statistiques d'utilisation de cette politique.

iptc_strerror()
: Cette fonction retourne une explication plus sensée d'un code d'erreur qui s'est passe dans le code de la librairie iptc. Si une fonction plante, elle mettra à jour la valeur de 'errno' : cette valeur peut être passée à 'ipt_sterror()' pour retourner le message d'erreur correspondant.

4.3 Comprendre la NAT

Bienvenue dans le monde du 'Network Address Traslation (NAT)' (Traduction D'adresses Réseau). Notez que l'infrastructure offerte a été étudiée pour plus de complétion que de vitesse, et que des modifications mineures futures pourront certainement augmenter l'efficacité du code. Pour le moment, je suis déjà content que ça marche !

NAT est séparée en 'connection tracking' (suivi des connexions) qui ne manipule pas les paquets du tout, et le code de NAT lui même. Le suivi de connexions est aussi étudié pour être utilisé par un module spécial de iptables, et donc fait des différences subtiles d'état de connexion que NAT n'a pas besoin.

4.3.1 Connection Tracking (Suivi de connexions)

Le suivi de connexions s'accroche dans le début du hook NF_IP_LOCAL_OUT et NF_IP_PRE_ROUTING, de telle sorte qu'il puisse voir les paquets avant qu'ils n'entrent dans le système.

Le champs ‘ncft’ dans le skb est un pointeur vers un des tableaux infos[] dans la structure ip_contrack. Du coup, on est capable de dire l’état du skb en sachant vers quel élément de ce tableau il pointe : ce pointeur encode la structure d’état et la relation entre ce skb vis-à-vis de cet état.

Le meilleur moyen d’extraire le champs ‘ncft’ est d’appeler la fonction ‘ip_conntrack_get()’, qui retourne NULL si ‘ncft’ n’est pas mis, et rempli le ‘ctinfo’ qui décrit la relation entre le paquet et la connexion. Cette énumération peut avoir plusieurs valeurs :

IP_CT_ESTABLISHED
: Le paquet fait partie d’une connexion établie, dans la direction originale.

IP_CT_RELATED
: Le paquet est en relation avec la connexion, et il est en train de passer dans la direction originale.

IP_CT_NEW
: Le paquet est en train d’essayer de créer une nouvelle connexion (évidement, il est dans la direction originale).

IP_CT_ESTABLISHED + IP_CT_IS_REPLY
: Le paquet fait partie d’une connexion établie, mais dans la direction réponse.

IP_CT_RELATED + IP_CT_IS_REPLY
: Le paquet est en relation avec une connexion, mais il est dans la direction réponse.

Donc, un paquet réponse peut être identifie en testant la condition ‘>= IP_CT_IS_REPLY’.

4.4 Étendre le suivi de connexions/NAT

Ces canevas sont étudiés pour accommoder n’importe quel type de protocole et type de mapping. Quelques un de ces types de mapping peuvent être très spécifiques, comme le type de mapping pour le load-balancing et le fail-over.

En interne, le code de suivi de connexions convertit un paquet en un n-uplet qui représente les parties intéressantes du paquet, avant de chercher les règles qui correspondent. Ce n-uplet a une partie manipulable, et une partie non manipulable, appelées respectivement "src" et "dst", comme il s’agit du point de vue du code du Source NAT quand il voit le premier paquet (ce serait un paquet réponse dans le code du Destination NAT). Le n-uplet pour chaque paquet d’une même connexion dans le même sens est tout simplement le même.

Par exemple, le n-uplet d’un paquet TCP contient la partie manipulable : adresse IP source et port source, et la partie non manipulable : l’adresse IP de destination et le port de destination. Les parties manipulable et non manipulable ne sont pas nécessairement du même type. Par exemple, un n-uplet d’un paquet ICMP contient une partie manipulable : adresse IP source et l’id icmp, et la partie non manipulable : l’adresse IP de destination et l’id icmp et code icmp.

Chaque n-uplet a un inverse, qui est le n-uplet du paquet réponse dans le flux. Par exemple, l’inverse d’un paquet icmp ping avec un numéro id 12345 en provenance de 192.168.1.1 vers 1.2.3.4 est un paquet icmp icmp pong avec un numéro id 12345 en provenance de 1.2.3.4 vers 192.168.1.1.

Ces n-uplets, représentés par la ‘struct ip_contrack_tuple’, sont utilisé largement. En fait, avec le hook d’où provient le paquet (qui a effet sur le type de manipulation attendu) et l’interface en question, cela est en fait l’information complète d’un paquet.

La plupart des n-uplets sont contenus dans un ‘struct ip_conntrack_tuple_hash’, qui ajoute une entrée a une liste doublement chaînée, et un pointeur vers la connexion à laquelle appartient ce n-uplet.

Une connexion est représentée par une ‘struct ip_conntrack’ : ça a 2 champs ‘struct ip_conntrack_tuple_hash’ : un qui référence la direction du paquet originel (tuplehash[IP_CT_DIR_ORIGINAL]), et 1 qui référence un paquet dans la direction réponse (tuple-hash[IP_CT_DIR_REPLY]).

De toutes façons, la première chose que le code de NAT fait est de voir si le code de suivi de connexions a réussi à extraire un n-uplet et à trouver une connexion existante, en regardant le champs ‘nfct’ du skbuff. Cela nous dit si c’est une tentative de nouvelle connexion, ou sinon, dans quelle direction le paquet va, auquel cas les manipulations déterminées précédemment sont faite.

Si c’était le début d’une nouvelle connexion, on cherche une règle qui correspond à ce n-uplet, en utilisant le mécanisme standard d’iptables de traversé, sur la table ‘nat’. Si une règle match, c’est utilisé pour initialiser les manipulations pour les 2 directions. Le code de suivi de connexions est notifié que la réponse qu’il doit attendre est maintenant différente. Alors, c’est manipulé comme au dessus.

Si il n’y a pas de règles, une attache ‘null’ est crée : elle ne correspond pas au paquet, mais existe pour être sur qu’on n’accroche pas un autre flux existant. De temps en temps l’attache nulle ne peut pas être crée parce que l’on a déjà attaché un flux existant dessus, dans quel cas la manipulation par-protocole peut essayer de le ré-attacher, même si c’est en fait une attache nulle.

4.4.1 Les targets standard de NAT

Les targets standard de NAT sont comme les autres extensions de target d’iptables, mis à part qu’elle insistent sur le fait de n’être utilisées que dans la table ‘nat’. Aussi bien la target SNAT que DNAT prennent une ‘struct ip_nat_multi_range’ en données externes. C’est utilisé pour spécifier l’interval d’adresses qu’une attache est autorisée à utiliser. Un interval consiste en une ‘struct ip_nat_range’ et en une adresse IP maximum inclusive, une adresse IP minimum inclusive, un minimum et maximum inclusifs pour les valeurs spécifiques au protocole (par exemple des ports pour TCP). Il y a aussi de la place pour des flags qui disent si une adresse IP peut être masquée (de temps en temps on ne veut masquer que la partie spécifique au protocole et pas IP), et une autre pour dire si l’interval concernant la partie spécifique au protocole est valide.

Un ‘multi-range’ est un tableau de ces éléments ‘struct_ip_nat_range’. Ça veut dire qu’un interval peut être "1.1.1.1-1.1.1.2 ports 50-55 ET 1.1.1.3 port 80". Chaque élément interval ajoute sont interval (une ‘union’ pour ceux qui aiment les théories mathématiques).

4.4.2 Nouveaux protocoles

Dans le kernel Implémenter un nouveau protocole signifie d’abord décider qu’est ce que les parties manipulable et non manipulable doivent être. Un n-uplet a la propriété d’identifier un flux de manière unique. La partie manipulable du n-uplet est celle avec laquelle vous pouvez faire la DNAT : pour TCP c’est le port source, pour ICMP c’est l’id icmp. Quelque chose à utiliser comme identifiant de flux. La partie non manipulable est le reste du paquet qui identifie le paquet de manière unique, mais on ne peut pas jouer avec (ex : port de destination d’un paquet TCP, le type d’un paquet ICMP).

Une fois que vous vous êtes décidé, vous pouvez écrire une extension au code de suivi de connexions dans le répertoire, et commencer à remplir la structure ‘ip_conntrack_protocol’ que vous devez passer en paramètre à ‘ip_conntrack_register_protocol()’.

Les champs d’une ‘struct ip_conntrack_protocol’ sont :

list

Mis à ‘{NULL, NULL}’, utilisé pour vous coudre dans la liste.

proto

Votre numéro de protocole, voir ‘/etc/protocols’.

name

Le nom de votre protocole. C'est le nom que l'utilisateur va voir. C'est mieux si c'est le non canonique trouvé dans '/etc/protocols'.

pkt_to_tuple

La fonction qui remplis les parties du n-uplet spécifiques au protocole, donné le paquet. Le pointeur 'datah' pointe vers le début de votre header (juste après le header IP), et la variable 'datalen' contient la longueur du paquet. Si le paquet n'est pas assez long pour contenir les informations du header, retourne 0. Néanmoins, 'datalen' sera toujours au moins 8 octets (forcé par le canevas).

invert_tuple

Cette fonction est simplement utilisée pour changer la partie spécifique au protocole du n-uplet en ce à quoi ressemblerai le n-uplet d'une réponse au paquet.

print_tuple

Cette fonction est utilisée pour imprimer la partie spécifique au protocole du n-uplet. Généralement, il s'agit de 'sprintf()'s dans le tampon fournis. Le nombre de caractères du tampon est retourné. C'est utilisé pour imprimer les états de connexions dans une entrée /proc.

print_conntrack

Cette fonction est utilisée pour imprimer la partie privée de la structure conntrack, si il y en une, aussi utilisé pour imprimer l'état de la connexion dans /proc.

paquet

Cette fonction est appelée quand un paquet qui fait partie d'une connexion établie est vu. Vous recevez un pointeur vers la structure conntrack, vers le header IP, la longueur, et le 'ctinfo'. Vous retournez un verdict pour le paquet (habituellement NF_ACCEPT), ou -1 ne fait pas partie d'une connexion valide. Vous pouvez effacer la connexion à l'intérieur de cette fonction, mais vous devez suivre la marche suivante (pour éviter les conditions de course) :

```
if (del_timer(&ct->timeout))
        ct->timeout.function((unsigned long)ct);
```

new

Cette fonction est appelée quand un paquet crée une nouvelle connexion pour la première fois. Il n'y pas de paramètre 'ctinfo' puisque le premier paquet a toujours une 'ctinfo' égale a IP_CT_NEW par définition. Ça retourne 0 pour rater la création de connexion, ou un timeout de la connexion en jiffies.

Une fois que vous avez écrit et testé que vous pouvez suivre le nouveau protocole, il est temps d'apprendre à NAT comment traduire l'adresse. ça veut dire écrire un nouveau module, une extension au code de NAT. Vous devez remplir la structure 'ip_nat_protocol' que vous devez passer en paramètre à la fonction 'ip_nat_protocol_register()'.

list

Mettez le à '{ NULL, NULL }' ; Utilisé pour vous coudre dans la liste.

name

Le nom de votre protocole. Ce nom est le nom que l'utilisateur verra. C'est mieux si c'est le nom canonique du protocole comme trouvé dans '/etc/protocols', pour le chargement automatique des modules par le userspace, comme on le verra plus tard.

protonum

Votre numéro de protocole, voir '/etc/protocols'

manip_pkt

C'est la deuxième moitié de la fonction pkt_to_tuple du code de suivi de connexions : vous pouvez la comparer à un "tuple_to_pkt". Il y a néanmoins quelques différences : vous recevez un pointeur sur le début du header IP, et la longueur totale du paquet. C'est parce que quelques protocoles (UDP, TCP) ont besoin de connaître le header IP. Vous recevez le champs 'ip_nat_tuple_manip' du n-uplet (ex : le champs "src"), plutôt que le n-uplet entier, et le type de manipulation que vous avez à effectuer.

in_range

Cette fonction est utilisée pour dire si la partie manipulable du n-uplet est dans l'interval donné. Cette fonction est un peu difficile : on vous donne le type de manipulation qui a été effectué sur le n-uplet ce qui nous dit comment interpréter l'interval (est ce que c'est un interval source ou un interval destination dont on parle ?)

Cette fonction est utilisée pour vérifier si une attache existante nous met dans l'interval voulu, et aussi pour vérifier si aucune manipulation n'est nécessaire du tout.

unique_tuple

Cette fonction est le noyau de NAT : donné un n-uplet et un interval, on va changer la partie spécifique au protocole du n-uplet pour le mettre dans l'interval et le rendre unique. Si on ne peut pas trouver de n-uplet pas encore utilisé dans l'interval, retourne 0. On reçoit aussi un pointeur vers la structure conntrack, qui est requise par 'ipt_nat_used_tuple()'.

L'approche habituelle est de simplement itérer la partie spécifique au protocole du n-uplet dans l'interval, en vérifiant avec 'ip_nat_used_tuple()' dessus, jusqu'à ce que ça retourne faux.

Notez que le cas d'attache nulle a déjà été vérifie : soit c'est en dehors de l'interval donné, soit c'est déjà pris.

Si IP_NAT_RANGE_PROTO_SPECIFIED n'est pas encore mis, ça signifie que l'utilisateur fait du NAT, et pas du NAPT : soyez sensibles avec les intervals. Si aucune attache n'est désirable (par exemple, dans TCP, une attache de destination ne devrait pas changer le port TCP sauf si on le lui demande), retournez 0.

print

Donné un tampon de caractères, match un n-uplet et un masque, et écrit la partie spécifique au protocole, et finalement retourne la longueur du tampon utilise.

print_range

Donné un tampon de caractères et un interval, écrit l'interval faisant partie de la partie spécifique au protocole du n-uplet, et retourne la longueur du tampon utilisé. Ce ne sera pas appelé si le flag IP_NAT_RANGE_PROTO_SPECIFIED n'a pas été mis à vrai pour l'interval.

4.4.3 Nouvelles targets NAT

C'est vraiment la partie intéressante. Vous pouvez écrire de nouvelles target NAT qui fournissent de nouvelles type d'attaches : deux targets supplémentaires sont fournies dans le package par défaut : MASQUERADE et REDIRECT. Elles sont assez simple pour illustrer le potentiel et la puissance d'écrire de nouvelles targets NAT.

Elles sont écrites comme n'importe quelles autres target iptables, mais en interne, elles vont extraire la connexion et appeler la fonction 'ip_nat_setup_info()'.

4.4.4 Protocol Helpers

Les Protocol helpers permettent au code de suivi de connexions de comprendre certains protocoles qui utilisent de multiples connexions réseaux (ex : FTP) et de marquer la connexion 'fille' comme étant en relation ('RELATED') à la connexion initiale, habituellement en lisant les adresses en relation dans le flux de données.

Les Protocol helpers pour Nat font 2 choses : d'abord elles permettent au code de NAT de manipuler le flux de données pour changer l'adresse contenu dedans, et ensuite de s'occuper du NAT sur la connexion en relation quand elle arrive, basé sur la connexion originale.

4.4.5 Helpers Modules pour le suivi de connexions

Description Le devoir du module de suivi de connexions est de spécifier quels paquets appartient à quel connexions déjà existantes. Le module a les moyens suivant pour faire ça :

- Dire à Netfilter par quel paquet notre module est intéressé (la plupart de Helpers opèrent sur un port en particulier).
- Enregistrer une fonction avec Netfilter. Cette fonction est appelée pour chaque paquet qui match le critère du dessus.
- Une fonction 'ip_conntrack_expect_related()' qui peut être appelée pour dire à Netfilter de s'attendre à des connexions en relation.

S'il y a plus de travail à effectuer au premier paquet reçu d'une connexion en relation, le module peut enregistrer une fonction callback qui sera rappelée à ce moment.

Structures et Fonctions Disponibles La fonction 'init' de votre module kernel doit appeler 'ip_conntrack_helper_register()' avec un pointeur vers une 'struct ip_conntrack_helper'. Cette structure a les champs suivants :

list
: C'est le header pour la liste chaînée. Netfilter gere cette liste en interne. Initilisez la juste à '{ NULL, NULL }'.

name
: Un pointeur vers une chaîne de caractères contenant le nom du protocole. ("ftp", "irc", "...")

flags
: Un groupe d'option avec un ou plusieurs des flags suivants :
  - IP_CT_HELPER_F_REUSE_EXPECTRe-utilise les entrées de connexions attendues si la limite (voir max_expected ci-dessous) est atteinte.

me
: Un pointeur sur la structure de module. Initialisez ça à la macro 'THIS_MODULE'.

max_expected
: Nombre maximum de connexions attendues qui ne sont pas encore confirmées (en attente).

timeout
: Le temps de vie maximum (en secondes) pour chaque connexion attendue. Une connexion attendue est effacée 'timeout' secondes après que l'attente de connexion a été émise par la fonction 'ip_conntrack_expect_related()'.

tuple
: C'est une 'struct ip_conntrack_tuple' qui spécifié les paquets par lesquels notre Helper pour le suivi de connexions est intéressé.

mask
: Encore une fois, une 'struct ip_conntrack_tuple'. Ce masque spécifie quels bits du `tuple` sont valides.

help
: La fonction que Netfilter doit appeler pour chaque paquet qui match le n-uplet+mask.

Exemple de base d'un Helper Module pour le suivi de connexions :

```
#define FOO_PORT        111

static int foo_expectfn(struct ip_conntrack *new)
{
        /* appelé quand le premier paquet d'une connexion attendue arrive */

        return 0;
}

static int foo_help(const struct iphdr *iph, size_t len,
                struct ip_conntrack *ct,
                enum ip_conntrack_info ctinfo)
{
        /* Analyse les données passées sur cette connexion et
           décide à quoi les paquets en relation vont ressembler */
à
        /* Mets à jour les données privées de chaque connexion maîtresse (session, état, ...) */
        ct->help.ct_foo_info = ...

        if (il_y_aura_de_nouveaux_paquets_en_relation_avec_cette_connexion)
        {
                struct ip_conntrack_expect exp;

                memset(&exp, 0, sizeof(exp));
                exp.t = tuple_specifying_related_packets;
                exp.mask = mask_for_above_tuple;
                exp.expectfn = foo_expectfn;
                exp.seq = tcp_sequence_number_of_expectation_cause;

                /* les données des connexions esclaves.
                exp.help.exp_foo_info = ...

                ip_conntrack_expect_related(ct, &exp);
        }
        return NF_ACCEPT;
}

static struct ip_conntrack_helper foo;

static int __init init(void)
{
        memset(&foo, 0, sizeof(struct ip_conntrack_helper);

        foo.name = "foo";
        foo.flags = IP_CT_HELPER_F_REUSE_EXPECT;
        foo.me = THIS_MODULE;
        foo.max_expected = 1;   /* Une attente à la fois */
        foo.timeout = 0;        /* L'attente n'expire jamais */

        /* Nous sommes intéressés par tous les paquets TCP avec un
           port destination de 111 */
        foo.tuple.dst.protonum = IPPROTO_TCP;
        foo.tuple.dst.u.tcp.port = htons(FOO_PORT);
        foo.mask.dst.protonum = 0xFFFF;
```

```
        foo.mask.dst.u.tcp.port = 0xFFFF;
        foo.help = foo_help;

        return ip_conntrack_helper_register(&foo);
}

static void __exit fini(void)
{
        ip_conntrack_helper_unregister(&foo);
}
```

4.4.6 Helper Modules pour NAT

Description Les Helper Modules pour NAT gèrent la partie spécifique à une application du NAT. Habituellement, ça inclue la manipulation de données à la volée : pensez à la commande 'PORT' en FTP, quand le client dit au serveur sur quel port se connecter. Donc le Helper Module pour NAT de FTP doit remplacer l'adresse IP et le port après la commande PORT dans connexion de contrôle FTP.

Si on fait du TCP, les choses deviennent un peu plus compliquées. La raison est que la taille du paquet peut changer (ex : FTP, la longueur de la chaîne de caractères représentant une adresse IP/Port n-uplet après la commande PORT a changé). Si on change la taille du paquet, on a un SYN/ACK de différence entre le coté gauche et le coté droit de la machine NAT. (ex : si on a étendu un paquet de 4 octets, on doit ajouter cet offset au numéro de séquence pour chaque paquets suivants)

NAT pour le paquet en relation doit être géré spécialement aussi. Prenez encore comme exemple FTP, quand tous les paquets qui arrivent de la connexion DATA doivent être NATés à l'adresse IP/port donnés par le client avec la commande PORT sur la connexion de contrôle, plutôt que de passer dans les tables normalement.

- un 'callback' pour le paquet qui a généré la connexion en relation (foo_help)
- un 'callback' pour tous les paquets en relation (foo_nat_expected)

Structures et Fonctions disponibles la fonction 'init()' de votre Helper Module pour NAT appelle 'ip_nat_helper_register()' avec un pointeur vers une 'struct ip_nat_helper'. Cette structure a les champs suivants :

list

Mettez la à '{ NULL, NULL }'

name

Un pointeur sur une chaîne de caractères contenant le nom du protocole.

flags

Un groupe d'option avec un ou plusieurs des flags suivants :
- IP_NAT_HELPER_F_ALWAYSAppelle le NAT helper pour chaque paquet, et pas seulement pour les paquets où le module de suivi de connexion a détecté une attente.
- IP_NAT_HELPER_F_STANDALONEDit au coeur de NAT que ce NAT helper n'a pas de module de suivi de connexions, mais qu'il a seulement un NAT helper.

me

>Un pointeur sur la structure de module. Initialisez ça à la macro 'THIS_MODULE'.

tuple

Une structure 'struct ip_conntrack_tuple' décrivant par quels paquets notre Helper Module pour NAT est intéressé.

mask

Une ‘struct ip_conntrack_tuple’ disant à Netfilter quels bits `tuple` sont valides.

help

La fonction d’aide qui est appelée pour chaque paquet qui match n-uplet+masque.

expect

La fonction d’attente appelée pour chaque premier paquet de la connexion.

C’est a peu près la même chose que d’écrire un helper de connexion.

Exemple de module de NAT Helper

```
#define FOO_PORT        111

static int foo_nat_expected(struct sk_buff **pksb,
                        unsigned int hooknum,
                        struct ip_conntrack *ct,
                        struct ip_nat_info *info)
/* Appellée des que l'on reçoit le premier paquet d'une connexion en relation.
   params:      pksb    le tampon de paquets
                hooknum Le hook duquel l'appel vient (POST_ROUTING, PRE_ROUTING).
                ct      les informations à propos de cette connexion (en relation).
                info    &ct->nat.info
   return value: Le verdict (NF_ACCEPT, ...)
{
        /* Change le port/adresse IP du paquet vers leur valeurs masqueradées
           (lu à partir de  master->tuplehash), pour le faire de la même manière,
           appelez ip_nat_setup_info, retournez NF_ACCEPT. */

}

static int foo_help(struct ip_conntrack *ct,
                    struct ip_conntrack_expect *exp,
                    struct ip_nat_info *info,
                    enum ip_conntrack_info ctinfo,
                    unsigned int hooknum,
                    struct sk_buff  **pksb)
/* Appellée pour chaque paquet où conntrack a détecté une attente
   params:      ct      La struct ip_conntrack de la connexion maîtresse.
                exp     La struct ip_conntrack_expect de l'attente causée par
                        le helper conntrack pour ce protocole.
                info    État (STATE) : related, new, established, ... )
                hooknum Le hook duquel l'appel vient (POST_ROUTING, PRE_ROUTING).
                pksb    Le tampon de paquets.
*/
{

        /* Extrait les informations à propos des futurs paquets en relation
           (vous pouvez partager l'information avec la fonction foo_help
           du suivi de connexions). Échange ip adresse/port avec leur
           valeur masqueradée, insère le n-uplet des paquets en relation */
}

static struct ip_nat_helper hlpr;
```

```
static int __init(void)
{
        int ret;

        memset(&hlpr, 0, sizeof(struct ip_nat_helper));
        hlpr.list = { NULL, NULL };
        hlpr.tuple.dst.protonum = IPPROTO_TCP;
        hlpr.tuple.dst.u.tcp.port = htons(FOO_PORT);
        hlpr.mask.dst.protonum = 0xFFFF;
        hlpr.mask.dst.u.tcp.port = 0xFFFF;
        hlpr.help = foo_help;
        hlpr.expect = foo_nat_expect;

        ret = ip_nat_helper_register(hlpr);
        return ret;
}

static void __exit(void)
{
        ip_nat_helper_unregister(&hlpr);
}
```

4.5 Comprendre Netfilter

Netfilter est relativement simple, et il est décrit de manière assez complète dans les sections précédentes. Cependant, il est nécessaire de temps en temps d'aller en dessous de ce que les infrastructure NAT ou ip_tables offrent, ou vous pourriez vouloir les remplacer complètement.

Une chose importante pour Netfilter (oui enfin...dans le future) est le 'caching'. Chaque snk a un champs 'nfcache' : un masque de bits décrivant quel champs du header on été examinés, et si oui ou non le paquet a été modifié. L'idée est que chaque hook de netfilter fait un OR (OU binaire) avec les bits qui le concernent, pour que plus tard on puisse écrire un système de cache qui serait suffisamment intelligent pour réaliser quand un paquet n'a pas besoin du tout de passer à travers netfilter.

Les bits les plus importants sont NFC_ALTERED, signifiant que la paquet a été modifié (c'est déjà utilisé par le hook NF_IP_LOCAL_OUT d'IPv4 pour re-router les paquets modifiés) et NFC_UNKNOWN, qui signifie que le caching ne doit pas être fait, parce que quelques propriétés qui ne peuvent pas être exprimées ont été examinées. Si vous ne savez pas, mettez simplement le flag NFC_UNKNOWN sur le champs 'nfcache' du skb à l'intérieur de votre hook.

4.6 Écrire de nouveaux modules pour netfilter/

4.6.1 Se brancher dans les hooks de Netfilter.

Pour recevoir/modifier des paquets à l'intérieur du kernel, vous pouvez simplement écrire un module qui enregistre un "hook Netfilter". C'est intéressant pour certaines choses. Les points effectifs sont spécifiques à chaque protocole, et définis dans les headers de protocole de netfilter, comme "netfilter_ipv4.h" par exemple.

Pour enregistrer et dé-enregistrer un hook netfilter, vous devez utiliser les fonctions 'nf_register_hook()' et 'nf_unregister_hook()'. Elles prennent chacune un pointeur sur une 'struct nf_hook_ops' qui a les champs suivants :

list

Initialisez la à '{ NULL, NULL }'.

hook

La fonction à appeler quand un paquet touche ce point de hook. Votre fonction doit retourner NF_ACCEPT, NF_DROP ou NF_QUEUE. Si elle retourne NF_ACCEPT, le hook suivant attaché à ce point va être appelé. Si elle retourne NF_DROP, le paquet est détruit. Si elle retourne NF_QUEUE, le paquet est queuté. Vous recevez un pointeur vers un pointeur de skb, donc vous pouvez remplacer totalement le skb si vous le souhaitez.

flush

Pour le moment inutilisé : fait dans l'idée de passer le hit de paquets quand le cache est vide. Peut ne jamais être implémenté : mettez le à NULL.

pf

La famille de protocole, par exemple 'PF_INET' pour IPv4.

hooknum

Le numéro du hook qui vous intéresse, par exemple 'NF_IP_LOCAL_OUT'.

4.6.2 S'occuper des paquets Queutés

Cette interface est pour le moment utilisée par ip_queue. Vous pouvez vous enregistrer pour gérer les paquets queutés pour un protocole donné. ça ressemble à l'enregistrement d'un hook, sauf que vous pouvez faire que le paquet soit bloqué pour un certain temps, et aussi que vous ne voyiez que les paquets pour lesquels un hook a répondu 'NF_QUEUE'.

Les 2 fonctions utilisées pour s'enregistrer en tant que partie intéressée par les paquets queutés sont : 'nf_register_queue_handler()' et 'nf_unregister_queue_handler()'. Les fonctions que vous enregistrez seront appelées avec le pointeur 'void *' que vous avez donné à 'nf_register_queue_handler()'.

Si personne n'est enregistré pour gérer un protocole, alors un verdict NF_QUEUE devient équivalent à un verdict NF_DROP.

Une fois que vous vous êtes enregistré pour recevoir les paquets queutés, ils commencent à arriver. Vous pouvez faire ce que vous voulez avec ces paquets, mais vous devez appeler 'nf_reinject()' quand vous en avez fini avec eux (ne faites pas simplement un kfree_skb() sur eux). Quand vous ré-injectez un skb, vous donnez en paramètre à la fonction : le skb, la structure 'struct nf_info' que le gestionnaire vous a donné, et un verdict : NF_DROP provoquera la destruction du paquet, NF_ACCEPT fera que l'itération à travers les hooks reprendra, NF_QUEUE fera qu'il seront queutés encore une fois, et NF_REPEAT fera que le hook qui a queuté le paquet soit reconsulté (attention aux boucles sans fin ! !).

Vous pouvez regarder dans le 'struct nf_info' pour obtenir des informations supplémentaires à propos du paquet, comme les interfaces et hooks qu'il a fréquenté.

4.6.3 Recevoir des commandes à partir du userspace.

Il est commun pour les composants de Netfilter de vouloir interagir avec le userspace. La méthode pour faire cela est d'utiliser le mécanisme 'setsockopt()'. Notez que chaque protocole doit être modifié pour appeler nf_setsockopt() avec un nombre setsockopt qu'il ne comprend pas (et nf_getsockopt() avec un nombre getsockopt qu'il ne comprend pas), et pour le moment seulement IPv4, IPv6, et DECnet on été modifiés.

En utilisant maintenant une technique familière, on enregistre une 'struct nf_sockopt_ops' en utilisant la fonction 'nf_register_sockopt()'. Les champs de cette structure sont :

list

Mettez le à '{ NULL, NULL }'.

pf
: La famille de protocole que vous gérez, ex : PF_INET.

set_optmin
: et

set_optmax
: Ceux la spécifient l'interval (exclusif) de nombre setsockopt gérés. Et donc, 0 et 0 signifie que vous n'avez pas de nombre setsockopt.

set
: C'est la fonction qui est appelée quand un utilisateur appelle une de vos setsockopts. Vous devriez vérifier dans cette fonction qu'il a la capabilité 'NET_ADMIN'.

get_optmin
: and

get_optmax
: Ceux-là spécifient l'interval (exclusif) de nombres getsockopt gérés. Et donc, 0 et 0 signifie que vous n'avez pas de nombre getsockopt.

get
: C'est la fonction qui est appelée quand un utilisateur appelle une de vos getsockopts. Vous devriez vérifier dans cette fonction qu'il a la capabilité 'NET_ADMIN'.

Les deux champs finaux sont utilisés en interne.

4.7 Gérer les paquets du userspace.

En utilisant la librairie 'libipq' et le module 'ip_queue', presque tout ce qui peut être fait au niveau du kernel peut maintenant être fait en userspace. Ça veut dire qu'avec une petite perte de vitesse, vous pouvez développer votre code entièrement en userspace. A moins que vous vouliez filtrer une grosse bande passante, vous devriez trouver cette approche supérieure à la modification de paquets dans le kernel.

Dans les débuts de Netfilter, j'ai prouvé ça en portant une version embryonique de iptables en userspace. Netfilter vous ouvre la porte pour que plus de gens puissent écrire des modificateurs de paquets efficaces, dans n'importe quel langage.

5 Porter les modules de filtrage de paquets 2.0 et 2.2 vers 2.4.xx

Regardez le fichier ip_fw_compat.c pour une couche de traduction simple qui devrait rendre le portage assez simple.

6 Les hooks de netfilter pour les auteurs de driver de tunnels

Les auteurs de drivers de tunnels (ou encapsulation) devraient suivre deux règles simples avec les noyaux 2.4 (comme le font les drivers qui sont déjà dans le noyau comme net/ipv4/ipip.c) :

- Relâchez le skb->nfct si vous allez rendre le paquet inreconnaissable (par exemple décapsulation/encapsulation). Vous n'avez pas besoin de faire ça si vous déroulez le nouveau paquet dans un *nouveau* skb, mais si vous voulez le faire dans le même, alors il faut relâcher.
  Sinon : le code de NAT va utiliser les anciennes informations de suivit de connexion pour modifier le paquet, ce qui aura de mauvaises conséquences.

– Faites en sorte que le paquet encapsulé va par le hook LOCAL_OUT, et que le paquet décapsulé passe par le PRE_ROUTING hook (la plupart des tunnels utilisent ip_rcv(), qui fait déjà ça pour vous)
Sinon : L'utilisateur ne sera pas capable de filtrer les paquets avec votre tunnel comme ils l'entendent.

La forme canonique pour mettre en place le premier conseil est d'insérer du code comme celui qui suit, avant de dérouler le paquet :

```
        /* Dire a netfilter que ce paquet n'est plus le même
          que celui d'avant! */
#ifdef CONFIG_NETFILTER
        nf_conntrack_put(skb->nfct);
        skb->nfct = NULL;
#ifdef CONFIG_NETFILTER_DEBUG
        skb->nf_debug = 0;
#endif
#endif
```

D'habitude, tout ce que vous avez à faire pour le second, est de trouver où le paquet nouvellement encapsulé va avec "ip_send()", et de remplacer ça avec quelque-chose comme :

```
        /* Envoie le "nouveau" paquet à partir de localhost */
        NF_HOOK(PF_INET, NF_IP_LOCAL_OUT, skb, NULL, rt->u.dst.dev, ip_send);
```

Suivre ces règles veut dire que la personne qui met en place une règles de filtrage sur la machine qui tunnelle verra la séquence suivante pour un paquet qui est tunellé :

1. FORWARD hook : le paquet normal (eth0 -> tunl0)
2. LOCAL_OUT hook : le paquet encapsulé (vers eth1)

Et pour le paquet réponse :

1. LOCAL_IN hook : paquet réponse encapsulé (en provenance de eth1)
2. FORWARD hook : paquet réponse (eth1 -> eth0)

7 La suite de tests

Au sein du dépôt CVS vit une suite de tests : plus la liste la suite de tests couvre de points, plus vous avez l'assurance que le changement apporté ne casse pas autre chose. Les tests triviaux sont au moins aussi importants que les tests difficiles : ce sont les tests triviaux qui simplifient les tests complexes (puisque vous savez que le test trivial fonctionne avant même de lancer le test complexe).

Les tests sont simples : ce sont juste des scripts shell dans le répertoire 'testsuite/' qui sont supposés réussir. Les tests sont lancés par ordre alphabétiques, donc '01test' serait lancé avant '02test'. Pour l'instant il y a 5 répertoires de tests :

00netfilter/
Tests généraux du canevas netfilter.

01iptables/
Tests d'iptables.

02conntrack/
Les tests couvrant le suivi de connexions.

03NAT/
Les test couvrant la NAT.

04ipchains-compat/
Les tests couvrant la compatibilité avec ipchains/ipfwadm

A l'intérieur du répertoire 'testsuite/' il y a un script appelé 'test.sh'. Il configure 2 interfaces bidons ('tap0' et 'tap1'), allume la fonction de forwarding, et enlève tous les modules netfilter du kernel. Ensuite il parcours les répertoires décrit au dessus, et lance chacun de leur script 'test.sh' jusqu'à ce que l'un d'entre eux rate.'-v' veut dire d'afficher chaque test au moment ou il est effectué, et optionnellement le nom du test : si ce dernier est donne en paramètre, il va passer tous les tests pour arriver directement au test demandé.

7.1 Écrire un test

Créez un nouveau fichier dans le répertoire approprié : essayez de donner un nombre à votre script pour qu'il soit lancé au bon moment. Par exemple, avant de tester si le suivi de ICMP-reply fonctionne, on doit d'abord tester que les paquets ICMP sortant sont suivis correctement.

Il est habituellement mieux de créer plus de petit tests atomiques plutôt qu'un gros test qui couvre tout. Chaque test peut alors couvrir un aspect particulier, et mieux isoler le problème immédiatement pour le gens qui lancent la suite de tests.

Si quelque chose ne se passe pas bien, faite simplement un 'exit 1' qui va causer le ratage du test. Si il s'agit de quelque chose attendu à rater exprès, vous devriez imprimer un message unique. Votre test doit terminer par 'exit 0' si tout ce passe bien. Vous devez vérifier le succès de **chaque** commande, soit en utilisant 'set -e' au début du script, ou en ajoutant '|| exit 1' à la fin de chaque commande.

Les fonctions d'aide 'load_module' et 'remove_module' peuvent être utilisées pour charger des modules : vous ne devriez jamais vous appuyer sur la fonction d'auto chargement des modules dans la suite de tests, à moins que ce ne soit ce que vous cherchez à tester.

7.2 Variables et Environnement

Vous avez 2 interfaces pour jouer : tap0 et tap1. L'adresse IP des interfaces sont dans `$TAP0` et `$TAP1` respectivement. Elles ont toutes les deux un netmask de '255.255.255.0', et leur adresse network sont dans `$TAP0NET` and `$TAP1NET` respectivement.

Il y a un fichier temporaire vide dans $TMPFILE. Il sera effacé à la fin de votre test.

Votre script sera lancé du répertoire 'testsuite', où qu'il soit. Et donc, vous devez accéder aux utilitaires (comme iptables) en utilisant un chemin commençant pas '../userspace'

Votre script peut imprimer plus d'information si $VERBOSE est mis (ce qui veut dire que l'utilisateur a spécifié '-v' sur la ligne de commande)

7.3 Outils utiles

Il y a quelques outils de la suite de tests qui sont utiles dans le répertoire "tools" : chacun d'entre eux retourne une valeur non nulle si il y a eu un problème.

7.3.1 gen_ip

Vous pouvez générer un paquet IP en utilisant 'gen_ip', qui sort un paquet IP vers la sortie standard. Vous pouvez donc envoyer ces paquet IP en redirigeant la sortie standard vers /dev/tap0 ou /dev/tap1 (ces fichiers seront créer après le 1er lancement de la suite de tests si ils n'existaient pas déjà).

gen_ip est un programme simpliste qui est très très rigoureux sur l'ordre des paramètres que vous lui donnez :

FRAG=offset,length
: Génère le paquet, et ensuite le transforme en un fragment à l'offset donné avec la longueur donnée.

MF
: Met le bit 'More Fragments' sur le paquet.

MAC=xx :xx :xx :xx :xx :xx
: Définit la MAC adresse source du paquet.

TOS=tos
: Définit le 'TOS' du paquet (0 <= TOS <= 255).

Ensuite viennent les arguments obligatoires :

source ip
: Adresse IP source du paquet.

dest ip
: Adresse IP destination du paquet.

length
: Longueur totale du paquet, en comptant les headers.

protocol
: Numéro du protocole encapsulé dans le paquet, ex : 17=UDP.

Ensuite, les argument dépendent du protocole : pour UDP (17), il y a le port source et le port destination. Pour ICMP (1), il u a le type et le code ICMP du paquet. Si le type est 0 ou 8 (pong ou ping) alors deux paramètres additionnels (l'ID et le champs de séquence) sont requis. Pour TCP (6), le port source et le port destination, et les flags TCP ("SYN", "SYN/ACK", "ACK", "RST" or "FIN") sont requis. Il y a 3 arguments optionnels : "OPT=" suivi d'une liste (les éléments sont séparés par une virgule) d'options, "SYN=" suivi d'un numéro de séquence, et "ACK=" suivi par un numéro de séquence. Finalement, l'argument optionnel "DATA" indique que les données du paquet sont à remplir avec le contenu de l'entrée standard.

7.3.2 rcv_ip

Vous pouvez voir les paquets IP en utilisant 'rcv_ip', qui imprime la ligne de commande aussi proche que possible de la commande qui a servit à générer le paquet (les fragments sont une bonne exception).

C'est extrêmement utile pour analyser les paquets. Elle prend 2 arguments obligatoires :

wait time
: Le temps (en secondes) maximum pour attendre le paquet à partir de l'entrée standard.

iterations
: Le nombre de paquets à recevoir.

Il y a un argument optionnel, "DATA" qui force les données du paquets à être imprimées sur la sortie standard, après le header du paquet.

La façons standard d'utiliser 'rcv_ip' dans un script shell est la suivante :

```
# Mets en place le contrôle des taches, pour que l'on puisse utiliser
# '&' dans le script
set -m

# Attend 2 secondes un paquet sur tap0
```

```
../tools/rcv_ip 2 1 < /dev/tap0 > $TMPFILE &

# Soyons sûr que rcv_ip a été démarré.
sleep 1

# Envoyer un paquet ICMP ping
../tools/gen_ip $TAP1NET.2 $TAP0NET.2 100 1 8 0 55 57 > /dev/tap1 || exit 1

# Attend rcv_ip
if wait %../tools/rcv_ip; then :
else
    echo rcv_ip failed:
    cat $TMPFILE
    exit 1
fi
```

7.3.3 gen_err

Ce programme prend un paquet (comme généré par gen_ip, par exemple sur l'entrée standard, et en fait une erreur ICMP)

Il prend 3 arguments : un adresse IP source, un type et un code. L'adresse IP destination va être remplies avec l'adresse IP source du paquet donné sur l'entrée standard.

7.3.4 local_ip

Ce programme prend un paquet à partir de l'entrée standard et l'injecte dans le système à partir d'une 'raw socket'. Cela fait semblant d'être un paquet généré localement (comparé à un paquet passé par une des interfaces ethertap qui semble être des paquets générés par une interface étrangère).

7.4 Conseils pratiques

Tous les utilitaires assument qu'ils peuvent tout faire en lecture ou écriture : c'est tout à fait vrai avec les interfaces ethertap, mais n'est pas forcement vrai si vous faites des bidouilles avec les pipes.

'dd' peut être utilisé pour couper des paquets, dd a une option 'obs' ("output block size"=taille de bloc de sortie) qui peut être utilisé pour écrire les paquets en une seule écriture.

Vérifier les tests de succès d'abord, ex : testez que les paquets sont bien bloqués. Testez d'abord que les paquets passent bien d'abord, et **ensuite** testez que les paquets que vous voulez voir être bloqués sont bel et bien bloqués. Sinon, une erreur quelconque aurait tres bien pu bloquer le paquet...

Essayez d'écrire des tests exacts, et ne pas faire des 'tests aléatoires et voir ce qui arrive'. Si un test exact ne marche pas, c'est toujours bon à savoir. Si un test aléatoire ne marche pas une fois, ça n'aide peut à trouver l'erreur.

Si le test plante sans message d'erreur, vous pouvez ajouter un '-x' à la première ligne du script (ex : '#! /bin/sh -x') pour voir quelles commandes sont lances.

Si un test plante de manière aléatoire, vérifier qu'il n'y ai pas d'interférences avec d'autres interfaces. Éteignez toutes vos interfaces externes. Étant sur le même réseau que Andrew Tridgell, j'ai l'habitude d'être inondé avec des paquets broadcast de Windows par exemple.

8 Motivation

Comme j'ai développé ipchains, j'ai réalisé (pendant l'un de ces moments flash-tout-blanc-pendant-que-j-attend dans un restaurant chinois à Sydney) que le filtrage de paquets était fait au mauvais endroit. Je n'arrive plus à le retrouvé maintenant, mais j'ai envoyé un émail à Alan Cox, qui a en quelque sorte répondu 'pourquoi tu ne finis pas ce que tu as commencé d'abord, même si tu as probablement raison'. Version courte : le pragmatisme allait gagner devant 'The Right Thing' ('La Bonne Methode').

Après avoir fini ipchains, qui devait être initialement une modification mineure des parties kernel de ipfwadm, et qui a fini par être une ré-écriture quasi-totale de la chose, et après avoir écrit le HOWTO, je me suis rendu compte à quel point toute la communauté Linux était confuse avec des choses comme le filtrage de paquets, NAT et port forwarding, et les choses du même genre.

C'est la joie de fournir votre propre support : vous avez une idée plus précise de ce que les utilisateurs cherchent à faire, et ce avec quoi ils ont des difficultés. Le Free Software est le plus gratifiant quand il est dans les mains de la plupart des utilisateurs (c'est le but hein ?), et ça veut dire le simplifier. L'architecture, et pas la documentation, était le problème principal.

Donc j'avais l'expérience avec le code d'ipchains, et une bonne idée de ce que les gens cherchaient à faire. Mais il y avait deux problèmes :

D'abord, je n'avais pas envie de me remettre à la sécurité. Être un consultant en sécurité est toujours une bataille dans votre cerveau entre votre conscience et votre porte-monnaie. A un niveau de base, vous vendez le sentiment de sécurité, qui est à l'opposé de la vrai sécurité. Peut être travailler dans un corps militaire, où ils comprennent la sécurité, ça aurait été diffèrent.

Le deuxième problème est que les nouveaux utilisateurs ne sont plus les seuls à poser problème, un nombre grandissant de gros ISPs utilisent ce truc. J'avais besoin d'informations exactes en provenance de ces gens, si c'était pour aménager la chose pour les utilisateurs de demain.

Ces problèmes ont été résolus quand j'ai rencontré David Bonn, de chez WatchGuard, à Usenix en 1998. Ils cherchaient un codeur pour le kernel Linux, à la fin on s'est mis d'accord que j'allais passer un mois dans leurs bureaux à Seattle, et on verra si on pourra s'arranger sur un accord ou ils sponsoriseraient mon travail. Le prix a été plus que ce que j'avais demandé, et donc je n'ai pas souffert de perte de salaire. Ce qui veut dire que je n'avais pas à me soucier à faire du consulting en externe pour un bon moment.

Travailler avec WatchGuard m'a donné une exposition avec de large clients, ce qui était ce de quoi j'avais besoin. Et étant indépendant d'eux, ça me permettait d'aider les autres (ex : les compétiteurs de WatchGuard) de manière égale.

Donc j'aurais pu avoir écrit netfilter, et porté ipchains par dessus, et avoir fini mon boulot. Malheureusement, ça aurait laissé tout le code de masquerading dans le kernel : et faire en sorte que le masquerading soit indépendant du filtrage était un but très important.

Aussi, mon expérience avec la fonction 'interface-address' d'ipfwadm (celle que j'ai retiré d'ipchains), m'a enseigné qu'il n'y avait aucun espoir d'enlever simplement le code de masquerading du kernel et d'attendre que quelqu'un le porte sous netfilter pour moi.

Donc j'avais besoin d'avoir au moins autant de fonctionnalité que le code présent, et de préférence plus, pour encourager des utilisateurs marginaux à être les premiers testeurs. Cela veut dire remplacer les fonctions de proxy transparent (enfin !), de masquerading et de port-forwarding. En d'autres mots, avoir une couche de NAT.

Même si j'avais décidé de porter la couche de masquerading existante, à la place d'écrire une couche de NAT générique, le code de masquerading montrait son âge, et son manque de maintenance. Vous voyez, il n'y avait pas de mainteneur du code de masquerading et ça se voit ! Il semblerait que les utilisateurs sérieux n'utilisent pas généralement la masquerade, et il y a peu d'utilisateur à la maison qui serait prêt à maintenir le code de masquerading. Des gens courageux et braves comme Juan Ciarlante envoyaient des patchs pour corriger des bugs de temps en temps, mais ça avait atteint le stage ou (après avoir été étendu-patché-et-ré-étendu) ou une ré-écriture était nécessaire.

Notez s'il vous plaît que je n'étais pas celui qui avait écrit le système de NAT : je n'utilisait pas la masquerading du tout, et je n'avais pas étudié le code existant à cette époque. C'est pourquoi ça m'a pris plus longtemps que ça n'aurait du en prendre. Mais le résultat est assez bien, à mon avis, et j'ai vraiment appris énormément. Sans aucun doutes la seconde version sera encore meilleure, une fois que l'on verra comment les gens l'utilisent.

9 Remerciements

Merci à ceux qui ont aidé, spécialement Harald Welte pour avoir écrit la section Helpers de protocoles.