AVRIL/MAI 2009 N°41

GNU LINUX MAGAZINE / FRANCE HORS-SÉRIE

Administration et développement sur systèmes UNIX

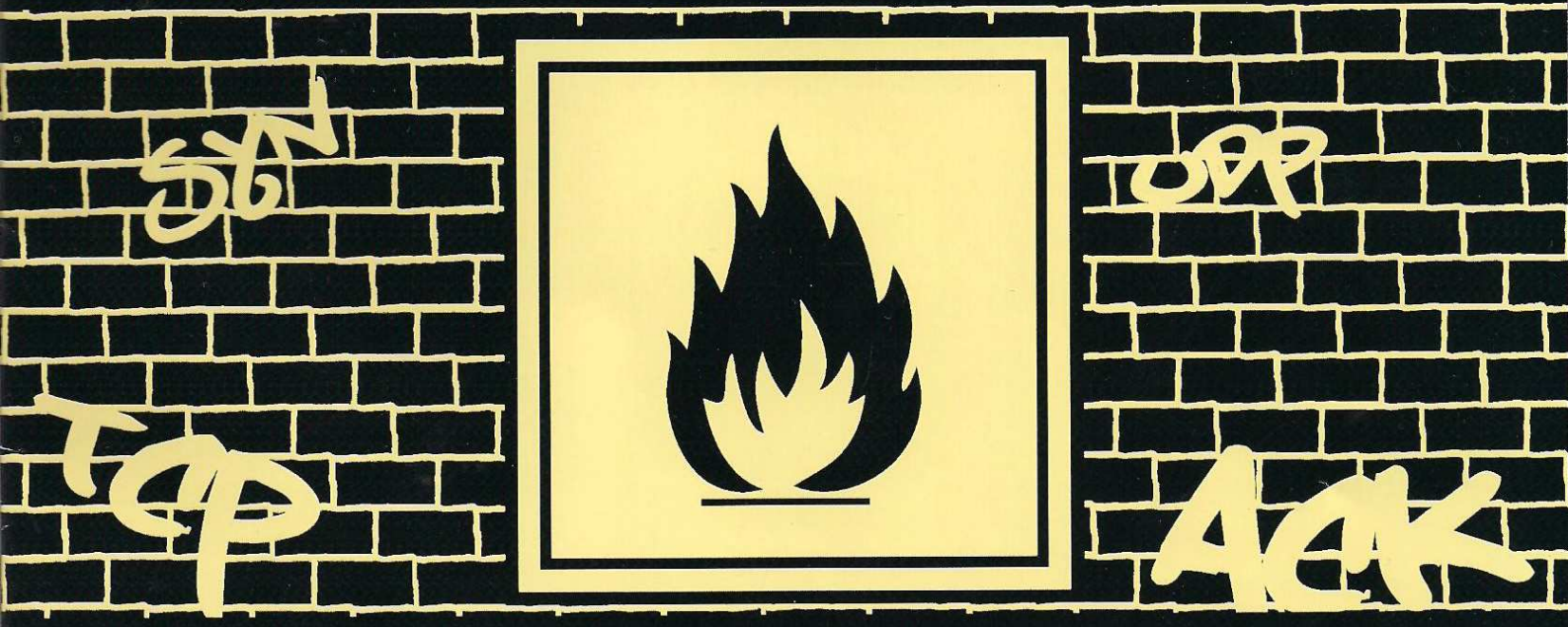
BIEN DÉBUTER

INTRODUCTION SANS DOULEUR AUX RÈGLES DE FILTRAGE AVEC NETFILTRER ET IPTABLES

(p. 20)

CONFIGUREZ ET OPTIMISEZ VOTRE FIREWALL

GRÂCE AUX FONCTIONNALITÉS DU NOYAU ET APPLICATIONS OPENSOURCE



SURVEILLANCE

Gardez une trace des connexions avec ULOG/NFLOG

(p. 30)

PEOPLE

Interviews exclusives des développeurs du noyau : Patrick McHardy, Pablo Neira et Dave Miller

(p. 09)

EXTENSION

Créez des applications de filtrage en dehors du noyau grâce aux cibles QUEUE/NFQUEUE

(p. 67)

L 15066 - 41 H - F: 6,50 € - RD

France Métro : 6,50 € - DOM : 7,00 €
TOM Surface : 950 XPF
POL. A : 1400 XPF
BEL/PORT.CONT : 7,50 €
CH : 13,8 CHF
CAN : 13 $CAD
MAR : 75 MAD

Sommaire

Édito

Netfilter est l'ensemble du code qui fournit au noyau Linux les fonctions de pare-feu, de traduction d'adresse et d'historisation du trafic réseau. L'utilisation la plus populaire de ces fonctionnalités est, sans le moindre doute, la mise en place d'une passerelle NAT permettant d'offrir l'accès Internet à un LAN. Le pare-feu, bien entendu, est également là pour protéger les serveurs, le LAN et toute infrastructure réseau, des attaques venant de l'extérieur comme de l'intérieur.

Cependant, se limiter à une telle utilisation est à la frontière de l'injure à l'égard du travail de développeurs comme Dave Miller et de nombreux autres.

Ce hors-série intègre, bien entendu, l'incontournable introduction à Netfilter et à l'utilisation de la commande `iptables`, mais il va bien plus loin. Je vous propose de partir à la découverte d'un univers riche et passionnant. Une quête qui vous conduira dans des contrées magiques, à l'intérieur même du fonctionnement du noyau et plus particulièrement des fonctionnalités de filtrage.

Vous découvrirez ainsi toute l'étendue des possibilités qui s'offrent à vous, les travaux effectués, les principes de fonctionnement, mais aussi les extensions possibles. Plus qu'un hors-série, ce magazine se veut être un point de départ pour une utilisation optimale des fonctionnalités mises à disposition par Netfilter.

Je tiens à remercier les auteurs des articles qui peuplent les pages qui suivent : Pierre Chifflier, Éric Leblond, Gwenaël Rémond, Victor Stinner et Sébastien Tricaud. J'ai pris énormément de plaisir à composer ce hors-série avec eux et, tout comme vous, je l'espère, à lire et comprendre tout ce qu'il regroupe.

Enfin, vous noterez la présence, au centre du magazine, d'un poster de promotion pour OpenOffice.org par le groupe de travail « sensibilisation » de l'April. En tant qu'utilisateurs d'OpenOffice.org et d'acteurs du logiciel libre, nous considérons qu'il est important d'informer les utilisateurs d'applications propriétaires. Nous, tout comme les autres groupes et associations qui soutiennent ce projet, vous invitons donc à utiliser ce poster et ainsi à faire connaître OpenOffice.org.

Sur ce, je vous donne rendez-vous au 16 mai prochain pour un nouveau hors-série dont le contenu (mais, je n'en dirai pas plus) sera sans précédent...

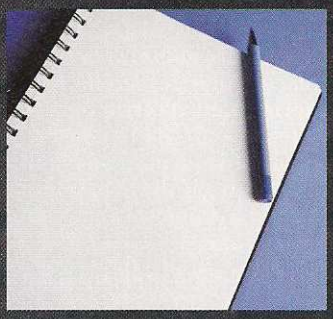
Denis Bodor

Gnu/Linux Magazine France Hors-série
est édité par Diamond Editions
B.P. 20142 - 67603 Sélestat Cedex
Tél. : 03 88 58 02 08
Fax : 03 88 58 02 09
E-mail : lecteurs@gnulinuxmag.com
Service commercial : abo@gnulinuxmag.com
Sites : www.gnulinuxmag.com
www.ed-diamond.com
Directeur de publication : Arnaud Metzler
Rédacteur en chef : Denis Bodor

Secrétaire de rédaction : Véronique Wilhelm
Relecture : Dominique Grosse
Conception graphique : Fabrice Krachenfels
Responsable publicité : Tél. : 03 88 58 02 08
Service abonnement : Tél. : 03 88 58 02 08
Impression : VPM Druck Allemagne
Distribution France :
(uniquement pour les dépositaires de presse)
MLP Réassort :
Plate-forme de Saint-Barthélemy-d'Anjou.
Tél. : 02 41 27 53 12
Plate-forme de Saint-Quentin-Fallavier.
Tél. : 04 74 82 63 04
Service des ventes :
Distri-médias :
Tél. : 05 61 72 76 24
IMPRIMÉ en Allemagne - PRINTED in Germany
Dépôt légal : À parution /N° ISSN : 1291-78 34
Commission paritaire : K78 976
Périodicité : Bimestrielle
Prix de vente : 6,50 €

REPORTAGE

Convention 2008 des développeurs



Le sixième workshop Netfilter s'est tenu cette année à Paris début octobre. Les développeurs importants de Netfilter se sont retrouvés pendant une semaine pour présenter leur travail et décider des orientations du travail de l'année. Voici le compte-rendu de cet évènement bien chargé.

Auteur

- Éric Leblond

1 Une semaine intense

Comme presque tous les ans depuis 2001, la convention Netfilter (ou *Workshop*) a de nouveau réuni les principaux développeurs Netfilter. La réunion était organisée cette année par la société INL, éditrice de NuFW et de l'*appliance* EdenWall.

Réitérant l'expérience de Séville en 2005, la convention a commencé par une journée ouverte aux utilisateurs. Cette journée ainsi que les suivantes se sont déroulées dans les locaux de l'ESIEA, qui propose, entre autres, le Mastère Spécialisé Sécurité de l'Information et des Systèmes. L'assistance, de bon niveau technique, a pu bénéficier de présentations réalisées tant par des développeurs que par des utilisateurs intensifs de Netfilter. Les deux jours suivants ont été deux jours de présentations entre développeurs où le travail de l'année et les prospectives ont été évoqués. Enfin, la semaine s'est terminée par deux jours de développement en commun où chacun a pu profiter de la présence des autres pour initier certains codes et obtenir des réponses rapides à des questions pointues.

2 Les journées utilisateurs

La journée a commencé par une présentation par Stephen Hemminger de l'interface en ligne de commande (CLI) de Vyatta. Assez peu liée à Netfilter, elle a néanmoins mis en lumière l'approche de Vyatta dans la réalisation d'une CLI pour routeur. L'intérêt de leur approche est principalement d'utiliser les capacités de bash pour en transformer complètement son comportement jusqu'à ce qu'il ressemble à une interface de routeur typique (semblable à celle de certains routeurs propriétaires).

L'exposé suivant était d'un niveau technique assez intense. David Miller, mainteneur de la couche réseau et du port sparc de Linux, nous a en effet présenté ses travaux sur les périphériques réseau multiqueues. Ces nouvelles cartes réseau offrent la possibilité de gérer de multiples files d'émissions parallèles. Avec la multiplication des cœurs des CPU, cette fonctionnalité devrait entraîner des améliorations notables de performance, puisque la file unique d'émission d'une carte réseau ne sera plus le goulot d'étranglement devant lequel les CPU viennent entasser leurs données. En termes de sécurité, ce système est également intéressant puisqu'il est possible d'aiguiller les flux suivant certaines propriétés réseau (typiquement des filtres). Facile alors d'imaginer des systèmes d'accélération matérielle de certains flux.

Pour terminer la matinée, Jesper Dangaard Brouer est venu présenter son travail chez l'opérateur Danois ComX. Ce dernier est un opérateur grand public qui propose un système de filtrage IP paramétrable à ses clients. À l'inverse de certains opérateurs français, le filtrage n'est pas réalisé sur les routeurs à domicile, mais plutôt sur les points de collectes locaux. Cela a pour conséquence des pare-feu gérant quelques centaines voire quelques milliers de domiciles. Chaque personne pouvant avoir des règles spécifiques, le nombre de règles *iptables* sur chaque pare-feu est donc extrêmement important (souvent supérieur à 50000). Comme Netfilter vérifie séquentiellement les règles, le nombre de tests pour chaque paquet est bien trop élevé et il provoque un ralentissement important du système. Jesper a donc travaillé à l'optimisation de Netfilter et d'iptables pour ce type de jeu de règles. Les structures de données utilisées dans iptables n'étaient pas performantes lors de l'insertion de nouvelles règles sur des jeux de règles massifs et ses modifications de code se sont principalement portées sur l'utilisation de structures adaptées à ce genre d'environnement.

Le problème de la vitesse de décision a été réglé en classant les règles. Au lieu de parcourir linéairement l'ensemble des règles, le jeu de règles est construit de manière à envoyer des morceaux de réseaux arbitraires dans une sous-chaîne qui est elle-même ensuite découpée en prenant des réseaux plus petits. En utilisant ce genre de découpage arborescent (voir Figure 1.), on passe d'une complexité

proportionnelle au nombre de règles total à une complexité proportionnelle au nombre de règles par client, auquel on ajoute la profondeur de l'arbre.

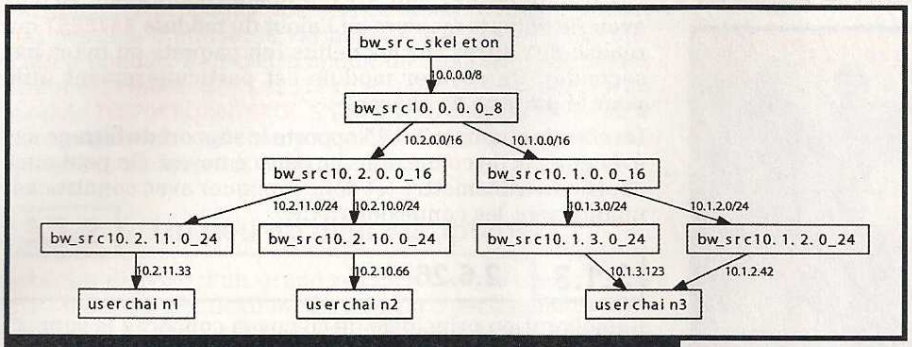


Figure 1 : Principe du découpage mis en place par Jesper Dangaard Brouer

La représentation du résultat global est visible Figure 2. Au vu de la complexité du graphe, on comprend que Jesper ait développé un module perl (`Iptables::SubnetSkeleton`) pour générer automatiquement les règles iptables construisant l'arbre.

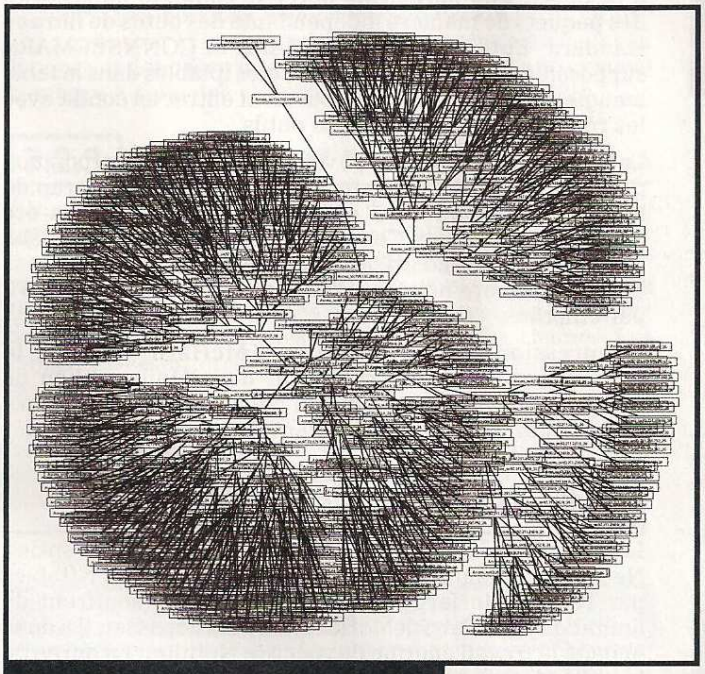


Figure 2 : Diagramme d'aiguillage des règles

Après la pause repas, Pierre Chifflier d'INL nous a présenté ses travaux sur le pare-feu météo, grande avancée en termes de sécurité. Grâce à ses développements autour des bibliothèques de Netfilter, il a pu écrire en quelques lignes une application codée en Perl qui autorise ou non la connexion réseau en fonction de la météorologie de la destination dudit paquet... Comme un article est consacré à ses travaux dans ce hors-série, je ne m'étends pas davantage sur ce sujet.

Luc Bourdot du ministère de l'Éducation nationale nous a ensuite présenté le projet Eole dont il est le responsable depuis 2000. De plus en plus connu par le grand public, ce projet financé et réalisé par le ministère de l'Éducation nationale développe des appliances logicielles, appelées « modules » (serveur éducatif basé sur Samba, client léger, pare-feu IP ou authentifiant avec NuFW, serveur VPN, console d'administration) qui sont déployés massivement dans les écoles, collèges et les lycées français. On compte ainsi par exemple plus de 6000 installations du pare-feu Amon basé sur Netfilter. Les modules d'Eole ont pour particularité d'être facilement installables (0 question) ce qui permet leur déploiement sans connaissance informatique. Grâce à un mécanisme d'enregistrement sur un serveur central (module Sphinx) ou un questionnaire pour le mode autonome, il est possible d'instancier le serveur et de le rendre opérationnel facilement. La configuration et les règles du pare-feu sont générées à partir d'un modèle propre à chaque académie (dérivé lui-même d'un des modèles nationaux) au moment de l'instanciation. Un professeur, responsable de l'informatique dans un établissement, n'a donc pas besoin de connaissances spécifiques pour mettre en place la solution. L'édition des modèles de règle de filtrage se fait grâce à une interface graphique (ERA). Ainsi, même le responsable académique n'a nul besoin d'une connaissance approfondie de Netfilter.

J'ai ensuite présenté le logiciel Ulogd2 sur lequel je travaille maintenant depuis plus d'un an. Ce successeur d'Ulogd enrichit les capacités de ce dernier en offrant une modularité accrue et un support des événements venant du suivi de connexions. C'est donc une solution de journalisation pour pare-feu complète. Un article étant consacré à ce logiciel dans ce hors-série, je vous laisse vous y référer.

Jozsef Kadlecsik a clôturé la journée en parlant de son projet **ipset**. Ce logiciel ajoute à iptables le support d'ensembles complexes (blocs d'adresses IP, blocs de réseaux, ensemble IP:port) tout en garantissant un filtrage rapide et efficace. Un des nombreux exemples d'utilisation est la constitution de listes d'adresses de spammeurs. En constituant de telles listes, il est alors possible de ralentir leur trafic en leur appliquant la cible `TARPIT` qui amène la bande passante d'une connexion TCP à 0 :

```
ipset -N spammers iptree --timeout $((60*60*24*7))
iptables -A FORWARD -d <honeypot> -p tcp --dport 25 \
        j SET addset spammers src
iptables A FORWARD -p tcp --dport 25 \
        -m set --set spammers src -j TARPIT
```

Les objectifs de Jozsef pour `ipset` sont de passer les échanges *userspace* noyau dans un mode nfnetlink. Utilisant les mêmes fondements que les autres sous-systèmes de Netfilter, il pourra alors prétendre à l'inclusion dans la distribution officielle de Netfilter et donc dans celle du noyau. Selon toute vraisemblance, cela devrait se produire avant la deuxième moitié de l'année 2009.

Cette journée d'un bon niveau technique a rassemblé développeurs et utilisateurs dans une ambiance positive et détendue. Des échanges informels entre utilisateurs et développeurs ont eu lieu tout au long de l'événement.

3 Les journées développeurs

Les journées développeurs ont réuni une vingtaine de personnes venant d'un peu partout. Parmi les nations les plus représentées, c'est l'Allemagne qui arrive en tête avec 5 développeurs. La France est dans la moyenne avec 3 développeurs à égalité avec la Hongrie ou les États-Unis.

Figure 3 : Photo de groupe, avec de gauche à droite et de haut en bas :

- Stephen Hemminger, Henrik Nordström, Harald Welte, Holger Eitzenberger, Sanjay Rao, Samir Bellabes, Pablo Neira Ayuso

- Nishit Shah, Jimit Mahadevia, Balazs Scheidler, Jesper Dangaard Brouer, Jozsef Kadlecsik, Pierre Chifflier, Jan Engelhart

- David Miller, Krisztián Kovács, Patrick McHardy, Moritz Grimm, Eric Leblond

3.1 Évolution de Netfilter de septembre 2007 à 2008 (Patrick McHardy)

La première journée développeurs a commencé par une présentation de Patrick sur les évolutions de Netfilter au cours de l'année écoulée. Il nous a décrit dans le détail les changements majeurs intervenus.

3.1.1 2.6.24

Le noyau Linux 2.6.24 a apporté quelques fonctionnalités intéressantes comme l'inclusion du module de filtrage `time` qui fait ce que son nom indique. Déjà disponible comme extension depuis longtemps, il fait maintenant partie de la branche officielle.

La couche ctnetlink responsable des interactions entre le suivi de connexion et l'espace utilisateur a connu des évolutions intéressantes. En premier lieu, il est devenu possible de créer des connexions relatives à la volée. Plus anecdotique, mais bon à savoir, l'identifiant numérique unique d'une connexion a été supprimé. Cela peut toutefois poser des soucis pour identifier une connexion, mais un entier moins strict (boucle possible) a été mis en place.

Le suivi de connexions se voit lui ajouter du code pour supporter la défragmentation en IPv6. Ceci achève le support d'IPv6 au niveau de Netfilter.

3.1.2 2.6.25

Ce noyau apporte bon nombre de changements. On notera principalement un accroissement des performances au niveau du *conntrack* grâce notamment à l'utilisation de RCU au lieu d'utiliser de simples *lock*. Le gain de performance est notable, Jesper Dangaard a ainsi observé une baisse de 25% de la charge CPU sur ses pare-feu. Parmi les autres modifications, on peut citer la réécriture du module de suivi de connexion pour SIP qui semble maintenant ne plus avoir de défauts majeurs ou l'ajout du module `RATEEST` qui réalise des filtres sur les débits (en paquets ou bytes par seconde). Ce dernier module est particulièrement utile pour le partage de charge.

Du côté de ctnelink, 2.6.25 apporte le support du filtrage des événements de connexions en espace noyau. On peut ainsi choisir de transmettre (et donc répliquer avec conntrackd) uniquement les connexions TCP.

3.1.3 2.6.26

L'amélioration principale de ce noyau concerne le support de la traduction d'adresse pour des protocoles rares comme UDP-LITE, SCTP, DCCP.

3.1.4 2.6.27

L'ajout principal de ce noyau est une nouvelle table dédiée à SELinux. Elle permet de gérer le contexte de sécurité des paquets de manière indépendante des outils de filtrage standard. L'utilisation de SECMARK et CONNSECMARK supposait en effet d'ajouter des règles iptables dans la table mangle, règles iptables qui pouvaient entrer en conflit avec les règles posées par d'autres outils.

Le suivi de connexions TCP se voit amélioré par l'incorporation d'un patch permettant de terminer de manière plus rapide (5 min) une connexion où des paquets n'ont pas été confirmés par le pair. Ceci permet de passer outre le délai de destruction standard (au bout de 5 jours) des connexions TCP. La dernière nouveauté concerne le support de SCTP par ctnetlink.

L'impression globale de Patrick McHardy est que le développement s'est focalisé sur des détails au lieu de porter sur des modifications du cœur de Netfilter comme l'an dernier. Les choses semblent donc en meilleur état.

3.2 Nftables (Patrick McHardy)

Le morceau de bravoure de cette sixième convention Netfilter a indéniablement été la présentation de Nftables par Patrick McHardy. Iptables et Netfilter souffrent de limitations que Patrick McHardy a voulu dépasser. Il a donc engagé le travail énorme de réécrire Netfilter. La durée de la présentation a été à la hauteur du travail, puisqu'elle a duré toute une après-midi.

3.2.1 Les problèmes Iptables et Netfilter

Iptables souffre de certains problèmes. Le point le plus critique concerne la gestion des modifications de règles où un *dump* complet du jeu de règles est réalisé à chaque appel d'iptables. Ceci entraîne évidemment un coût important. Moins critique, mais gênant pour l'utilisateur, le code de balayage des options d'iptables n'est pas dans le cœur du programme, et introduit des incohérences dans le fonctionnement de l'outil en ligne de commande. Une autre limitation fréquemment citée est l'existence d'une

seule cible par règle : cela suppose une copie des règles et des changements dupliqués (par exemple, utilisation de LOG + DROP ou MARK + ACCEPT).

Au niveau de Netfilter, certaines parties ont un trop faible niveau d'abstraction. Par exemple, les gestions des filtres sur les ports TCP et UDP ne partagent pas de code. Le paramétrage statique des modules cibles entraîne des multiplications de code, comme l'option de la cible qui est toujours propagée sous forme de constante. Il n'est pas possible d'utiliser la variable de manière plus souple. Cela a conduit à la construction de modules spécifiques comme **IPMARK**, **IPCLASSIFY**, et pourrait amener à un module **TCPPORTCONNTRACK**. S'il avait été possible d'indiquer dynamiquement quel champ modifier et avec quelle méthode, un seul module aurait suffit.

3.2.2 Un peu de bien sur Iptables

Iptables dispose d'un grand nombre de critères et d'une classification très flexible. Il s'interface parfaitement avec les fonctionnalités connexes au filtrage comme le partage de charge, le routage *multipath*, la manipulation de paquets réseau. Les opérations internes sont rapides comparées à celles de TC et il est aisé d'ajouter de nouvelles extensions. Cependant, ces extensions sont à la source de la mauvaise qualité des modules externes. Iptables a ses propres classificateurs et n'est pas capable, à l'inverse de tc, d'utiliser u32 qui est très puissant, mais difficile à utiliser. Iptables ne supporte pas non plus BPF (*Berkeley Packet Filter* utilisé par **tcpdump** et **libpcap**) qui est totalement programmable. Il pourrait donc être intéressant de disposer d'une syntaxe similaire à celle des classificateurs de Linux et BPF.

3.2.3 Principes et aperçu de Nftables

Pour fixer le problème du coût de modification du jeu de règles, il est souhaitable d'utiliser Netlink pour réaliser des changements incrémentaux du jeu de règles, et une remontée des modifications. En mettant en place un système de multicibles, les extensions pourraient renvoyer un verdict arbitraire pour orienter les flux dans des sous-chaînes. Les cibles devraient être paramétrables à la volée et durant leur fonctionnement.

Nftables est découpé en trois composants, une implémentation noyau, un frontal nftables en espace utilisateur, une bibliothèque **libnl** chargée des interactions bas-niveau avec le noyau.

La syntaxe du frontal utilisateur ressemble à celle de BPF. Un exemple basique est par exemple :

```
nft rule add filter output tcp dport ssh
nft rule add filter output ip daddr 191.68.0.1 ip protocol tcp
```

La syntaxe est flexible, puisque l'on peut aussi écrire :

```
nft rule add filter output tcp dport == 22
nft rule add filter output ip daddr == 191.68.0.1 ip protocol == tcp
```

La puissance de Nftables s'exprime sur la définition et l'utilisation d'objets composés :

```
nft add filter output ip addr {192.168.0.0/24, 192.168.1.1, 10.0.0.0/8}
```

Les impressions de l'auditoire à la suite de cette présentation marathon de Patrick McHardy ont été unanimes. Comme énoncé par David Miller, la qualité du code produit tant du côté architecture que du coté réalisation est exceptionnelle (alors que le projet est très jeune). En termes de fonctionnalités et d'utilisation, les changements entraînés seront importants. Cependant, compte tenu de l'ampleur de la tâche, une date pour l'abandon de la couche actuelle n'est pas encore prévue.

3.3 Xtables-addons (Jan Engelhart)

3.3.1 Les problèmes de Patch-o-matic

Patch-o-matic est un système qui gère l'ajout d'un ensemble de fonctionnalités pour Netfilter et pour iptables. Il présente des problèmes de qualité, car le code est souvent centré sur une architecture (les contributions sont souvent des réponses à des problèmes individuels). Patch-o-matic contient en fait des patchs pour le noyau et pour iptables. Il s'appuie donc directement sur les sources du noyau. Les extensions doivent être mises à jour à chaque changement dans l'API. La pire chose est que, le code devant supporter plusieurs versions du noyau, il y a un usage massif des compilations conditionnelles de gcc (**#ifdef**) dans le code.

3.3.2 Principe et apport de Xtables-addons

Xtables-addons ne s'appuie sur aucun patch et n'a besoin que des en-têtes du noyau pour compiler. Une couche d'API supplémentaire a été ajoutée pour s'abstraire des dépendances sur les versions du noyau, et le code spaghetti. Le code des fonctionnalités supplémentaires est alors développé au-dessus de l'API de Xtables-addons et n'a pas besoin d'être modifié suivant la version du noyau.

Xtables-addons contient quelques-unes des extensions les plus intéressantes qui existent dans patch-o-matic :

- **condition** : filtre selon un drapeau modifiable en espace utilisateur. En écrivant dans une entrée dans **/proc**, on décide si une condition est vérifiée ou non.
- **geoip** : filtre de correspondance sur les pays.
- **TEE** : « reroute » une copie du paquet.
- **TARPIT** : fait tendre le débit d'une connexion vers 0, ce qui peut être utile pour compliquer la vie de certaines personnes (lire par exemple spammeurs).
- **DELUDE** : réalise la négociation de sessions TCP et ferme la connexion. Il est par exemple utilisable pour faire croire aux scanners que tous les ports sont ouverts.

Dans la discussion suivant la présentation, il a été décidé de supprimer patch-o-matic de l'arbre subversion. Xtables-addons va devenir le dépôt officiel pour les fonctionnalités non officielles du projet.

3.4 MIPv6 (Yasuyuki Kosakai)

Yasuyuki Kosakai est à l'origine du support d'IPv6 dans Netfilter. C'est ce travail qui lui a valu d'être nommé dans la *coreteam* du projet. Son œuvre est maintenant quasiment complète et le seul travail restant est sur Mobile IPv6 ou MIPV6.

MIPv6 est la couche d'IPv6 gérant la mobilité. Il s'agit d'un mécanisme extrêmement complexe qui permet à une machine de garder son IP alors qu'elle passe d'un réseau à un autre. Une redirection des flux est assurée de manière transparente en utilisant 2 modes. La première possibilité repose sur un tunnel IPSEC bidirectionnel. Tous les flux passent par le réseau d'origine (appelé « Home »). La deuxième solution est basée sur une optimisation du routage vers cette IP.

Dans les deux cas, cela conduit à une problématique de définition des *tuples* (couple IP:port) du conntrack. Le conntrack doit en effet être construit pour gérer ce problème. Deux choix sont possibles. Ou `nf_conntrack_ipv6` construit les tuples par adresse Home ou `nf_conntrack_ipv6` gère les tuples comme « tunnelés ». Le premier choix est le meilleur, car il correspond au fonctionnement interne. Par ailleurs, c'est bien l'adresse finale qui sera prise en compte. Il est cependant facile d'envoyer des paquets comportant une fausse adresse Home (c'est un simple champ IP supplémentaire). Ainsi, le filtre à états peut laisser passer le paquet forgé si les paramètres de contrôle ne sont pas inspectés.

La proposition de développement est la suivante. Le nouveau module de suivi de connexion inspecte la signalisation MIPv6 (proto = 135) et garde une liste des passerelles Home. En faisant cela, on peut rejeter les paquets qui proviennent de passerelles non enregistrées. Comme d'habitude, le chiffrement des données suffit à tout casser, puisque l'on ne peut plus *parser* la signalisation. Or, le `draft-irtf-mip6-cn-ipsec-05` introduit le chiffrement des mises à jour et accusés de réception, et il n'existe alors aucun moyen de filtrer sur ces données.

3.5 Panorama de bibliothèques et d'outils (Pablo Neira)

Pablo Neira a réalisé un panorama de l'état des bibliothèques et des outils en espace utilisateur. La moitié des bibliothèques présentent deux API (`libnfnetlink`, `libnetfilter_conntrack`). La nouvelle API permet une meilleure gestion des erreurs. Les bibliothèques `libnetfilter_queue` et `libnetfilter_log` n'ont presque pas évoluées en un an.

L'évolution principale est celle de `libnetfilter_conntrack`. La nouvelle API est basée sur une logique d'envoi/réception et propose un nombre appréciable de fonctions *helper* (afficher, comparer, copier). L'évolution principale en termes fonctionnels est arrivée avec la dernière version. Elle introduit un filtre BSF (*Berkeley Socket Filter*) qui peut être utilisé pour filtrer les entrées avant qu'elles ne quittent le noyau. Avec un filtre BSF, on peut par exemple éviter de récupérer un événement de connexion pour *localhost*. Il est ainsi possible de limiter le nombre d'événements envoyés depuis le noyau vers l'espace utilisateur. Il s'en suit un accroissement des performances des logiciels utilisant la bibliothèque, puisqu'ils ne traitent plus de messages inutiles.

3.6 Ulogd2 (Éric Leblond et Pierre Chifflier)

Lors du précédent workshop en septembre 2007, j'avais indiqué mon souhait de reprendre le développement de ulogd2 qui avait été abandonné par Harald Welte. Un travail tant d'enrichissements fonctionnels que de stabilisation a été mené au cours de l'année 2008. Il a notamment conduit à l'introduction de nouveaux schémas de bases de données qui ont été présentés à l'assemblée par Pierre Chifflier (voir article sur ulogd2 pour une description).

Les buts fonctionnels ont presque tous été atteints et le seul module dans un état instable est IPFIX (export Netflow des données). La coreteam décide donc que moyennant conclusion sur le sujet IPFIX, une sortie d'une nouvelle version stable d'ulogd2 est envisageable après une période de stabilisation et d'observation du code.

4 Résultats

En dehors des quelques présentations détaillées ici, la courte présentation de Tproxy a été une des plus importantes. Après quatre ans d'efforts, l'inclusion de tproxy dans Netfilter et dans Linux a enfin été réalisée. Elle l'a d'ailleurs été en direct, puisque David Miller, présent dans la salle, a intégré les patchs dans son arborescence git 10 minutes après la fin de la présentation.

La sixième convention Netfilter aura été très dense sur le point technique. Mais l'organisation avait néanmoins aménagé des moments de détentes. Ce fût notamment le cas lors d'une croisière dînatoire sur un bateau mouche privatisé pour le groupe. À cette occasion, des *goodies* dont un tablier « *Netfilter hackers like to cook fish* » ont été offerts par INL aux participants.

Figure 4 : Échange technique entre des développeurs et des passants.

Références

- Présentations de la journée utilisateur : http://nfws.inl.fr/?p=83.
- Description de nftables par Patrick McHardy : http://people.netfilter.org/kaber/weblog/2008/08/20/.
- Xtables-addons : http://jengelh.medozas.de/projects/xtables/.
- Draft IETF mip6 ipsec : http://tools.ietf.org/html/draft-ietf-mip6-cn-ipsec-05.

Auteur : Éric Leblond

Utilisateur GNU/Linux depuis 1996. Directeur technique d'INL. Développeur principal de NuFW, commiteur Netfilter.

Interview de Patrick McHardy

Patrick McHardy est le chef du projet Netfilter depuis 2007. Il a eu la gentillesse de répondre à une interview par mail où il présente son travail et ses motivations.

Propos recueillis par

- Éric Leblond

Pouvez-vous vous présenter et nous dire comment vous avez découvert Linux et Netfilter ?

J'ai 29 ans et je vis actuellement dans la jolie ville de Freiburg en Allemagne. Contrairement à ce que beaucoup de personnes pensent en regardant mon nom, je ne suis ni anglais, ni américain, mais je suis né à Frankfort et ma langue natale est l'allemand. J'ai étudié l'informatique à Freiburg, mais cela m'a vite ennuyé et j'ai arrêté mes études après quelques semestres.

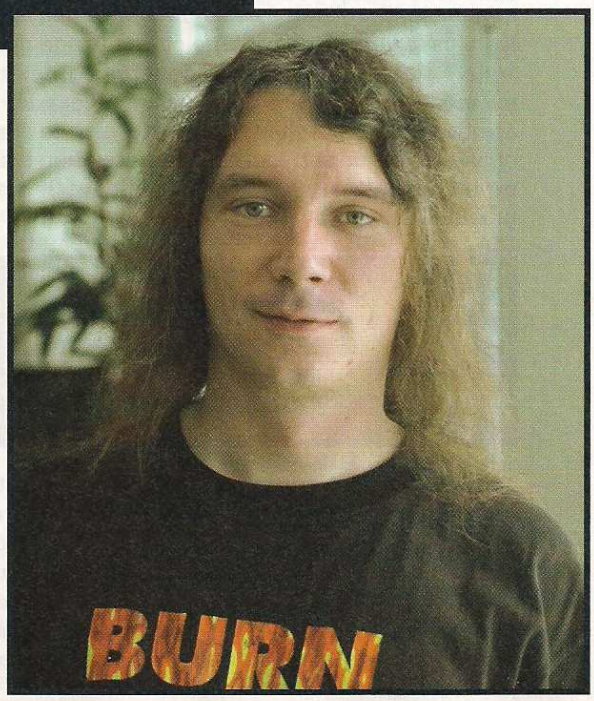
Je gagne ma vie en tant que freelance sur des projets de *consulting* et de développement sur le noyau Linux. J'essaie cependant de passer le plus de temps possible sur des projets Linux pour lesquels j'ai un intérêt personnel. Astaro sponsorise mon travail depuis janvier 2006, ce qui me permet de passer beaucoup de temps sur Netfilter. Pendant mon temps libre, j'essaie de conserver une vie sociale en fréquentant des personnes au moins partiellement non *geek*, en prenant plaisir à traîner dans des bars sombres, en cuisinant un peu (Je suis un grand fan de la cuisine française) et en assistant aux cours de droit de l'université de Freiburg.

J'ai découvert Linux en 1996 ou 1997. Si l'on omet d'autres architectures moins connues ou plus obscures, j'avais principalement été un utilisateur d'Amiga jusque-là, mais il est devenu de plus en plus évident que cette architecture était désespérément en perte de vitesse. J'ai donc décidé de prendre un ordinateur x86. Utiliser Windows était hors de question, la petite expérience que j'en avais m'avait suffit pour comprendre qu'il serait inutilisable pour moi en tant que développeur. En cherchant des alternatives, j'ai découvert Red Hat Linux, qui me paraissait bien plus attirant, spécialement du fait qu'il était livré avec le code source complet du noyau. À cette époque, tous les systèmes que j'avais utilisés étaient propriétaires et je n'avais donc jamais eu la chance de regarder le code d'un système d'exploitation. Comme j'avais joué avec des coupleurs acoustiques et des modems depuis l'âge de dix ans, j'étais intéressé depuis longtemps par découvrir comment une couche réseau fonctionnait vraiment et j'ai donc commandé une copie de RH Linux. L'installation était ardue et le gestionnaire de fenêtres était incroyablement laid, mais, une fois que tout fut paramétré et fonctionnel, la première chose que j'ai faite fut de regarder le code du noyau. Ça m'a vraiment dépassé de loin à l'époque. Je pense que c'était au-delà de mes capacités à ce moment. Je me concentrais donc sur les moyens d'améliorer le système, mais, pendant les années suivantes, je continuais de manière irrégulière à étudier le code quand j'en avais le temps.

Quand avez-vous décidé de participer à Netfilter ?

J'ai écrit mon premier morceau de code conséquent lié à Netfilter aux alentours de 2000. Il s'agissait d'une implémentation de contrôle du débit TCP, une technique (malheureusement brevetée aux États-Unis) de contrôle du débit du trafic entrant par régulation du débit des TCP ACK et de la taille de fenêtre TCP.

Patrick McHardy



NDLR

L'idée de cet algorithme est de travailler sur deux paramètres qui influent sur la bande passante d'une connexion. Le premier est l'envoi des ACK qui accusent la réception des données. Si l'émetteur reçoit les ACK moins vite, il aura tendance à réduire son débit d'émission. Le deuxième paramètre est la taille de fenêtre TCP qui définit le volume de données que l'émetteur peut envoyer sans avoir reçu d'accusé de réception. L'action sur ses deux facteurs permet un contrôle du débit des données de l'émetteur, puisque l'on contrôle deux des facteurs de régulation du débit.

L'implémentation était une discipline TC, mais elle utilisait le suivi de connexions Netfilter pour maintenir un état par connexion et le patch (intégré plus tard dans l'arborescence officielle) de suivi de la taille de fenêtres TCP pour valider les flux de contrôle TCP, la détection des retransmissions, etc. À cause du problème de brevet, il était clair que je ne pourrais pas faire intégrer le code dans le noyau officiel. J'ai donc demandé au propriétaire du brevet sa permission de l'utiliser, mais, sans surprise, il a décliné ma proposition et j'ai abandonné le projet.

Mes premières contributions actives remontent à 2002 ou 2003. J'avais un travail d'étudiant comme développeur chez Pyramid Computer, une société de Freiburg qui, à cette époque, produisait des *appliances* basées sur Linux. Je développais là-bas le composant pare-feu de leur nouvelle appliance. Il utilisait une bonne partie des fonctionnalités les plus avancées de routage et de filtrage de Linux et j'ai trouvé pas mal de bogues, fonctionnalités manquantes ou incohérences. J'ai donc commencé à les corriger et à envoyer les patches. Pendant l'ère de Linux 2.4, les choses évoluaient très lentement et même les corrections de bogues atterrissaient souvent dans le patch-o-matic pendant des mois.

NDLR

Le patch-o-matic était un ensemble de patches et de fonctionnalités ajoutés à Netfilter (donc au noyau) ou iptables qui n'était pas disponible dans le noyau officiel mais qui était distribué par le projet Netfilter.

Mais l'impression que mes contributions étaient appréciées, le fait que l'on pouvait poser des questions et avoir des réponses des auteurs et en particulier l'arrivée plus ou moins rapide de mon travail dans le noyau officiel me motivaient fortement. En devenant plus familier avec le code, je découvrais des bogues, des incohérences et des choses qui méritaient d'être corrigées. Porté par tout cela, je continuais le travail.

Quelle question vous énerve-t-elle le plus sur une liste de diffusion ?

Je suis persuadé d'être une personne au caractère tempérée (ndlr : je confirme) et je ne m'énerve pas facilement. Je n'aime pas quand les gens font trop de bruit sur une liste de diffusion, mais c'est tout. On apprend à ignorer ces choses-là rapidement. Cependant, j'aime toujours observer une bonne joute par emails et anticiper sur son apparition :)

Quel est le meilleur retour que vous ayez eu jusqu'ici ?

Parfois, des utilisateurs envoient des courriels pour exprimer leur gratitude pour notre travail ce qui est à mon avis un geste sympathique. Le meilleur... je pense que c'est quand un type dans un club est venu vers moi, m'a demandé si mon nom était Patrick McHardy (ce qui a dû me perdre, c'est que je portais un T-shirt de Nerd) et a commencé à m'offrir des verres :)

En ce qui concerne les retours des développeurs, il y en a malheureusement peu

En ce qui concerne les retours des développeurs, il y en a malheureusement peu ; la plupart de mes patches sont *mergés* à leur première soumission sans beaucoup de retour. Il est toujours difficile de juger si c'est parce que personne ne les a regardés en détail ou si c'est parce qu'ils étaient simplement parfaits. Pour moi, le meilleur retour non technique de la part de développeurs est quand quelqu'un reconnaît la beauté derrière quelque chose de particulièrement intelligent ou élégant dans le code (ndlr : cela s'est notamment produit à l'issue de la présentation de nftables lors du *workshop* 2008 à Paris).

Comment considérez-vous vos responsabilités en tant que leader du projet Netfilter ?

Il ne s'agit pas réellement d'un ensemble bien défini de responsabilités, il s'agit plutôt de s'occuper des choses que personne d'autre ne fait et bien sûr de relire et intégrer des patches. Théoriquement, je suis aussi supposé prendre la décision finale si la *coreteam* n'arrive pas à se décider sur quelque chose, mais je ne me rappelle pas que cela soit déjà arrivé.

En ce qui concerne les tâches individuelles, être responsable en dernier lieu de l'évaluation et de l'incorporation des patches est positif sur de nombreux aspects. Le plus important est que c'est probablement le levier qui me permet de pouvoir orienter au moins partiellement les développements dans la direction qui me semble la bonne. Cela me force aussi à identifier les erreurs au plus tôt au lieu d'avoir à les corriger une fois qu'elles ont atteint le noyau officiel. Il y a beaucoup plus de travail dans ce cas puisqu'il faut gérer les problèmes de compatibilité, le *backport* des corrections,

etc. Je pense que j'ai fait un bon travail d'amélioration de la qualité du code. D'un autre côté, cela fait de moi un point d'échec central (SPOF), ce qui peut être frustrant et créer du travail supplémentaire pour les contributeurs quand je n'ai pas le temps d'évaluer et d'incorporer leurs patches. J'essaie donc d'alléger la dépendance, mais il n'y a pas de moyens de faire ça simplement, et cela est aggravé par le fait que je sois le seul membre de la coreteam à être actuellement payé pour travailler sur Netfilter.

> J'aime réellement fixer des bogues, mais les rapports de bogues peuvent être vraiment désorganisants

Une autre facette d'être le point d'entrée du flux de patches est que les plaintes qui descendent en sens inverse atterrissent d'abord dans votre boite mail. Le type de plainte le plus urgent est celui où l'ordinateur de l'utilisateur ne *boote* pas après que vous avez une fois encore cassé quelque chose :)

La publication de nouvelles versions des outils en espace utilisateur est une des tâches que je n'aime pas beaucoup, principalement parce que c'est répétitif et que le mise à jour du site web est pénible.

Ensuite, viennent les corrections de bogues et les réponses aux rapports de bogues. J'aime réellement fixer des bogues, mais les rapports de bogues peuvent être vraiment désorganisants : il faut répondre rapidement, sinon on court le risque que le rapporteur du bogue disparaisse, ou que l'on réagisse lentement sur un bogue sérieux.

Malheureusement, peu de gens, y compris parmi les développeurs les plus actifs, participent aux rapports de bogues. En particulier les rapports de bogues reçus depuis le *bugzilla* de kernel.org sont pratiquement tous exclusivement traités par moi-même. Là encore, je suis le point de passage obligé, mais c'est un peu plus un problème pour les utilisateurs que pour les développeurs.

Et enfin, il y a l'organisation de la conférence des développeurs, à laquelle j'ai participé (pour ce qui est la partie non locale) ces deux dernières années. Cela prend toujours beaucoup de temps et la collecte de l'argent des sponsors m'expose personnellement à certains risques légaux et financiers. Le concept global de sponsoring semble troubler les conseillers fiscaux et j'en ai rencontré un seul capable de me donner des conseils légaux sur comment gérer cela proprement. Cela fait donc partie des activités que j'apprécie le moins.

De combien de temps disposez-vous pour implémenter vos propres idées ?

Je dispose de beaucoup de temps pour travailler sur Netfilter et sur le noyau. Le temps que je peux allouer à mes propres idées dépend de facteurs externes, comme ce que les autres développeurs font et combien de temps est utilisé pour analyser et incorporer les patches, participer aux discussions, gérer les rapports de bogues, etc. Comme je suis aussi actif dans d'autres parties du noyau, il serait facile d'utiliser tout mon temps sur des revues de patches et des discussions, mais cela ne passe pas à l'échelle, donc, parfois, il faut juste laisser passer les choses et faire confiance aux autres personnes qui feront les bons choix.

Un problème sérieux est que, même quand il est possible d'avoir du temps, la communication avec les autres développeurs qui se fait exclusivement par e-mail peut être désorganisatrice pour le travail en cours, car l'activité est toujours multitâche ; à un certain moment le coût du changement de contexte devient tel qu'il n'est plus possible d'effectuer un travail réel. La seule manière pour moi de travailler efficacement sur des gros projets est d'ignorer complètement les e-mails pendant quelque temps.

Vous travaillez actuellement sur Nftables. Quel est le but de ce projet ?

Le but premier est de créer un successeur pour iptables, ip6tables, arptables et ebtables. Cela sera, je pense, la quatrième génération de pare-feu pour Linux. Il y a bien sûr un sérieux syndrome du second système. Il essaye d'adresser de nombreux buts et souhaits.

Commençons par les deux buts premiers, qui expliquent aussi pourquoi j'ai commencé une nouvelle implémentation depuis le début au lieu d'améliorer l'existant.

1 Une interface utilisateur moderne, basée sur Netlink et travaillant de manière incrémentale

L'interface d'iptables souffre d'un certain nombre de sérieux problèmes de conception qui ne peuvent être corrigés sans briser la compatibilité binaire (ABI). Le premier problème majeur est que le jeu de règles est représenté par un énorme *blob* qui ne peut être remplacé qu'atomiquement, ce qui entraîne une complexité quadratique pour les changements de jeux (incrémentaux) de règles et, si les développeurs n'y prennent garde, cela peut entraîner la perte par le noyau de l'état interne des règles lorsque des changements dissociés de règles dissociées sont effectués. Le second problème majeur est directement relié à cela. La représentation interne dans le noyau est exactement celle du blob qui est utilisé dans l'ABI de l'espace utilisateur. Cela réduit sérieusement le degré de liberté disponible pour améliorer l'implémentation interne du noyau, puisque tout changement sur les structures de données brisera la compatibilité applicative.

2 Un système de classification générique, indépendant d'une famille de protocoles spécifique :

Un autre choix malheureux dans l'ABI utilisateur a été d'embarquer des informations spécifiques au protocole comme des adresses IPv4 dans le cœur des structures de données. Cela a rendu impossible l'utilisation de la même ABI pour des protocoles différents comme IPv6, ARP ou le *bridge* et la réutilisation des parties du cœur du code de

classification, si bien que les gens ont créé trois « ports » couper-coller (NDLR : IPv6, arp et bridge), quadruplant le coût de maintenance et de développement.

Buts secondaires

Support des règles de synchronisation à état

Certains filtres maintiennent un état interne. L'exemple le plus évident est le filtre *limit*. Afin de supporter la synchronisation de jeux de règles à état entre des systèmes différents l'espace utilisateur doit être capable d'accéder à ces états. (NDLR : Par exemple pour la réplication d'état entre deux pare-feu actif/actif),

Cibles multiples par règle

Une plainte courante est que si l'on veut, par exemple, journaliser et accepter un paquet, il est nécessaire de dupliquer la règle ou créer une chaîne utilisateur dédiée, qui contient les deux actions. nftables ne fait plus la distinction entre filtre et cible et permet l'utilisation d'un nombre arbitraire des deux dans une simple règle.

Support natif des évaluations de jeux de règles (et des ensembles) non linéaires

L'évaluation linéaire des règles telle que faite par iptables est lente, les ensembles permettent une classification plus rapide et des expressions plus naturelles pour certains types de règles. Pour iptables, nous avons le module non officiel `ipset` de Jozsef Kadlecsik, qui rend possible l'utilisation d'ensembles pour certains types de données comme les adresses ou les numéros de ports. nftables pousse cela un pas plus loin et supporte des ensembles pour tous les types de données. Il supporte aussi les recherches par intervalles dans des ensembles et utilise des ensembles comme des dictionnaires qui permettent de représenter des algorithmes de classification plus intéressants.

Cibles paramétrables dynamiquement

Les ensembles sont utiles dans le cas où il y a de nombreux filtres similaires, mais quand le paramètre de la cible change, il est tout de même nécessaire d'avoir des règles séparées. Une cible paramétrable dynamiquement permet de déterminer ces paramètres au moment de l'exécution, par exemple en réalisant une recherche dans un dictionnaire. Il est ainsi possible d'exprimer quelque chose comme :

```
"marque les packets de l'adresse IP x1 avec la valeur m1, x2 avec la valeur m2, …"
```

dans une règle unique. Cela offre aussi la possibilité d'exprimer des choses un peu plus folles comme

```
"DNAT la connexion vers l'adresse source du paquet"
```

Limitation du code dupliqué pour des tâches similaires

De nombreuses tâches peuvent être généralisées assez facilement, par exemple le filtre TCP, le filtre UDP, le filtre adresse IP, etc. Tous chargent des données depuis le contenu du paquet et les comparent à une constante. La seule différence est l'offset et la longueur des données. Par conséquent, nftables utilise un seul module « *payload* », l'espace utilisateur est responsable de la traduction des filtres spécifiés par l'utilisateur en des paramètres appropriés. La même chose s'applique à d'autres types de filtres.

Sémantique et intelligence en espace utilisateur

Les retours d'erreurs d'iptables sont plutôt pauvres parce que la plupart (sinon toutes) des vérifications sémantiques sont faites par le noyau à qui il manque un bon mécanisme de retours d'erreurs. Généralement, l'espace utilisateur est plutôt stupide et se limite à construire des structures de données et à les passer au noyau. Pour être capable de réaliser toute sorte de transformation, nous avons besoin en espace utilisateur d'une meilleure compréhension des jeux de règles. Le noyau de l'autre côté n'a pas besoin d'une compréhension totale, il a seulement besoin d'être sûr que le jeu de règles ne contient pas de choses qui peuvent crasher ou blesser le noyau lors de l'exécution. Ainsi, avec nftables, le noyau fournit principalement un langage de classification léger et tout ce qui est nécessaire à la compréhension de la sémantique est fait en espace utilisateur.

Planifiez-vous de merger nftables dans le noyau officiel ?

Absolument. Je n'ai pas actuellement d'estimation du moment, mais j'espère l'avoir en bon état aux alentours de la deuxième moitié de cette année. Une première version alpha sera disponible dans le mois (NDLR : janvier 2009).

Y a-t-il d'autres nouvelles choses de prévu pour Netfilter ?

Je n'ai actuellement rien de particulièrement intéressant en vue, mais laissez-moi répondre à la question avec une petite anecdote.

Au début de 2006, Astaro m'a demandé de finir le module de suivi de connexions pour H.323. Je travaillais dessus avec une priorité basse depuis l'année précédente. H.323 est un ensemble de protocoles bien compliqué et mon module de suivi de connexions nécessitait encore un bonne quantité de travail. J'ai libéré du temps pour m'en occuper de manière plus urgente ; le jour où j'ai voulu commencer à travailler dessus, je me suis levé, j'ai ouvert mon mail et j'ai trouvé une soumission de patch de Jing Min Zhao, contenant un module de suivi de connexions complet pour H.323, qui, à part quelques points mineurs, était tout simplement parfait et sans doute meilleur que le mien.

Cela montre que même les choses imposantes et compliquées peuvent en fait surgir du néant, sans qu'on n'en sache rien à l'avance. Nous avons un bon nombre de contributeurs actifs travaillant dans leur propre domaine d'intérêt. Je suis donc persuadé que nous continuerons à voir arriver de nouvelles choses intéressantes.

Cela dit, nous avons réalisé ces dernières années un ensemble d'inclusion de fonctionnalités utiles dans le noyau officiel, si bien que les demandes pour de nouvelles fonctionnalités sont en déclin constant et l'attention se porte de plus en plus sur la maintenabilité et l'amélioration de la qualité du code. Les domaines les plus actifs sont actuellement ctnetlink et, par extension, les améliorations de la gestion du *failover* et de la synchronisation des tables d'états, la journalisation en espace utilisateur, l'unification des codes de arptables, ebtables, iptables et ip6tables ainsi que la fin du travail sur le support des espaces de noms réseau.

La figure 1 montre le travail de la coreteam netfilter deuis 2.6.12. Avez-vous des remarques sur ce graphe ?

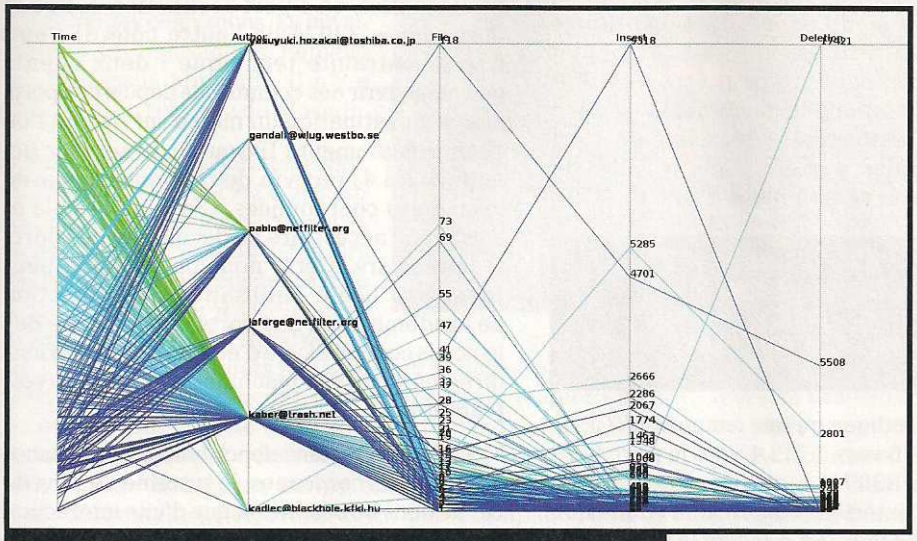


Figure 1 : Visualisation du travail de la coreteam sur Netfilter depuis 2.6.12

Le graphique de la figure 1 utilise les coordonnées parallèles pour représenter le travail sur Netfilter des membres de la coreteam depuis 2.6.12. Chaque ligne brisée représente un commit. Pour tracer un commit, on place la date à laquelle il a été effectué sur le premier axe en partant de la gauche (les plus anciens en bas les plus récents en haut), sur le deuxième axe, on place l'auteur, sur le troisième le nombre de fichiers modifiés, sur le quatrième le nombre de lignes effacées et, enfin, sur le cinquième, on place le nombre de lignes ajoutées. On relie ensuite les points entre eux pour former une ligne brisée. Dans le cas de ce graphe, la couleur de cette ligne est choisie en utilisant un dégradé sur la date de commit. On voit ainsi que le travail récent en vert a principalement été effectué par Patrick McHardy (kaber). On voit aussi que les commits récents, pris individuellement ne sont pas très volumineux. Il y a en effet plus de bleu ciel dans les commits avec beaucoup de choses modifiées que de vert. (NDRL) Merci à Sébastien Tricaud et son excellent outil Picviz utilisé pour la génération de ce graphe.

Je suis heureux d'avoir effacé autant de lignes de code de Netfilter. J'espère que cela continuera encore.

De manière plus générale, je ne considère pas les métriques basées sur la taille et le nombre de patches comme un indicateur très significatif puisqu'elles mesurent uniquement la quantité et ne disent rien sur l'importance et la qualité des contributions et du temps de travail qui a été nécessaire. Elles montrent principalement les niveaux généraux d'activité, ce qui est, dans mon cas, une petite surprise.

Votre travail sur Netfilter est impressionnant, mais, en fait, vous travaillez aussi sur d'autres parties du noyau. Pouvez-vous nous en dire plus ?

Merci. Je travaille uniquement dans des domaines relatifs au réseau. À côté de Netfilter, je maintiens les pilotes VLAN et MACVLAN, je suis listé comme co-mainteneur de IPv4/IPv6 et je travaille occasionnellement sur le cœur de l'infrastructure réseau, IPsec, etc. L'ordonnancement de trafic est un autre de mes centres d'intérêt, j'y ai implémenté les ordonnanceurs HFSC et DRR, le classifier de flux et fait beaucoup de travail sur l'infrastructure et les corrections de bogues. Un autre sujet qui m'intéresse est le protocole netlink et les interfaces avec l'espace utilisateur. Il y a historiquement beaucoup d'interfaces mal conçues. Les corriger, ajouter de nouvelles interfaces correctes et prévenir l'ajout de nouvelles interfaces mal conçues est à mon avis très important, car, lorsqu'une interface est dans le noyau officiel, on est bloqué avec elle pour des années. Malheureusement, les auteurs de pilotes en particulier ne semblent pas avoir de vision d'ensemble lorsqu'ils implémentent une nouvelle interface, et une attention constante est donc nécessaire.

Lors des mois précédents, j'ai dû cependant réduire mes activités dans ces domaines, car mes projets actuels demandent beaucoup de temps. En plus de travailler sur nftables, j'ai récemment démarré une implémentation de la couche de protocoles DECT (oui, le truc pour téléphone sans fil :)). C'est un projet en basse priorité, que je réserve pour le temps où j'ai besoin d'une pause sur nftables, mais c'est un projet compliqué et il demandera probablement une grande quantité de travail.

Propos recueillis par : Éric Leblond

Utilisateur GNU/Linux depuis 1996. Directeur technique d'INL. Développeur principal de NuFW, commiteur Netfilter.

Introduction : Pablo Neira et les

Pablo Neira Ayuso est l'initiateur des logiciels conntrack-tools et membre de la coreteam Netfilter depuis 2007. Il évoque dans cette interview par mail son travail et ses motivations.

Propos recueillis par

- Éric Leblond

1 Quelques mots sur le suivi de connexions

Le suivi de connexions de Netfilter ou conntrack est l'une des parties les plus importantes de Netfilter, puisqu'elle est responsable du filtrage à état (Voir **[intro]**). Ce sous-système de Netfilter a été introduit en même temps que le système dans le noyau 2.4.0. Le principe de fonctionnement du conntrack est le suivant : le noyau maintient une table contenant la liste de tous les flux passant à travers le pare-feu. Comme les paquets peuvent être modifiés lors des opérations de traduction d'adresses (NAT), le paquet arrivant sur le pare-feu peut être différent du paquet sortant de la machine. Il est donc nécessaire de stocker les deux valeurs. Ceci nous donne des entrées dans le suivi de connexions qui ressemblent à ceci :

```
tcp      6 431978 ESTABLISHED src=192.168.3.176
dst=1.2.3.4 port=33974 dport=4129 packets=3889
bytes=623952 src=1.2.3.4 dst=2.4.5.6 sport=4129
dport=45944 packets=3887 bytes=339242 [ASSURED] mark=0
secmark=0 use=1
```

Cette entrée indique qu'une connexion TCP de 192.168.3.176 vers 1.2.3.4 vers le port 4129 (port source 33974) a été translatée et est vue depuis l'extérieur comme une connexion TCP de 1.2.3.4 vers 2.4.5.6 vers le port 45944 (port source 4129). Cela peut sembler absurde, mais tout s'explique logiquement :

1. Les IP sont inversés : les deuxièmes coordonnées sont celles vues lors de la réception des paquets depuis l'extérieur. Cela est donc comparable à ce qui peut être vu dans un tcpdump.

2. Les ports, même inversés, ne correspondent pas : en effet, on a port destination 45944 et port source 33974 de l'autre côté. Il s'agit d'une contrainte technique : deux clients peuvent ouvrir des connexions depuis des ports égaux à destination du même service. Si l'on change seulement l'IP source, le serveur (ici à l'IP 1.2.3.4) recevra deux connexions avec les mêmes coordonnées. C'est impossible et il est donc nécessaire de décaler un des ports si nécessaire. Cette non-concordance peut aussi être liée à l'utilisation de la fonction de randomisation des ports qui empêche des logiciels comme Skype d'établir des connexions directes entre des machines translatées (voir **[random]**).

Malgré son importance fonctionnel et l'importance des données contenues, le système de suivi de connexions ne bénéficiait que d'une interaction limitée avec l'espace utilisateur.

2 Présentation des conntrack-tools

Pablo Neira Ayuso a donc initié un travail d'enrichissement des interactions avec le suivi de connexion au moment où une refonte globale des communications avec l'espace utilisateur était en œuvre (Voir **[GLMF 86]**). Ce travail a mené à une suite d'outils, les conntrack-tools, transformant complètement la façon d'envisager le suivi de connexions. Ce projet comporte deux logiciels, **conntrack** qui interroge et modifie le suivi de connexions et **conntrackd** capable de répliquer le suivi de connexions entre plusieurs machines.

L'utilitaire conntrack

Le programme conntrack est en charge d'interroger, de modifier ou de détruire des entrées du suivi de connexions. C'est un outil en ligne de commande qui peut, par exemple, remplacer la commande

```
# cat /proc/net/ip_conntrack
tcp      6 431989 ESTABLISHED src=192.168.1.129
dst=80.248.214.47 sport=56532 dport=5222 packets=115
bytes=16982 src=80.248.214.47 dst=192.168.1.129
sport=5222 dport=56532 packets=122 bytes=36045
[ASSURED] mark=0 secmark=0 use=1
```

en le lançant avec l'option `-L` :

conntrack-tools

```
# conntrack -L
udp      17 28 src=192.168.1.129 dst=212.27.40.241 sport=39502 dport=53
packets=1 bytes=65 src=212.27.40.241 dst=192.168.1.129 sport=53
dport=39502 packets=1 bytes=125 mark=0 secmark=0 use=1
tcp      6 431960 ESTABLISHED src=192.168.1.129 dst=80.248.214.47
sport=56532 dport=5222 packets=113 bytes=16730 src=80.248.214.47
dst=192.168.1.129 sport=5222 dport=56532 packets=120 bytes=35941
[ASSURED] mark=0 secmark=0 use=1
```

J'entend d'ici les grincheux, mais c'est pareil, le résultat est le même. Sur le dernier point, ils n'auront pas tort, mais les mécanismes mis en œuvre sont bien différents. Lorsque l'on lit le fichier **ip_conntrack**, on interroge directement le module de suivi de connexions et on bloque tout trafic jusqu'à ce que l'on ait lu les données. Avec l'utilitaire **conntrack**, on envoie un message au noyau demandant un *dump* des connexions. Le noyau génère un message et l'envoie au processus appelant. Il n'y a donc pas ou très peu de blocage.

Wolfotrack, qui reprend la version GPL du jeu Wolfenstein 3D, utilise conntrack pour lier chaque personnage à une connexion de Netfilter. Pour tuer une connexion, il suffit de tuer le personnage qui lui est associé.

Là où conntrack devient intéressant, c'est lorsque l'on décide d'agir sur le conntrack. Prenons, par exemple, la deuxième connexion listée lors de la commande précédente. Elle a un *timeout* de 431960 (soit environ 5 jours). Ceci signifie que, en l'absence de paquets reçus sur cette connexion, elle sera détruite après ce timeout. Par contre, si un paquet passe et est attribué à cette connexion, le timeout est repassé à 5 jours. Supposons que l'on désire que la connexion soit détruite au bout d'un temps fixe (pour interrompre un téléchargement juste avant la fin par exemple). Pour ce faire, il suffit d'utiliser l'option **-U** :

```
conntrack -U -s 192.168.1.129 -d 212.27.40.241 -p tcp --dport 5222 \
--sport 56532 -t 20 -u FIXED_TIMEOUT
```

D'autres opérations sont possibles comme le changement de la marque de connexion. Cette modification est intéressante, puisqu'elle peut entraîner un changement de file de qualité de service pour les paquets de la connexion. Il est donc ainsi possible de modifier la qualité de service d'un flux au cours de son existence.

Vous me répondrez qu'une mort rapide et sans douleur est bien plus directe. Pour cela, il faut utiliser l'option **-D** :

```
conntrack -D -s 192.168.1.129 -d 212.27.40.241 -p tcp --dport 5222 \
--sport 56532
```

Attention, ici, il s'agit d'une mort dans le suivi de connexions. Si le jeu de règle autorise la reprise à chaud de connexions, il est possible que la connexion soit recréée et donc que le flux ne soit pas interrompu. Pour interdire toute reprise de connexion sur TCP, il faut être strict sur l'ouverture des connexions dans le jeu de règles et n'autoriser comme nouvelle connexion que des sessions commençant par un paquet SYN.

Le démon conntrackd

Conntrackd est un démon de réplication du suivi de connexions. C'est un outil essentiel dans la construction de pare-feu résistant aux pannes. En répliquant le suivi de connexions, il est en effet possible de réaliser des pare-feu redondants sans perte de connexions lors d'une bascule.

Lorsque l'on passe d'un pare-feu actif à un autre, les flux routés par une machine sont alors brusquement routés par l'autre machine. Le filtrage avec suivi d'état du nouveau pare-feu actif reçoit donc des paquets issus de connexions déjà existantes. N'ayant jamais rien vu passer, le suivi de connexion du pare-feu perçoit donc les paquets comme des tentatives d'initialisation de connexion. Mais, ceux-ci n'ont pas les caractéristiques de paquets d'initialisation et ils sont donc bloqués par le pare-feu. Les utilisateurs perdent donc les sessions qui étaient en cours (et leurs données). Pour éviter ce problème, il y a deux solutions. Soit, on abandonne le filtrage à état, soit on réplique le suivi de connexion d'un pare-feu vers l'autre de manière à ce que, au moment de la bascule, le pare-feu nouvellement actif ait les informations nécessaires à une poursuite du service sans interruption.

Conntrack est une application qui utilise les mêmes mécanismes que l'utilitaire conntrack pour répliquer le suivi de connexion. Il utilise notamment un mécanisme d'écoute des événements qui lui permet de recevoir un message à chaque événement majeur (changement d'état, destruction) dans la vie de chaque connexion. Il établit alors un cache en local des informations du suivi de connexions et il les synchronise avec un pare-feu utilisant lui aussi le démon. L'utilisation de conntrackd ne nécessite pas beaucoup de configuration et la principale contrainte est le besoin d'utiliser pour des raisons de performances et de confidentialité un (ou plusieurs) lien dédié entre les pare-feu.

Conntrackd ne prend en charge que la réplication du suivi de connexion et il faut donc utiliser un logiciel classique de gestion de la redondance comme Keepalived ou Heartbeat pour assurer la détection des pannes et la bascule.

L'interview de Pablo Neira

Pouvez-vous vous présenter et nous dire comment vous avez découvert Linux ?

J'ai commencé des études en Informatique en 1998, mais ce n'était pas mon premier contact avec le monde des ordinateurs. À l'âge de 11 ans, en 1991, mon père avait acheté un Sinclair Spectrum 48k d'occasion pour ma mère (elle est physicienne et mon père a toujours pensé qu'une « personne intelligente » comme ma mère devait être en contact avec l'informatique).

Heureusement pour moi, ma mère n'a jamais trouvé l'informatique intéressante. Cela m'a donné de passer de longues heures devant notre petit télé à programmer en BASIC sur cet ordinateur.

En 2002, un peu après l'écroulement de la bulle internet, j'ai commencé à travailler – tout en continuant mes études, ce qui est très difficile ! – pour quelques petites sociétés informatiques de ma région. J'étais consultant sécurité, développeur et administrateur système à temps partiel. Malgré cela, j'ai réussi à finir mes études d'informatiques. En 2005, j'étais sur le point de partir en France pour y faire une thèse pendant 3 ans quand j'ai eu la possibilité de rejoindre l'Université de Séville, ce que j'ai fait finalement. Depuis lors, j'y travaille comme professeur et chercheur.

Pablo Neira Ayuso

Je suis arrivé relativement sur le tard dans le monde de Linux. En 2001, un très bon ami m'a poussé à installer Linux et m'a motivé à passer des heures et des heures en face de l'écran. La disponibilité complète du code source, et donc la chance de savoir, de première main, comment les choses sont réellement faites, a occupé tout mon temps libre, et probablement aussi une grande partie du temps que j'étais supposé passer à étudier.

Quand avez-vous connu Netfilter ? Quand avez-vous décidé de vous investir dans ce projet ?

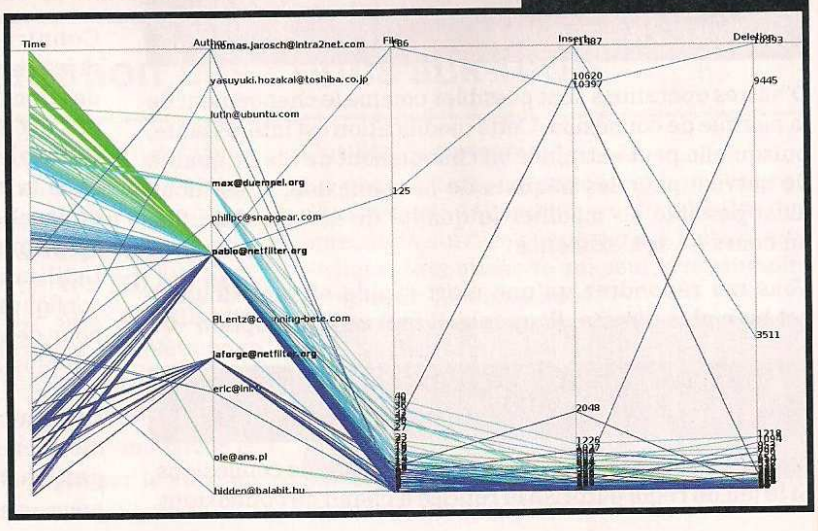
J'ai commencé à utiliser les logiciels libres du projet Netfilter lors de ma carrière dans le privé. Lorsque, en 2003, j'ai commencé à envoyer des petites contributions et des hacks, cela a été intéressant d'avoir des retours des autres, et ce, même lorsque le hack initial était très expérimental. J'ai été invité à rejoindre la Coreteam Netfilter en 2007. Depuis lors, je passe une partie de mon temps libre à la maintenance, la publication de nouvelles versions et à coder sur le projet.

Quelle question vous énerve-t-elle le plus sur une liste de discussions ?

Aucune en fait. Les questions ne sont pas un problème pour moi. Je pense que ce qui m'ennuie vraiment ce sont les attitudes inflexibles et non coopératives.

Vous êtes l'initiateur des conntrack-tools. Comment avez-vous décidé de démarrer le projet ?

Il y avait un patch qui ajoutait une interface Netlink au suivi de connexions. L'interface texte dans `/proc` était très limitée et ajouter de nouvelles fonctionnalités aurait été une bien piètre conception. J'ai donc décidé de récupérer le vieux patch et de le proposer au projet. Je pense que la principale raison pour laquelle j'ai décidé de travailler sur ce sujet était, si je me souviens bien, que je pensais que la place pour réaliser la synchronisation d'état était en espace utilisateur et non en espace noyau, en utilisant un système de message comme netlink, et donc en suivant une conception de type micro-noyau.



Représentation du travail sur les conntrack-tools

Pour ceux qui ne sont pas familiers avec Netlink, il s'agit d'un protocole de communication entre les sous-systèmes du noyau et leurs contre-parties en espace utilisateur en utilisant une sémantique proche des *sockets*.

Quel est l'état actuel du projet ?

La dernière version (0.9.10 au moment de l'interview) est en bon état. Les fonctionnalités souhaitées sont développées et il est maintenant temps de recevoir des retours et d'attendre quelques rapports de réussite avant de pouvoir sortir la 1.0.

Pouvez-vous nous en dire plus sur le mode actif-actif ?

La configuration active-active est indépendante des conntrack-tools. Elle se paramètre en utilisant iptables.

Mon environnement de test, qui est composé de deux pare-feu en *cluster* haute-disponibilité utilisant un réseau Gigabit, est capable dans la configuration actif/passif de filtrer jusqu'à 21000 connexions TCP par seconde (sur du matériel limité et peu coûteux) en réalisant du filtrage à état et de la translation d'adresse. Avec le filtre « cluster » et d'autres accessoires, la configuration actif/actif peut filtrer jusque 30000 connexions TCP par seconde, ce qui signifie un gain de 45% en termes de performance. La chose la plus importante ici est qu'il n'y a pas de perte de ressources comme dans une configuration actif/passif, puisque les deux pare-feu filtrent. Basiquement, l'approche est un partage de charge basé sur une table de hachage, ce qui est différent d'un partage de charge, puisque le partage n'est pas nécessairement équilibré.

45% de performance en plus, c'est exceptionnel !

C'est probablement plus en choisissant bien le matériel et le jeu de règles, mais je préfère être prudent avec les nombres. Je ne sais d'ailleurs rien des limites de passage à l'échelle en termes de nombre maximal de pare-feu.

Comment se positionne les conntrack-tools par rapport à pf-sync ou nf-sync ?

Bon, il me faut un tableau. Je pense tout de même que je devrais enquêter un peu plus pour être certain de toutes les informations :

	pf-sync	nf-sync	conntrackd
Support de liens dédiés multiples	Oui*	Non	Oui
Réplication des états garantie	Non	Oui**	Oui
Implémentation	Noyau	Noyau	Espace utilisateur
Réplication sélective***	Non	Non	Oui

(*) Je ne suis pas sûr de l'approche suivie par pf-sync, car je pense que si on paramètre un ensemble de liens pour propager les changements d'états, ils envoient tous les changements d'états. Conntrackd envoie seulement les mise à jour à un périphérique à la fois, même s'il y a plusieurs liens dédiés dans l'installation. Si un lien a des problèmes, il bascule alors sur un autre.

(**) nf-sync est censé utilisé un système d'acquittement, mais il souffre de limitations sévères.

(***) nf-sync et pf-sync autorisent seulement la réplication d'état en temps réel souple (il me semble pour pf-sync). Conntrackd supporte un mode batch et un mode temps réel depuis la version 0.9.10.

Conclusion

Grâce au travail de Pablo Neira Ayuso sur les conntrack-tools, Linux et Netfilter comblent l'un de leurs retards les plus importants sur la concurrence. Netfilter gagne là une fonctionnalité qui lui manquait pour s'imposer sereinement dans le monde de l'entreprise.

Les récents travaux de Pablo portent sur les outils, modules noyau et modifications d'iptables et arptables, nécessaires à l'établissement facile de systèmes de pare-feu actif-actif performants. Les patchs sont en cours de validation et Netfilter devrait donc bientôt (vraisemblablement dans 2.6.30) avoir tous les outils nécessaires pour une redondance performante et simple à mettre en œuvre.

Références

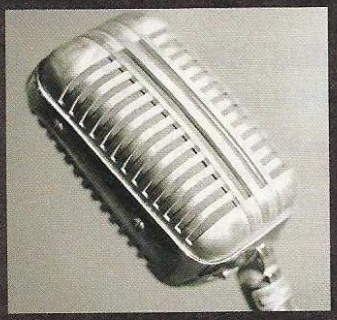
- Page des conntrack-tools : http://conntrack-tools.netfilter.org/
- **[intro]** Introduction à Netfilter et iptables, voir page 20 dans ce numéro.
- **[random]** http://software.inl.fr/trac/wiki/contribs/RandomSkype
- **[GLMF 86]** Interaction avec l'espace utilisateur

Propos recueillis par : Éric Leblond

Utilisateur GNU/Linux depuis 1996. Co-fondateur et directeur technique d'INL. Développeur principal de NuFW.

INTRODUCTION

Interview : Dave S. Miller, responsable



David S. Miller est l'une des huiles du développement noyau Linux. Il est le responsable principal de l'implémentation SPARC de Linux et le développeur principal de la pile TCP/IP. Il est également un acteur très impliqué dans le développement de GCC. Il nous fait l'honneur de répondre à nos questions.

Propos recueillis par

- Éric Leblond

GLMF : Pouvez-vous vous présenter et nous parler de la manière dont vous avez découvert Linux ?

David S. Miller : Mon nom est David S. Miller. Je vis à Seattle.

Je *hacke* des trucs en *open source* depuis à peu près 15 ans, et j'ai découvert Linux en lisant des groupes Usenet, il y a un paquet d'années.

En fait, à ce moment-là, j'étais principalement intéressé par les machines de la série Atari ST et je lisais donc les groupes en rapport avec Atari.

Le portage m68k de Linux, sur lequel travaillaient des gens comme Hamish Macdonald, y faisait l'objet de discussions. J'ai donc commencé à approfondir le sujet.

Le portage m68k était un vrai chemin de croix à installer. J'ai donc fini par me tourner vers une première installation sur le 486 de mon père. Il s'agissait d'une des premières distributions SLS (je pense que c'était la version 1.03). L'installation complète demandait une trentaine de disquettes ou quelque chose dans le genre. Si je me rappelle bien, le noyau était un 0.99pl15.

GLMF : Quand avez-vous décidé de prendre part au projet ?

DSM : Ce n'était pas une décision, c'est arrivé naturellement.

À ce moment-là, je commençais à jeter un œil à Linux. Le code en espace utilisateur passait tout juste du format binaire A.OUT à ELF. H. J. Liu et Eric Youngdale avaient achevé la plus grande partie de ce travail.

Un excellent test en espace utilisateur se résumait à la capacité à *bootstraper* Emacs. La raison de ce choix comme élément du processus de développement était qu'Emacs s'exécute lui-même, procède à quelques initialisations et, ensuite, *dumpe* l'image résultante en un exécutable final. Ce processus est appelé « *unexecing* ».

Quoi qu'il en soit, j'avais le support initial ELF qui fonctionnait sur le PC de mon père et j'essayais de faire en sorte qu'Emacs se dumpe lui-même de manière viable. J'étais coincé et j'ai donc configuré mon compte à l'université de sorte qu'Eric Youngdale puisse se connecter et faire fonctionner tout cela. Un jour où deux plus tard, c'était chose faite. Il avait réussi.

Après cela, mon intérêt s'est tourné vers le portage Sparc, puisque j'avais accès à bon nombre de matériels intéressants à la Rutgers University. J'ai fait booter Linux sur Sparc, j'ai mis en route la liste de diffusion et je me suis également occupé du code réseau.

GLMF : Quelle question vous énerve le plus sur une liste de diffusion ?

DSM : Fondamentalement, tout ce qui ressemble à des *trolls*. La plus anti-sociale des choses que vous puissiez faire au monde est d'interrompre un groupe de personnes productives et leur faire perdre leur temps.

Dans les faits, un certain nombre de personnes voient leurs adresses bloquées (au moins temporairement) dans les listes de diffusion en raison d'incidents « trollesques » assez négatifs. Ce qu'il y a de plus remarquable, c'est que quelques-uns d'entre eux essaient encore de manipuler pour voir le blocage retiré.

Par exemple, disons qu'une personne qui trolle sans cesse depuis des jours se fait bannir et que, finalement, après quelques jours ou semaines, elle comprend ce qui vient de se passer. Ainsi, elle va envoyer un message privé à des gens comme moi, Linus Torvalds, Andrew Morton, etc. Ceci en disant : « Eh ! J'essaie d'envoyer un rapport de bogue utile à la liste de diffusion et je suis banni. S'il vous plait réhabilitez-moi, je veux contribuer.» La bonne blague !

C'est manipulateur, mais les gens le font.

GLMF : Quel est le meilleur retour que vous ayez eu jusqu'ici ?

DSM : J'aime le retour que je reçois des utilisateurs et des autres personnes qui utilisent mon travail.

Mais, pour être parfaitement honnête, rien ne vaut le fait de recevoir un retour d'un pair. N'importe quoi de positif que je reçois des autres *core*-développeurs a tendance à faire mon bonheur.

GLMF : De quoi êtes-vous responsable dans le noyau Linux ?

Dans les faits, je m'occupe du réseau et du/des portages Sparc.

Le support réseau occupe la majorité de mon temps. C'est amusant, car parfois je passe tellement de temps avec la gestion des patchs qu'à la fin de la journée je n'ai plus d'énergie pour quoi que ce soit d'autre. En particulier, plus d'énergie pour mon propre travail.

Bien entendu, je ne laisse pas cela arriver trop souvent, sinon je vais devenir vieux et ennuyeux avant l'heure.

Le portage Sparc est à la fois amusant et un véritable challenge. Étant donné que j'écris la plupart du code pour le portage sparc64, n'importe quel problème est de mon fait et uniquement de mon fait :)

Je pense qu'il est important d'être extrêmement communicant en tant que responsable d'un sous-système. Lorsqu'on est en semaine, toujours au moins donner un feedback pour chaque soumission de patch dans la journée. Lorsque les gens doivent attendre plus d'un jour pour recevoir un retour sur leur travail, cela devient stressant et, parfois, cela à tendance à vraiment les énerver.

Donc, si l'objectif est de voir davantage de personnes contribuer, répondez rapidement. Vous devez vraiment être capable d'investir du temps là-dedans et

de la pile TCP/IP du noyau Linux

j'ai réalisé que la plupart des gens n'en sont pas capables. Ne vous faites pas appeler « responsable » si ce n'est pas le cas.

GLMF : Combien de temps par jour en moyenne passez-vous sur les mails ?

DSM : Cela me prend généralement une heure juste pour parcourir mes messages en début de chaque journée. Et ce n'est même pas vraiment lire quoi que ce soit. C'est tout juste trier et faire une première passe pour effacer ce qui ne m'intéresse clairement pas.

La plupart de ces messages sont destinés au *postmaster* de vger.kernel.org. Ainsi, dans cette première passe, je cherche également les problèmes de rebond et les choses de ce type.

J'ai habituellement besoin d'un autre café juste après cette première heure :-)

GLMF : De combien de temps disposez-vous pour implémenter vos propres idées ?

DSM : Bonne question.

Je fonctionne selon deux modes. Dans le premier mode, je m'attaque à la liste des patchs pour le réseau et le support Sparc de manière intensive et je ne fais presque rien sur mon propre travail.

Dans l'autre mode, je traite les patchs comme nécessaires, mais j'essaie de faire en sorte que d'autres personnes prennent soin de la gestion. Résultat, je peux passer la plupart de mon temps sur des idées qui m'intéressent.

Ainsi, dans le premier mode, je dois passer peut-être une heure ou deux au mieux. Dans le second mode, je peux passer la majorité de ma journée sur mon propre code.

Ce sont des phases qui vont et viennent. Je peux m'éclater sur l'implémentation de quelque chose qui m'est propre pendant deux semaines, puis, passer un mois à m'occuper des soumissions de patch intensément.

Cela dépend également d'où nous en sommes dans le développement du noyau. Ceci a une énorme influence sur mes priorités.

GLMF : Pouvez-vous nous en dire plus sur votre travail sur le Multiqueue Networking de Linux ?

DSM : Ceci prend énormément de temps. La base du support réseau est à jour du point de vue des facilités *multiqueue* que proposent les différents périphériques.

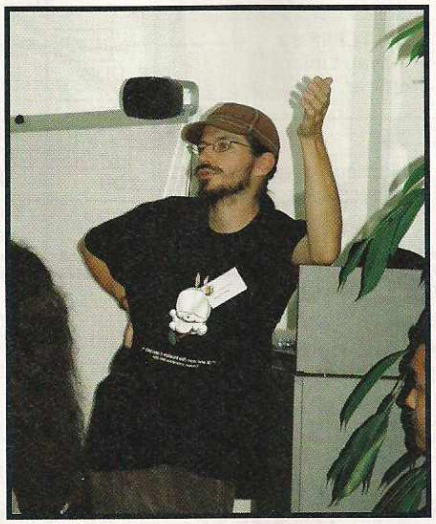
Dans un premier temps, nous avons facilité le support des pilotes de périphériques supportant le multiqueue du côté réception. Relativement parlant, c'était bien plus facile que la partie émission. C'était surtout une question de dégager les structures de données. Ce travail a été fait majoritairement par Stephen Hemminger.

Je récolte un peu plus les lauriers pour le support du côté émission multiqueue, mais, encore une fois, une importante quantité de personnes m'ont retourné des informations et corrigé les bogues. Aucun homme n'est une île (NDLR : « *No man is an island* », proverbe américain) en particulier pour les tâches importantes dans le noyau.

Le multiqueue en émission était bien plus difficile, car il a de graves implications sur le *scheduler*. Par exemple, dans une configuration multiqueue, quelle sémantique souhaitez-vous utiliser pour la classification et le contrôle de la bande passante dans le scheduler ? Est-ce que les classes de trafic sont encore par périphérique ou les considérez-vous à présent par queue ?

Ceci est important, puisqu'il y a une implication pour les performances et pour le comportement de la configuration.

Pour l'heure, nous gardons ce genre de chose en rapport avec le périphérique, comme cela a toujours été le cas. Mais, c'est un simple exemple des nombreuses questions sémantiques qui doivent trouver une réponse pour le support multiqueue en émission.



GLMF : Qu'est ce qui doit être amélioré dans Qdisc pour vous aider dans le travail sur multiqueue ? Est-ce qu'il reste un travail sur RCU (Read-copy-update) à faire comme votre dernière présentation à Seattle le laisse entendre ?

Il y a pas mal de problèmes de *locking* relativement complexes en rapport avec le chemin d'émission au travers du scheduler et du périphérique.

À la fois Qdics et le pilote de périphérique (TX) ont un locking différent. De plus, les requêtes de configuration de l'utilisateur arrivent pour le scheduler de paquets et ces requêtes demandent que l'arbre complet des objets Qdisc ne change pas durant ces opérations.

Au départ, il était établi que l'utilisation plus intensive de RCU aiderait, mais, en vérité, il est apparu qu'il était plus difficile et plus complexe si nous nous reposions uniquement sur RCU.

Nous n'utilisons RCU que pour la visibilité du pointeur vers la racine Qdisc attachée à un périphérique. Lorsque nous voulons changer la racine Qdisc, nous assignons le pointeur et attendons une période d'inactivité RCU. Cette période nous permet de nous assurer qu'aucun chemin de transmission asynchrone n'utilise encore le précédent objet Qdisc.

Pour la plupart du travail, nous utilisons le locking traditionnel. Nous partons du principe, pour l'instant, d'un point de vue Qdisc, qu'à partir du moment où vous bloquez la racine d'un arbre Qdisc avec cette méthode, tout ce qui se trouve en dessous restera en place.

GLMF : Est-il encore nécessaire de toucher au code des pilotes actuellement ?

DSM : Oui, tout le temps et pas uniquement pour les pilotes que j'ai écrits.

Nous avons beaucoup d'interfaces pour lesquelles nous cherchons des améliorations au cas par cas. À la fois parce qu'elles sont affreuses, pénibles à maintenir ou tout simplement inefficaces.

Lorsque cela se passe sur les interfaces de pilotes, il est nécessaire de toucher à pas mal de pilotes eux-mêmes.

Par exemple, si vous voulez faire des changements dans la méthode `->hard_start_xmit()`, il faut toucher à pas loin de 500 pilotes.

Il m'est déjà arrivé de faire ce genre de changement massif pour finalement me rendre compte que mon idée ne fonctionnait pas. Ainsi, j'ai dû annuler tous les changements effectués.

C'est une très bonne expérience d'un point de vue pédagogique :-)

GLMF : Est-ce que vous ou votre société prenez du temps pour jeter un œil à ce que les autres systèmes font pour influencer votre travail ?

DSM : Pas vraiment.

Je regarde dans les sources librement disponibles du noyau BSD, mais uniquement pour chercher des choses en rapport avec la sémantique des *sockets* visibles par les utilisateurs.

C'était déjà ainsi, même lorsque le noyau Linux ne couvrait pas certaines fonctionnalités. Ce n'est pas comme si j'avais pour habitude de fouiller et que j'avais arrêté en devenant gentil.

Il y a plus que suffisamment de personnes qui contribuent et apportent des idées fraîches dans le noyau Linux pour me tenir occupé.

Propos recueillis par : Éric Leblond

Introduction à Netfilter et

Afin de vous présenter la base de Netfilter, ainsi que de vous aider dans la compréhension des différents termes employés tout au long de ce hors-série, une introduction ne fera pas de mal. L'approche que nous prenons est pratique : comprendre ce qui est bloqué et utiliser les commandes permettant de le vérifier avec iptables, netcat et tcpdump.

Auteurs

- Éric Leblond
- Sébastien Tricaud

1 Introduction de l'introduction

Couches	Où ça passe vraiment	Protocoles	Applications
Physique	Lien — Carte réseau		Tournevis ;)
Liaison de données	Driver	ARP	ebtables
			tcpdump
Réseau	Pile TCP/IP du noyau Linux (Netfilter, QOS, ...)	IP — ICMP	iptables
Transport		TCP, UDP	netcat
Session		Netbios, SSH	libnet, libsmbclient
Présentation	Bibliothèques et applications	RELP — TLS, RDP, XDR	gnutls, zlib
Application		SMTP, HTTP, Finger	Thunderbird, Konqueror

Par quel bout prendre Netfilter pour en découvrir ses fonctionnalités ? Tenez, prenons le réseau. Puisqu'il s'agit de ce que nous voulons : bloquer ce que l'on appelle des **paquets**. Pour savoir comment un paquet arrive depuis une autre machine vers vos applications, il se passe plusieurs étapes :

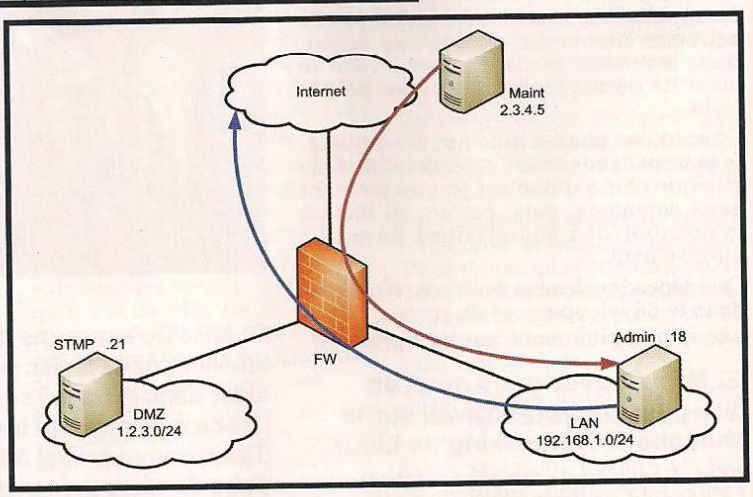
1. Une fois que la carte s'assure de l'intégrité des données, elle envoie ce signal à la **couche de liaison** qui s'assure grâce à l'adresse physique (MAC) de la carte que le signal lui est bien destiné.

2. Le signal remonte au niveau du réseau, et le couple IP source/destination est utilisé. À cette étape, il nous reste deux choix : soit l'adresse IP correspond à celle d'une interface locale, soit elle doit être **routée** vers une autre machine.

3. Puis, ce signal est **transporté** vers un protocole de transport (généralement **tcp** ou **udp**).

4. Enfin, les données sont délivrées aux **applications** à l'écoute sur ce port et cette adresse IP.

Pour donner du sens aux règles que nous allons écrire, nous choisissons le schéma suivant :



Il s'agit d'un exemple assez classique comportant un réseau local (LAN) et une zone démilitarisée (DMZ) dans laquelle se trouvent les serveurs. Un ordinateur isolé (Admin, 192.168.1.18) dans le LAN peut aller directement sur Internet pour le protocole HTTP. La DMZ ne contient qu'un seul serveur SMTP (1.2.3.18). Mais, il y a un accès peu recommandable : la machine Admin doit être accessible depuis la machine Maint située à l'extérieur (2.3.4.5) sur ssh (port 22) pour la maintenance.

iptables

1.1 Organisation des règles

1.1.1 Filtrage

Netfilter est l'implémentation au niveau du noyau du pare-feu Linux. Iptables est sa petite sœur le manipulant. Netfilter a été conçu avec comme idée d'écrire des règles de pare-feu de manière aussi aisée que l'écriture d'un schéma tel que vous le voyez ci-dessous. Un peu comme si vous le décriviez oralement :

1. Je veux autoriser à tout le monde d'accéder à mon serveur web.
2. Je veux autoriser mon pare-feu à *pinguer* sur internet.
3. Je veux que les utilisateurs dans mon LAN puissent utiliser le web.
4. J'autorise une exception pour telle adresse IP pour accéder à mon serveur SSH.

Il est facile de classifier les paquets gérés par le pare-feu en trois catégories :

- paquets à destination du pare-feu (4ème cas) ;
- paquets passant par le pare-feu (1er et 3ème cas) ;
- paquets émis par le pare-feu (2ème cas).

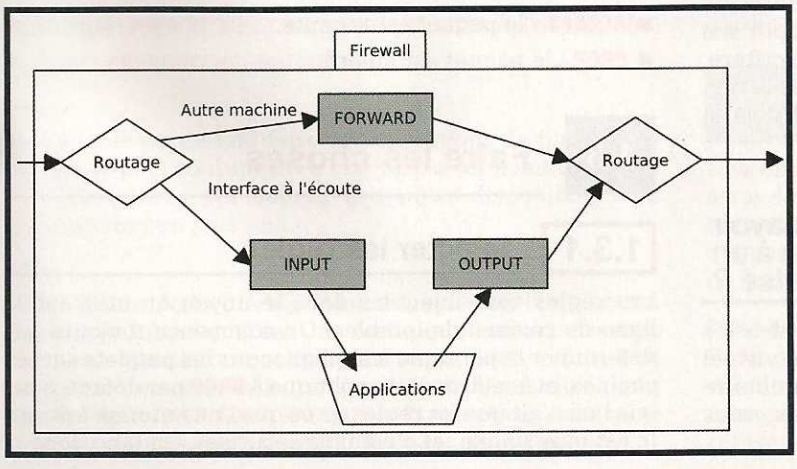
Si nous devions choisir un mot pour décrire les trois cas ci-dessus, nous prendrions :

- entrée (**INPUT**) ;
- aiguiller (**FORWARD**) ;
- sortie (**OUTPUT**).

Quand un paquet arrive sur l'interface, le noyau doit prendre une décision de routage. Le paquet qui arrive n'a que deux cas possibles :

- Il est à destination d'une interface locale (**INPUT**).
- Il est à destination d'une autre machine (**FORWARD**).

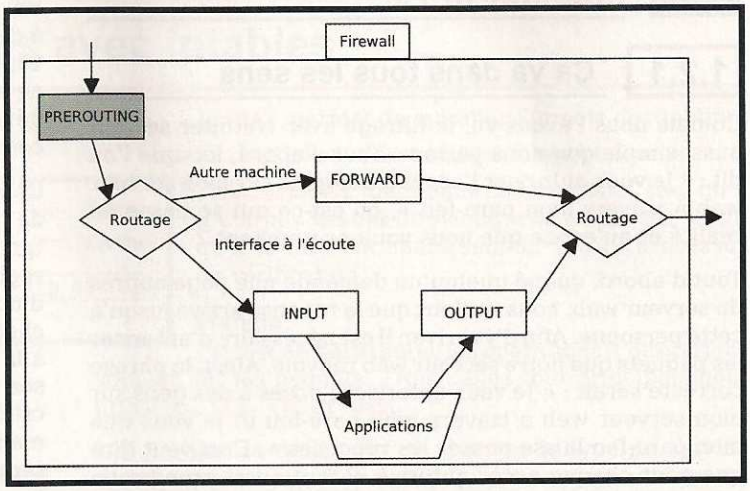
Quant aux paquets partant depuis le pare-feu, ils sont dans le sens de la sortie (OUTPUT). Ce qui donne en schéma :



1.1.2 Translation d'adresse réseau (NAT) de destination

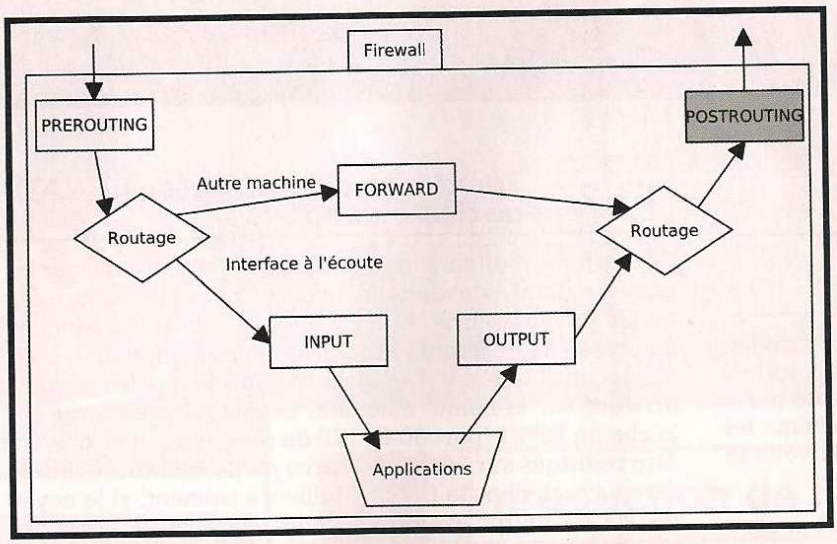
Ce que nous voulons maintenant faire, c'est de rediriger les paquets allant à destination du port 80 sur l'IP du pare-feu vers le serveur web de la DMZ. Cela s'appelle de la traduction d'adresse de destination tout simplement parce que nous changeons l'adresse IP de destination afin que les paquets arrivent sur la bonne machine. Les paquets arrivent sur la chaîne **INPUT** (port 80 de l'IP du pare-feu), mais doivent être redirigés sur **FORWARD** (parce qu'ils sont routés sur le serveur web dans la DMZ). Malheureusement, si le noyau décide de router avant que l'adresse IP destination des paquets ne soit changée, il va y avoir un problème de logique.

Heureusement, Netfilter permet grâce à la cible **DNAT** (Destination NAT) de changer l'adresse IP de destination. Il est cependant nécessaire de le faire avant que la décision de routage soit prise. À cette étape, nous sommes positionnés dans la chaîne de pré-routage (**PREROUTING**) :



1.1.3 Translation d'adresse réseau (NAT) source

Dans le cas contraire à celui du DNAT, si l'on veut permettre à un utilisateur du LAN d'aller sur un serveur web sur internet, les paquets auront leur IP source modifiée par le pare-feu. Ainsi, les paquets qui passeront par la chaîne **FORWARD** seront modifiés pour faire comme s'ils provenaient du pare-feu, ce qui aurait voulu dire qu'ils proviennent de la chaîne **OUTPUT**. Si l'adresse est modifiée avant ces deux chaînes, nous aurons des problèmes parce que les paquets ne proviennent pas de la pile réseau du pare-feu. Il est donc nécessaire d'agir après ces deux chaînes pour non seulement garder une certaine liberté, mais, surtout, ne pas embrouiller l'algorithme de routage. La modification se fera après la décision de routage, ce qui nous donne le schéma final des chaînes de Netfilter :



Ainsi, le code de Netfilter agit sur la chaîne de **POSTROUTING** pour changer les en-têtes relatives à la source (en faisant du SNAT) ou pour faire d'autres choses assez amusantes.

1.2 Filtrage à états, pour vous faciliter la vie

1.2.1 Ça va dans tous les sens

Comme nous l'avons vu, le filtrage avec Netfilter se veut aussi simple que nous parlons. Tout d'abord, lorsque l'on dit : « Je veux autoriser l'accès à des gens sur mon serveur web à travers mon pare-feu », qu'est-ce qui se passe en réalité et qu'est-ce que nous voulons vraiment ?

Tout d'abord, quand quelqu'un demande une page auprès du serveur web, nous voulons que la réponse arrive jusqu'à cette personne. Afin d'y arriver, il est nécessaire d'autoriser les paquets que notre serveur web renvoie. Ainsi, la phrase correcte serait : « Je veux autoriser l'accès à des gens sur mon serveur web à travers mon pare-feu et je veux que mon pare-feu laisse passer les réponses ». Ceci veut dire que pour chaque accès autorisé, il faut aussi prendre en compte la réponse. Ceci rend le pare-feu ennuyeux, et, du coup, il n'y a plus d'intérêt à continuer à lire ce hors-série. Nan ! Attendez ! « Pour chaque autorisation d'accès, nous voulons prendre en compte la réponse », c'est la réponse !

La phrase « J'autorise les paquets étant des réponses de quelque chose que j'ai déjà autorisé avant » remplit son rôle à merveille. Chaque réponse correspond à ce critère. Maintenant, il ne reste plus qu'à savoir si Netfilter le fait. Vous en doutez ? Vous connaissez certainement déjà la réponse !

1.2.2 Comment Netfilter fait-il pour savoir qu'un paquet est une réponse à un paquet précédemment autorisé ?

Le noyau maintient une table qui enregistre toutes les sessions. Cette table peut être lue depuis le fichier virtuel `/proc/net/ip_conntrack` ou mieux, en utilisant l'utilitaire conntrack (exemple, pour suivre les évènements, vous pouvez essayer `conntrack -E`).

Quand un paquet arrive sur une interface, Netfilter regarde à l'en-tête IP afin de voir si ce paquet fait partie d'une session connue. En fonction du cas, il détermine l'état du paquet parmi les cas suivants :

- **NEW** : le paquet n'est lié à aucune session.
- **ESTABLISHED** : la session existe au niveau du protocole de la couche transport comme une session TCP.
- **RELATED** : la session existe de façon relative à une autre, par exemple une session FTP ou le *pong* qui vient après le *ping*.
- **INVALID** : la session est invalide et ne rentre dans aucun des cas cités au-dessus

De cette manière, la phrase « J'autorise les paquets étant des réponses de quelque chose que j'ai déjà autorisé avant » est traduite en « J'autorise les paquets avec l'étant **NEW** et/ou **ESTABLISHED** à passer le pare-feu ».

1.2.3 Mais qu'est-ce vraiment qu'une règle Netfilter ?

Réutilisons une de nos phrases : « Je veux autoriser l'accès à des gens sur mon serveur web à travers mon pare-feu ». Une seule règle Netfilter suffira pour dire cela au noyau. Il ne s'agit que d'une décision sur les paquets. Une décision générique est juste « je veux que tous les paquets qui correspondent à ce critère aient la destinée que je choisis ici ».

Comme nous l'avons vu plus haut, nous avons beaucoup de tables et de chaînes sur lesquelles nous pouvons agir (principalement **INPUT**, **OUTPUT** et **FORWARD**). Ainsi, une règle est spécifique à une table. Comme nous avons besoin d'utiliser plus qu'une seule règle, le noyau maintient pour chaque table la liste des règles que nous avons rajoutées à la table. Alors, quand un paquet arrive sur une chaîne, son sort est décidé en regardant s'il correspond à l'un des critères d'une des règles de la chaîne. Cela se fait de la manière la plus simple qui soit. Le noyau prend les règles séquentiellement dans l'ordre qui lui a été donné. Ainsi, l'ordre est crucial, car dès qu'un paquet correspond à une règle, la première que le noyau trouvera sera déterminante. Si aucune règle ne correspond, il faut décider de ce qui sera fait et on lui applique la politique par défaut sur les paquets dans cette chaîne. Cette politique peut-être :

- **ACCEPT** : le paquet est accepté.
- **DROP** : le paquet est ignoré.

1.3 Faire les choses

1.3.1 Injecter les règles

Les règles sont injectées dans le noyau en utilisant la ligne de commande iptables. On commence toujours par déterminer la politique à appliquer sur les paquets sur les chaînes, et à mettre cette politique à **DROP** par défaut, pour que l'on n'ait que les règles de ce que l'on autorise à écrire (c'est plus simple, et c'est plus sécurisé). On tape donc :

```
iptables -P FORWARD DROP
iptables -P INPUT DROP
iptables -P OUTPUT DROP
```

1.3.2 Utiliser le suivi de connexion

```
iptables -A FORWARD -m state --state RELATED,ESTABLISHED -j ACCEPT
iptables -A INPUT -m state --state RELATED,ESTABLISHED -j ACCEPT
iptables -A OUTPUT -m state --state RELATED,ESTABLISHED -j ACCEPT
```

1.3.3 Ajouter les règles

Autorisons un accès à notre serveur SMTP depuis n'importe où :

```
iptables -A FORWARD -m state --state NEW -d 1.2.3.21 -p tcp --dport smtp -j ACCEPT
```

Autorisons l'accès à notre client vers le serveur web :

```
iptables -A FORWARD -m state --state NEW -s 192.168.1.18 -p tcp --dport http -j ACCEPT
```

Et la dernière règle pour l'accès à l'administration par SSH de la machine :

```
iptables -A FORWARD -m state --state NEW -s 2.3.4.5 -d 192.168.1.18 -p \
tcp --dport 22 -j ACCEPT
```

1.3.4 Gérer les traductions d'adresses

Il est nécessaire de faire la traduction de paquet pour notre admin depuis le réseau externe du pare-feu (noté **IP_PUBLIQUE**) :

```
iptables -t nat -A POSTROUTING -m state --state NEW -s 192.168.1.18 -p \
tcp --dport http -j SNAT --to $IP_PUBLIQUE
```

Vous remarquez ici le rajout de **-t nat** ? il s'agit en fait de l'utilisation explicite de la table nat. Il existe plusieurs tables que l'on peut utiliser pour manipuler les différentes fonctionnalités de Netfilter. Ces tables sont expliquées en détail dans la section suivante. Il nous faut faire la même chose pour les connexions entrantes depuis la maintenance :

```
iptables -A FORWARD -m state --state NEW -s 2.3.4.5 -d $PUBLIC_IP -p \
tcp --dport 22 -j DNAT --to 192.168.1.18
```

Attention, en cas d'utilisation du *forwarding*, veuillez bien vous assurer que le sysctl **/proc/sys/net/ipv4/ip_forward** contient le chiffre « **1** », car le forwarding est désactivé par défaut. Si vous voulez l'activer de façon permanente au démarrage, il suffit d'éditer **/etc/sysctl.conf** et de positionner la variable **net.ipv4.ip_forward** à 1.

2 Manipulation des tables avec iptables

Pour rappel, voici un résumé du vocabulaire lorsque l'on utilise la commande **iptables** :

```
root@quinificator:~# iptables -t filter -L -n          Table
Chain INPUT (policy ACCEPT)                            Chaine
target     prot opt source          destination
DROP       tcp  --  0.0.0.0/0       0.0.0.0/0       tcp dpt:23
                                                       Regle
```

2.1 Les tables

Plusieurs tables différentes peuvent être utilisées. Chaque table contient plusieurs chaînes prédéfinies. Elles peuvent aussi être personnalisées. Ces chaînes prédéfinies sont :

1. Filter : **INPUT**, **FORWARD**, **OUTPUT**
2. Nat : **PREROUTING**, **POSTROUTING**, **OUTPUT**
3. Mangle : **PREROUTING**, **INPUT**, **FORWARD**, **OUTPUT**, **POSTROUTING**
4. Raw : **PREROUTING**, **OUTPUT**

Filter

Cette table permet de faire des opérations de filtrage, ce qui est le plus courant avec iptables, à tel point que c'est la table utilisée par défaut. Les cibles disponibles sont décrites un peu plus loin.

Nat

Nat signifie *Network Address Translation* (Traduction d'Adresse Réseau). C'est une technique qui entre autres permet à plusieurs machines de se connecter en même temps via une passerelle (qui fait le NAT). Trois cibles sont disponibles : **DNAT**, **SNAT**, **MASQUERADE**. Attention, cette table n'est atteinte que sur les paquets étant dans l'état **NEW**.

1. **DNAT** permet de modifier l'ip/port destination.
2. **SNAT** permet de modifier l'ip/port source.
3. **MASQUERADE** fonctionne comme SNAT, sauf qu'il n'y a pas besoin de configurer toutes les IP, vu qu'il le fera automatiquement via un processus de vérification.

Mangle

Permet de faire toutes les opérations d'altérations voulues sur les paquets. Trois cibles sont principalement utilisées : **TOS**, **TTL** et **MARK**.

TOS permet de définir ou de changer le type de service d'un paquet en vue d'une politique de routage ultérieure.

TTL permet de changer le champ de la durée de vie d'un paquet.

MARK permet de marquer un paquet afin de l'utiliser ensuite dans une application utilisateur et de faire de la limitation d'utilisation de la bande passante (*traffic shaping*).

2.2 Chaînes

Une chaîne est une liste de règles qui correspond à une série de paquets. Chaque règle spécifie ce qu'il faut faire lorsque qu'un paquet lui correspond. Cela s'appelle la cible, qui peut aussi être un saut vers une chaîne personnalisée de la même table.

Mise en place d'une chaîne personnalisée

Il est possible d'avoir, en plus des chaînes proposées par défaut par iptables, d'en avoir des personnalisées. Si, par exemple, nous voulons journaliser tous les paquets qui sont refusés, il est beaucoup plus simple de faire :

```
root@quinificator:~# iptables -N LOG_WWD
root@quinificator:~# iptables -A LOG_WWD -j LOG --log-prefix "Mon Log perso : "
root@quinificator:~# iptables -A LOG_WWD -j DROP
root@quinificator:~# iptables -L -n
Chain INPUT (policy ACCEPT)
target     prot opt source               destination

Chain FORWARD (policy ACCEPT)
target     prot opt source               destination

Chain OUTPUT (policy ACCEPT)
target     prot opt source               destination

Chain LOG_WWD (0 references)
target     prot opt source               destination
LOG        all  --  0.0.0.0/0            0.0.0.0/0           LOG flags
0 level 4 prefix `Mon Log perso : '
DROP       all  --  0.0.0.0/0            0.0.0.0/0
```

Ensuite, nous pouvons utiliser cette chaîne pour journaliser et supprimer tout ce qui arrive sur le port 3538 :

```
root@quinificator:~# iptables -A INPUT -p tcp --dport 3538 -j LOG_WWD
root@quinificator:~# iptables -L -n
Chain INPUT (policy ACCEPT)
target     prot opt source               destination
LOG_WWD    tcp  --  0.0.0.0/0            0.0.0.0/0           tcp dpt:3538

Chain FORWARD (policy ACCEPT)
...
```

3 Les cibles d'iptables

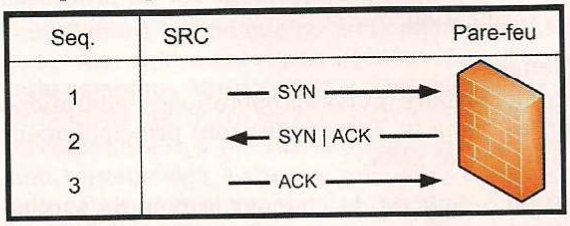
Une cible est une action entreprise suite à la correspondance d'un paquet sur une règle. Il existe deux types de cibles :

- **Terminales** : celles qui donne le verdict à appliquer au paquet pour qu'il poursuive ou stoppe sa progression dans le noyau. C'est la minorité, mais ce sont aussi celles que vous connaissez le mieux : **ACCEPT**, **DROP**, **REJECT** ou encore **NFQUEUE**.
- **Non-Terminales** : celles qui peuvent servir pour journaliser (**LOG**), modifier le paquet (**TTL**), remonter à la chaîne parente (**RETURN**), etc.

Voici les cibles disponibles :

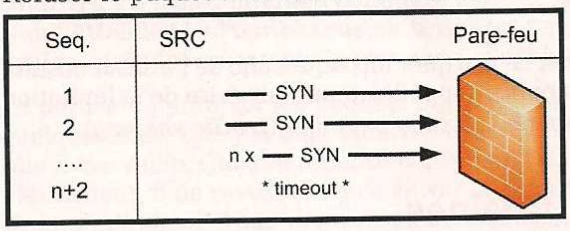
ACCEPT

Accepter le paquet.



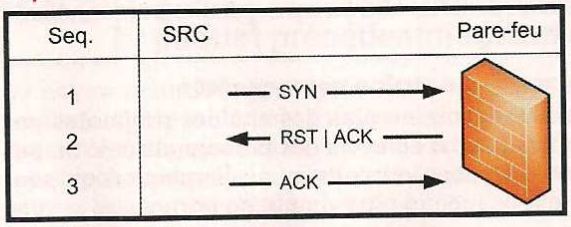
DROP

Refuser le paquet sans notifier la source.



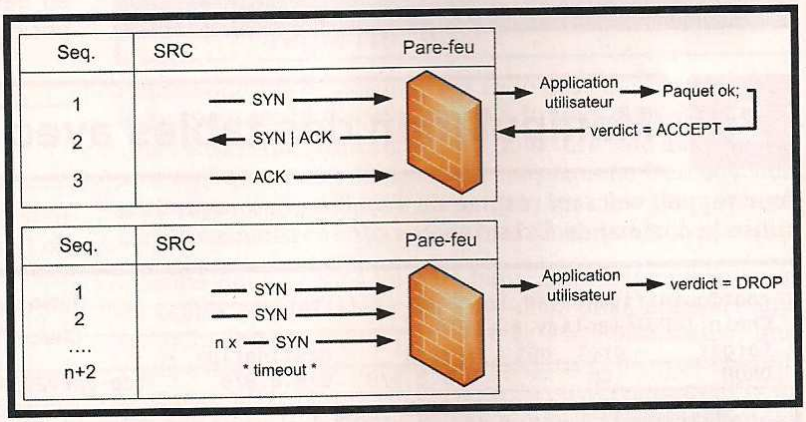
REJECT

Refuser le paquet en notifiant la source. Ici, le RST arrive si l'on ajoute l'option **--reject-with tcp-rst** à la cible.



QUEUE/NFQUEUE

Envoyer le paquet en espace utilisateur pour décider à ce niveau s'il doit être accepté ou refusé.



RETURN

Stoppe le parcours de la chaîne courante et continue de parcourir la chaîne parente.

Exemple : dans ce cas précis, nous ne parcourons pas la chaîne **LOG_WWD** tout simplement parce que **INPUT** a pour cible **RETURN** juste au-dessus.

```
root@quinificator:~# iptables -F
root@quinificator:~# iptables -A INPUT -p tcp --dport 3538 -j LOG_WWD
root@quinificator:~# iptables -I INPUT -p tcp --dport 3538 -j RETURN
root@quinificator:~# iptables -L -n
Chain INPUT (policy ACCEPT)
target     prot opt source               destination
RETURN     tcp  --  0.0.0.0/0            0.0.0.0/0           tcp dpt:3538
LOG_WWD    tcp  --  0.0.0.0/0            0.0.0.0/0           tcp dpt:3538

Chain FORWARD (policy ACCEPT)
target     prot opt source               destination

Chain OUTPUT (policy ACCEPT)
target     prot opt source               destination

Chain LOG_WWD (1 references)
target     prot opt source               destination
LOG        all  --  0.0.0.0/0            0.0.0.0/0           LOG flags 0 level 4 prefix `Mon Log perso : '
DROP       all  --  0.0.0.0/0            0.0.0.0/0
```

■ **LOG**

Écrire dans le *syslog* les informations ainsi que le sort réservé au paquet. Par exemple, nous voulons journaliser tous les paquets qui arrivent sur le port 31337.

```
root@quinificator:~# iptables -A INPUT -p tcp --dport 31337 -j LOG \
--log-prefix "Mon Log perso : "
```

produira la ligne suivante dans **/var/log/syslog** :

```
Nov  9 15:30:13 localhost kernel: [17197662.000000] Mon Log perso :
IN=lo OUT= MAC=00:00:00:00:00:00:00:00:00:00:00:00:08:00 SRC=127.0.0.1
DST=127.0.0.1 LEN=60 TOS=0x00 PREC=0x00 TTL=64 ID=29142 DF PROTO=TCP
SPT=3302 DPT=31337 WINDOW=32792 RES=0x00 SYN URGP=0
```

■ **ULOG** ou **NFLOG**

Permet d'écrire dans une application utilisateur comme MySQL au lieu de syslog les informations de log d'un paquet comme ce que l'on peut voir avec la cible **LOG**. Pour le faire, il utilise un socket netlink sur laquelle il faut s'enregistrer.

■ **DNAT**

Utilisé pour le NAT afin d'altérer l'IP et/ou le port de destination.

■ **SNAT**

Utilisé pour le NAT afin d'altérer l'IP et/ou le port source.

■ **MARK**

Permet de marquer un paquet pour ensuite récupérer cette marque pour faire du traffic shaping ou l'utiliser en espace utilisateur (**QUEUE**).

■ **MASQUERADE**

Utilisé pour les passerelles pour ne pas avoir à configurer toutes les IP à la main pour faire du SNAT.

■ **MIRROR**

Inverse la source et la destination du paquet. À ne pas utiliser.

■ **REDIRECT**

Permet de rediriger une connexion vers un(e) autre machine/port, exactement comme ce que fait **DNAT**, sans pouvoir modifier l'adresse IP. Mais, on peut mettre plusieurs ports de destination.

■ **TOS**

Utilisé pour la table mangle afin de modifier le type de service. Sert pour le routage, mais reste assez dangereux, car les routeurs ne l'interprètent pas tous de la même façon.

■ **TTL**

Permet à la table mangle de modifier la **TTL**.

4 Un peu de pratique

4.1 Exemple 1 : Nous fermons le port telnet en entrée :

```
root@quinificator:~# iptables -A INPUT -p tcp --dport telnet -j DROP
```

Nous vérifions les tables, ainsi que les statistiques sur les paquets acceptés, ainsi que ceux qui traversent la règle que l'on vient de définir.

```
root@quinificator:~# iptables -vL -n
Chain INPUT (policy ACCEPT 7 packets, 527@ bytes)
 pkts bytes target   prot opt in   out   source       destination
    @     @ DROP     tcp  --  *    *     @.@.@.@/@    @.@.@.@/@     tcp dpt:23

Chain FORWARD (policy ACCEPT @ packets, @ bytes)
 pkts bytes target   prot opt in   out   source       destination

Chain OUTPUT (policy ACCEPT 6 packets, 353 bytes)
 pkts bytes target   prot opt in   out   source       destination
```

Petite parenthèse : jusqu'à présent, pour simplifier, le paramètre **-v** n'était pas utilisé. Mais, il reste pourtant essentiel pour la compréhension de la bonne mise en place de vos règles : tout à gauche, il y a la colonne **pkts** qui indique le nombre de paquets qui sont passés par votre règle. Ainsi, il est facile de voir si vous avez mis une règle correctement en place. Ça peut s'avérer utile si vous souhaitez ajouter des règles sur un pare-feu déjà en production. Utilisez donc le mode verbeux, ça vous évitera bien des problèmes.

Maintenant que la règle est en place, afin de la vérifier, nous utilisons netcat d'un côté (l'option **-l** permet d'écouter) :

```
root@quinificator:~# nc -l -p 23
```

et **telnet** de l'autre (on aurait aussi pu utiliser netcat en mode client) :

```
toady@quinificator:~$ telnet localhost
Trying 127.0.0.1...
C'est tout bon ?

toady@quinificator:~$
```

Quant à **tcpdump** pour vérifier :

```
root@quinificator:~# tcpdump -i any -lpn -s@ -X 'port 23'
tcpdump: verbose output suppressed, use -v or -vv for full protocol decode
listening on any, link-type LINUX_SLL (Linux cooked), capture size 65535 bytes
11:19:37.204089 IP 127.0.0.1.4545 > 127.0.0.1.23: S 2218329157:2218329157(@) win 32792
        0x0000:  4510 003c c52e 4000 4006 777b 7f00 0001  E..<..@.@.w{....
        0x0010:  7f00 0001 11c1 0017 8439 0445 0000 0000  .........9.E....
        0x0020:  a002 8018 33b6 0000 0204 400c 0402 080a  ....3.....@.....
        0x0030:  0019 c16b 0000 0000 0103 0303            ...k........
11:19:40.200542 IP 127.0.0.1.4545 > 127.0.0.1.23: S 2218329157:2218329157(@) win 32792
        0x0000:  4510 003c c52f 4000 4006 777a 7f00 0001  E..<./@.@.wz....
        0x0010:  7f00 0001 11c1 0017 8439 0445 0000 0000  .........9.E....
        0x0020:  a002 8018 30c8 0000 0204 400c 0402 080a  ....@.....@.....
        0x0030:  0019 c459 0000 0000 0103 0303            ...Y........
11:19:46.200918 IP 127.0.0.1.4545 > 127.0.0.1.23: S 2218329157:2218329157(@) win 32792
        0x0000:  4510 003c c530 4000 4006 7779 7f00 0001  E..<.0@.@.wy....
        0x0010:  7f00 0001 11c1 0017 8439 0445 0000 0000  .........9.E....
        0x0020:  a002 8018 2aec 0000 0204 400c 0402 080a  ....*.....@.....
        0x0030:  0019 ca35 0000 0000 0103 0303            ...5........

3 packets captured
9 packets received by filter
@ packets dropped by kernel
```

Nous vérifions que Netfilter a bien reçu les trois paquets :

```
root@quinificator:~# iptables -vL -n
Chain INPUT (policy ACCEPT 573 packets, 113K bytes)
 pkts bytes target   prot opt in   out  source       destination
    3   180 DROP     tcp  --  *    *    0.0.0.0/0    0.0.0.0/0       tcp dpt:23

Chain FORWARD (policy ACCEPT 0 packets, 0 bytes)
 pkts bytes target   prot opt in   out  source       destination

Chain OUTPUT (policy ACCEPT 488 packets, 45851 bytes)
 pkts bytes target   prot opt in   out  source       destination
root@quinificator:~#
```

4.2 Exemple 2 : À l'opposé, si le port telnet est ouvert en entrée, cela donne :

```
root@quinificator:~# iptables -F
root@quinificator:~# iptables -A INPUT -p tcp --dport telnet -j ACCEPT

toady@quinificator:~$ telnet localhost
Trying 127.0.0.1...
Connected to localhost.localdomain.
Escape character is '^]'.
C'est tout bon ?

root@quinificator:~# tcpdump -i any -lpn -s0 -X 'port 23'
tcpdump: verbose output suppressed, use -v or -vv for full protocol decode
listening on any, link-type LINUX_SLL (Linux cooked), capture size 65535 bytes
11:26:34.034778 IP 127.0.0.1.2656 > 127.0.0.1.23: S 2647215082:2647215082(0) win 32792
        0x0000:  4510 003c 139f 4000 4006 290b 7f00 0001  E..<..@.@.).....
        0x0010:  7f00 0001 0a60 0017 9dc9 4bea 0000 0000  .....`....K.....
        0x0020:  a002 8018 42d6 0000 0204 400c 0402 080a  ....B.....@.....
        0x0030:  001b 5875 0000 0000 0103 0303            ..Xu........
11:26:34.035114 IP 127.0.0.1.23 > 127.0.0.1.2656: S 2647673144:2647673144(0) ack
2647215083 win 32768
        0x0000:  4500 003c 0000 4000 4006 3cba 7f00 0001  E..<..@.@.<.....
        0x0010:  7f00 0001 0017 0a60 9dd0 4938 9dc9 4beb  .......`..I8..K.
        0x0020:  a012 8000 0344 0000 0204 400c 0402 080a  .....D....@.....
        0x0030:  001b 5875 001b 5875 0103 0303            ..Xu..Xu....
11:26:34.035359 IP 127.0.0.1.2656 > 127.0.0.1.23: . ack 1 win 4099
        0x0000:  4510 0034 13a0 4000 4006 2912 7f00 0001  E..4..@.@.).....
        0x0010:  7f00 0001 0a60 0017 9dc9 4beb 9dd0 4939  .....`....K...I9
        0x0020:  8010 1003 dc61 0000 0101 080a 001b 5875  .....a........Xu
        0x0030:  001b 5875                                ..Xu
11:26:34.035809 IP 127.0.0.1.2656 > 127.0.0.1.23: P 1:28(27) ack 1 win 4099
        0x0000:  4510 004f 13a1 4000 4006 28f6 7f00 0001  E..O..@.@.(.....
        0x0010:  7f00 0001 0a60 0017 9dc9 4beb 9dd0 4939  .....`....K...I9
        0x0020:  8018 1003 fe43 0000 0101 080a 001b 5875  .....C........Xu
        0x0030:  001b 5875 fffd 03ff fb18 fffb 1fff fb20  ..Xu........... 
        0x0040:  fffb 21ff fb22 fffb 27ff fd05 fffb 23    ..!..''..'.....#
11:26:34.035821 IP 127.0.0.1.23 > 127.0.0.1.2656: . ack 28 win 4096
        0x0000:  4500 0034 114c 4000 4006 2b76 7f00 0001  E..4.L@.@.+v....
        0x0010:  7f00 0001 0017 0a60 9dd0 4939 9dc9 4c06  .......`..I9..L.
        0x0020:  8010 1000 dc49 0000 0101 080a 001b 5875  .....I........Xu
        0x0030:  001b 5875                                ..Xu
11:26:37.243550 IP 127.0.0.1.2656 > 127.0.0.1.23: P 28:46(18) ack 1 win 4099
        0x0000:  4510 0046 13a2 4000 4006 28fe 7f00 0001  E..F..@.@.(.....
        0x0010:  7f00 0001 0a60 0017 9dc9 4c06 9dd0 4939  .....`....L...I9
        0x0020:  8018 1003 fe3a 0000 0101 080a 001b 5b97  .....:........[.
        0x0030:  001b 5875 4327 6573 7420 746f 7574 2062  ..XuC'est.tout.b
        0x0040:  6f6e 203f 0d0a                           on.?..
11:26:37.243569 IP 127.0.0.1.23 > 127.0.0.1.2656: . ack 46 win 4096
        0x0000:  4500 0034 114d 4000 4006 2b75 7f00 0001  E..4.M@.@.+u....
        0x0010:  7f00 0001 0017 0a60 9dd0 4939 9dc9 4c18  .......`..I9..L.
        0x0020:  8010 1000 d5f3 0000 0101 080a 001b 5b97  ..............[.
        0x0030:  001b 5b97                                ..[.

root@quinificator:~# nc -l -p 23
C'est tout bon ?
```

On voit que nous avons 7 paquets qui sont passés par cette règle.

```
root@quinificator:~# iptables -vL -n
Chain INPUT (policy ACCEPT 1423 packets, 285K bytes)
 pkts bytes target   prot opt in   out  source       destination
    7   419 ACCEPT   tcp  --  *    *    0.0.0.0/0    0.0.0.0/0       tcp dpt:23

Chain FORWARD (policy ACCEPT 0 packets, 0 bytes)
 pkts bytes target   prot opt in   out  source       destination

Chain OUTPUT (policy ACCEPT 1244 packets, 114K bytes)
 pkts bytes target   prot opt in   out  source       destination
```

Pour effacer la règle que l'on vient de faire, il suffit de l'effacer avec le numéro qui l'identifie.

```
root@quinificator:~# iptables -D INPUT 1
root@quinificator:~# iptables -L -n
Chain INPUT (policy ACCEPT)
target     prot opt source               destination

Chain FORWARD (policy ACCEPT)
target     prot opt source               destination

Chain OUTPUT (policy ACCEPT)
target     prot opt source               destination
```

On peut utiliser l'option **`--line-numbers`**, qui affichera le numéro de chaque règle.

5 États

Dans le cas d'un firewall dit « *stateful* », tel Netfilter, les états permettent de suivre le sens d'une connexion. On peut, par exemple, accepter une connexion entrante uniquement si elle est la réponse d'une communication sortante. Ou inversement. Ou encore refuser une connexion invalide (Attention, invalide a une définition très précise, voir ci-dessous).

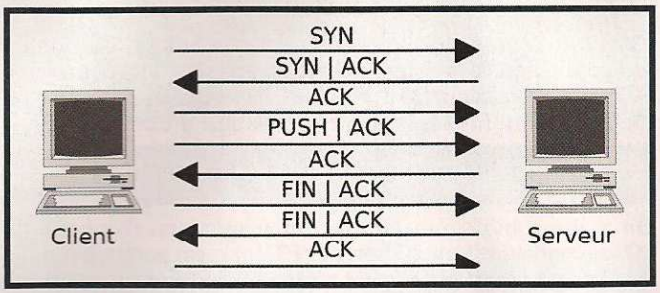
Nous avons déjà vu dans l'introduction les 4 états : **`INVALID`**, **`ESTABLISHED`**, **`NEW`** ou **`RELATED`**. Nous allons les mettre en pratique et regarder ce qui se passe. Mais avant, il serait bien de revoir un peu le mode de fonctionnement de TCP. Vous pouvez lister à tout moment l'état du conntrack grâce à la commande des conntrack-tools **`conntrack -L`**.

5.1 Bref rappel sur TCP

Si nous regardons une session simple sur TCP qui se termine proprement, nous voyons ceci (surtout la sixième colonne) :

```
12:35:49.004962 IP 127.0.0.1.1501 > 127.0.0.1.9999: S 2746511017:2746511017(0) win 32792
12:35:49.062930 IP 127.0.0.1.9999 > 127.0.0.1.1501: S 2737174071:2737174071(0) ack 2746511018 win 32768
12:35:49.062996 IP 127.0.0.1.1501 > 127.0.0.1.9999: .ack 1 win 4099
12:35:49.932459 IP 127.0.0.1.1501 > 127.0.0.1.9999: P 1:6(5) ack 1 win 4099
12:35:49.932483 IP 127.0.0.1.9999 > 127.0.0.1.1501: .ack 6 win 4096
12:35:50.429504 IP 127.0.0.1.1501 > 127.0.0.1.9999: F 6:6(0) ack 1 win 4099
12:35:50.429973 IP 127.0.0.1.9999 > 127.0.0.1.1501: F 1:1(0) ack 7 win 4096
12:35:50.429989 IP 127.0.0.1.1501 > 127.0.0.1.9999: .ack 2 win 4099
```

La trace **tcpdump** peut se résumer à l'image ci-dessous :



En fonction du flux TCP, le suivi de connexion (*conntrack*) saura si un paquet est ou non une réponse valide. Bien sûr, cela simplifie la réalité, car tout n'est pas aussi simple que cela en a l'air. Par exemple, TCP utilise différents *timeouts*, qui sont utilisés par le conntrack pour savoir à quel endroit dans les quatre états possibles le paquet ira. Ci-dessous, la liste des timeouts utilisés par les différents états du conntrack :

```
#define SECS * HZ
#define MINS * 60 SECS
#define HOURS * 60 MINS
#define DAYS * 24 HOURS

unsigned int ip_ct_tcp_timeout_syn_sent =      2 MINS;
unsigned int ip_ct_tcp_timeout_syn_recv =     60 SECS;
unsigned int ip_ct_tcp_timeout_established =   5 DAYS;
unsigned int ip_ct_tcp_timeout_fin_wait =      2 MINS;
unsigned int ip_ct_tcp_timeout_close_wait =   60 SECS;
unsigned int ip_ct_tcp_timeout_last_ack =     30 SECS;
unsigned int ip_ct_tcp_timeout_time_wait =     2 MINS;
unsigned int ip_ct_tcp_timeout_close =        10 SECS;
```

Et voici la liste des états internes à TCP que le noyau donne au système de suivi de connexion :

- **NONE: initial state**
- **SYN_SENT: SYN-only packet seen**
- **SYN_RECV: SYN-ACK packet seen**
- **ESTABLISHED: ACK packet seen**
- **FIN_WAIT: FIN packet seen**
- **CLOSE_WAIT: ACK seen (after FIN)**
- **LAST_ACK: FIN seen (after FIN)**
- **TIME_WAIT: last ACK seen**
- **CLOSE: closed connection**

5.2 Un contrôle d'état UDP ?

Dans les sources du noyau, dans le fichier **linux/net/netfilter/nf_conntrack_proto_udp.c**, on a les deux définitions suivantes :

```
unsigned int nf_ct_udp_timeout = 30*HZ;
unsigned int nf_ct_udp_timeout_stream = 180*HZ;
```

La première, **nf_ct_udp_timeout**, signifie à partir de quel moment on estime une connexion terminée lorsque l'on ne reçoit pas de réponse à un paquet (**IPS_SEEN_REPLY_BIT**). **nf_ct_udp_timeout_stream** est le timeout mis en place dans un *stream* (lorsqu'il s'agit d'une réponse à un paquet envoyé au préalable).

HZ correspond à la fréquence du *timer* définie dans la variable **CONFIG_HZ** lors de la compilation du noyau. Il s'agit du nombre d'interruptions provoquées par le timer en une seconde.

Ainsi, grâce à ces deux variables, Netfilter peut contrôler le flux UDP :

```
/* Returns verdict for packet, and may modify conntracktype */
static int udp_packet(struct nf_conn *conntrack,
                      const struct sk_buff *skb,
                      unsigned int dataoff,
                      enum ip_conntrack_info ctinfo,
                      int pf,
                      unsigned int hooknum)
{
        /* If we've seen traffic both ways, this is some kind of UDP
           stream.  Extend timeout. */
        if (test_bit(IPS_SEEN_REPLY_BIT, &conntrack->status)) {
                nf_ct_refresh_acct(conntrack, ctinfo, skb,
                                   nf_ct_udp_timeout_stream);
                /* Also, more likely to be important, and not a probe */
                if (!test_and_set_bit(IPS_ASSURED_BIT, &conntrack->status))
                        nf_conntrack_event_cache(IPCT_STATUS, skb);
        } else
                nf_ct_refresh_acct(conntrack, ctinfo, skb, nf_ct_udp_timeout);

        return NF_ACCEPT;
}
```

5.3 Suivi de connexion (connection tracking)

Le suivi de connexion, c'est tout simplement ce que vous venez de voir avec les états. Comme son nom l'indique, il fait un suivi de connexion (les paquets sont défragmentés). Il peut ainsi être utilisé pour mettre en relation plusieurs connexions. Une relation peut être juste comme ce que nous avons vu plus haut sur les états TCP ou UDP (**ESTABLISHED**), mais il peut aller jusque dans la lecture des protocoles plus bas, afin de savoir quelles sont les nouvelles connexions à attendre (**RELATED**). C'est le cas de SIP où les paquets de signalisation dans le VoIP donneront les ports à utiliser pour que le flux média (RTP) passe.

L'analyse du protocole se code dans un type de module qui s'appelle « *conntrack helper* ». Écrire un conntrack helper consiste juste à expliquer au système de suivi de connexion comment estimer qu'une connexion est liée à une autre, simplement en inspectant les paquets qui la composent. Il existe tout un tas de protocoles déjà pris en charge par des modules existants. Dans le cas échéant, sachez qu'il n'est pas (ou ne devrait pas être) difficile d'écrire un tel module. Parmi les modules actuels, on trouve la gestion des protocoles amanda, ftp, h323, irc, netbios, pptp, dccp, gre, sctp, sane, sip ou encore tftp...

5.4 Exemple de suivi de connexion sur FTP

Il faut que le module **ip_conntrack_ftp** soit chargé. Un des problèmes classiques rencontré avec FTP est l'omission du canal de données. FTP utilise le port 21 et FTP-data le port 20 pour communiquer. FTP-data fonctionne dans le sens opposé de la connexion FTP d'origine. Le fonctionnement de FTP est très particulier, notamment à cause de ses deux modes : actif ou passif.

5.4.1 FTP actif

Dans ce mode, le client FTP envoie, en utilisant la commande **PORT**, un numéro de port sur lequel le serveur FTP pourra ouvrir une connexion. Une fois la commande analysée, le serveur se connecte sur ce port depuis le port 20 pour envoyer les données. Sans helper pour le protocole FTP, il est impossible de connaître le numéro de port qui a été passé et il est donc nécessaire d'ajouter au jeu de règles une règle générale qui permet les connexions depuis le port 20 du serveur FTP vers les ports éphémères du client.

Mais, grâce au module **ip_conntrack_ftp**, il est possible de classifier ces connexions du serveur vers le port éphémère du client avec l'état **RELATED**. Pour cela, le module étudie le flux sur le port 21 et recherche les commandes **PORT** pour récupérer le port que le client met à disposition du serveur pour la connexion FTP-data. Lorsqu'il détecte la commande **PORT**, il crée une connexion provisoire (**EXPECT**) avec les paramètres récupérés et, lorsque le serveur ouvre la connexion FTP-data, il y a correspondance et cette nouvelle connexion est classifiée **RELATED**.

L'utilitaire conntrack est capable d'afficher la création de ces connexions provisoires. Ainsi, si on lance une connexion FTP vers le serveur 213.146.233.19 et qu'on récupère en mode actif un fichier, on voit la création d'une entrée :

```
# conntrack -E expect
300 proto=6 src=213.146.233.19 dst=192.168.1.129 sport=0 dport=53473
```

On a donc bien une connexion provisoire (timeout de 300 secondes) venant du serveur FTP vers un port dynamique du client. Le port source est zéro pour indiquer que l'on ne connaît pas le port source. C'est une insuffisance du module de suivi, puisque si l'on regarde le tcpdump, le port 20 est bien utilisé par le serveur :

```
11:48:11.385027 IP 213.146.233.19.20 > 192.168.1.129.53473: S
1933400329:1933400329(0) win 5840
11:48:11.385056 IP 192.168.1.129.53473 > 213.146.233.19.20: S
2297039665:2297039665(0) ack 1933400330 win 5792
11:48:11.471918 IP 213.146.233.19.20 > 192.168.1.129.53473: . ack 1 win 5840
```

Au niveau d'iptables, le nouveau jeu de règles à rajouter est le suivant :

```
iptables -A INPUT -p tcp --sport 20 --dport 1024: -m state --state RELATED -j ACCEPT
iptables -A OUTPUT -p tcp --sport 1024: --dport 20 -m state --state RELATED -j ACCEPT
```

5.4.2 FTP passif

Dans le mode passif, c'est le serveur qui envoie la commande **PORT** au client et le client se connecte ensuite sur le port fourni. Dans ce cas, le modules **ip_conntrack_ftp** reconnaît aussi la commande **PORT** émise depuis le serveur et crée une connexion provisoire depuis le client sans port source spécifié vers le serveur sur le port source récupéré par analyse des flux :

```
# conntrack -E expect
300 proto=6 src=192.168.1.129 dst=213.146.233.19 sport=0 dport=14963
```

Cette fois encore, le port source est zéro, mais c'est cette fois nécessaire. Les règles iptables correspondantes sont donc :

```
iptables -A INPUT -p tcp --sport 1024: --dport 1024: -m state --state RELATED -j ACCEPT
iptables -A OUTPUT -p tcp --sport 1024: --dport 1024: -m state --state RELATED -j ACCEPT
```

5.4.3 Les bugs

Nul logiciel n'est parfait, mais il s'avère que, dès que l'on sort du cadre standard d'iptables, on se retrouve avec pas mal de bugs. Ils ressemblent tout simplement à :

```
root@quinificator:~# iptables -A INPUT -p tcp --syn --dport 23 -m \
connlimit --connlimit-above 2 -j REJECT
iptables: Unknown error 4294967295
```

Il n'y a rien à faire, à part le rapporter, s'il persiste avec la dernière *release* du noyau.

6 Conclusion

En conclusion, nous pouvons maintenant vous avouer que nous sommes allés un peu plus loin que la simple introduction. Par exemple, il nous a semblé important de montrer comment tester vos règles, afin que vous puissiez comprendre au mieux le fonctionnement de Netfilter. Malheureusement, nous avons dû faire quelques compromis pour laisser un peu de place aux autres articles. Ils approfondissent un bon nombre de sujets et c'est donc l'occasion d'en profiter pour voir comment bloquer des attaques réseau, faire du filtrage sur IPv6, avoir des logs efficaces et encore bien d'autres choses. Bonne lecture !

Auteurs : Éric Leblond, Sébastien Tricaud

NOUVEAUTÉS

Ulogd2, journalisation avancée

Si « un homme sans passé est plus pauvre qu'un homme sans avenir » (Elie Wiesel), c'est pire encore pour un réseau. Sans journalisation, un réseau est aveugle : les attaques restent sans preuve, les problèmes de configuration restent incompréhensibles. Si Netfilter a toujours offert des outils intéressants pour la journalisation, ulogd2 étend encore ses capacités.

Auteurs

- Éric Leblond
- Pierre Chifflier

1 Sources d'événements de Netfilter

1.1 Journalisation de paquets

La journalisation de paquets sous Netfilter s'effectue principalement au moyen de règles de filtrage dont la cible est une cible de journalisation (**LOG**, **ULOG**, **NFLOG**). Ces règles sont non terminales et peuvent utiliser tous les filtres disponibles. Par non terminale, on entend que le fait de rencontrer une règle de journalisation n'entraîne pas une décision sur le paquet (à l'inverse de **DROP** ou **ACCEPT**). Par conséquent, le schéma d'utilisation des cibles de journalisation le plus souvent rencontré est le suivant :

```
iptables -A FORWARD -p tcp --dport 23 -j LOG --log-
prefix "Telnet is bad "
iptables -A FORWARD -p tcp --dport 23 -j DROP # DROP
telnet explicite
```

L'administrateur décide de faire journaliser le paquet suivant un filtre et prend juste après la décision sur ce même paquet.

1.1.1 LOG

La cible **LOG** est la cible historique pour la journalisation des paquets dans Netfilter et même dans Ipchains, la couche pare-feu des noyaux 2.2. Il s'agit d'une journalisation noyau standard qui envoie les paquets à loguer dans syslog.

1.1.2 ULOG

Apparu avec Netfilter, **ULOG** est la première cible évoluée pour la journalisation des paquets. Les données sont envoyées via une *socket* Netlink depuis le noyau vers l'espace utilisateur. Qui dit « socket » dit « communication » et il est donc nécessaire d'avoir un logiciel à l'écoute pour récupérer les données et les journaliser. Le logiciel ulogd a été développé par Harald Welte pour remplir cette tâche. Il récupère les données du noyau et les stocke dans différents formats grâce à des *plugins* de sortie.

En dehors des possibilités offertes par ulogd, la cible **ULOG** présente certains avantages comme pouvoir envoyer les paquets de l'espace noyau par bouffée ce qui accroît les performances et économise certains médias de stockage.

1.1.3 NFLOG

Le noyau 2.6.14 a vu une refonte des infrastructures d'interaction entre l'espace utilisateur et le noyau dans Netfilter. Le sous-système nfnetlink a été introduit pour servir de fondement aux différents types de communication entre l'espace utilisateur et le noyau. Cette factorisation des mécanismes d'échanges a rendu nécessaire de développer un successeur à **ULOG** utilisant ce nouveau mode de communication. En dehors de cette vision développeur, **ULOG** souffrait d'une limitation assez grave qui était l'absence de support d'IPv6.

Cela a mené à l'introduction de la cible **NFLOG** qui reprend les fonctionnalités d'**ULOG** en utilisant la nouvelle infrastructure et ajoutant IPv6 à la liste des protocoles supportés. Si l'on détaille les options de **NFLOG** (souvent communes avec celle d'**ULOG**), on a :

--nflog-group NUM : numéro de canal utilisé pour la journalisation ;

--nflog-range NUM : nombre d'octets de données à copier ;

--nflog-threshold NUM : nombre de messages à stocker avant envoi par le noyau ;

--nflog-prefix STRING : chaîne de préfixe pour les messages de logs.

`--nflog-group` doit être choisi entre 0 et 65535. On évitera 0 qui est réservé aux messages système (voir plus bas). Cette valeur définit quel canal utiliser lors de l'envoi des messages. L'intérêt de multiplier les canaux est de pouvoir différencier les messages. Cela offre un moyen de circonvenir à un problème classique de Netfilter : il n'existe pas de moyen explicite de différencier un paquet bloqué d'un paquet accepté lorsque l'on journalise un paquet. En ayant plusieurs canaux, il est ainsi possible de définir une file pour les paquets bloqués et une file pour les paquets acceptés. L'ensemble des autres scénarios d'utilisation est laissé à la sagacité du lecteur.

`--nflog-threshold` spécifie le nombre de paquets stockés au niveau du noyau avant émission d'un message vers l'espace utilisateur. Il s'agit d'un tampon qui se remplit avec les paquets journalisés et qui est vidé lorsque le nombre de paquets maximums ou lorsque des paquets sont en attente depuis un certain temps (par défaut une seconde). L'utilisation de ce paramètre limite la charge système en limitant le nombre de communication entre l'espace noyau et l'espace utilisateur, mais ne permet pas une remontée immédiate des informations.

Si le noyau Linux est plus ancien que 2.6.29, la journalisation par `NFLOG` est dépendante du système de journalisation interne (voir section 1.2). Il est donc possible qu'un paquet journalisé par `NFLOG` soit envoyé vers l'espace utilisateur comme un paquet journalisé par `LOG` ou `ULOG`.

1.2 Journalisation de paquets système

1.2.1 Système de journalisation interne

Un certain nombre de modules Netfilter sont capables de détecter des problèmes assez complexes ou assez étranges. Par exemple, le module de suivi de connexions TCP peut détecter des paquets invalides. Les causes de l'invalidité peuvent être multiples (paquet trop court, mauvais *checksum*, combinaison de drapeaux invalide,...). L'obtention de ces informations conduit donc à la détection d'attaques ou de graves problèmes système ou réseau.

Un système a donc été mis en place pour donner la possibilité aux composants Netfilter d'envoyer des messages précis à l'espace utilisateur. Un système interne de journalisation des paquets appelé `nf_log` a été développé. Il offre à tous les modules la possibilité d'envoyer des messages au moyen d'un simple appel de fonction.

Cependant, comme on vient de le voir, les voies possibles pour la journalisation (`LOG`, `ULOG`, `NFLOG`) sont multiples et `nf_log` doit donc permettre de choisir vers quel sous-système envoyer les paquets (on parlera par la suite de *logger*). Comme `ULOG`, par exemple, est uniquement compatible IPv4, il ne peut servir d'infrastructure de journalisation pour l'ensemble des protocoles et le choix du système est donc dépendant du protocole. Une table de correspondance protocole/logger a donc été mise en place. Elle est interrogeable au moyen du fichier `/proc/net/netfilter/nf_log`. Pour faire simple, les indices utilisés pour les protocoles sont ceux définis dans les en-têtes des sources du noyau (constantes `AF_*`) et ils ne correspondent donc pas à la valeur bien connue des protocoles. Ainsi, pour IPv4, il faut regarder la valeur 2 ; pour IPv6, c'est la valeur 10. À la décharge des développeurs, on a différents types de protocoles ici, par exemple `AF_BRIDGE` (7) pour les paquets reçus en mode *bridge* et une correspondance avec les numéros de protocoles standards (IP par exemple) n'est donc pas possible.

Ce système est actuellement utilisé par une bonne partie des modules de suivi de connexions et par le module `TRACE` qui permet de suivre les décisions prises pour un paquet pour chaque *hook* de Netfilter.

1.2.2 Configuration du module de sortie

Tout bon administrateur voudra paramétrer le logger qui lui convient. La bonne nouvelle, c'est qu'il peut le faire. La mauvaise nouvelle, c'est qu'il faut soit la bonne version du noyau, soit un peu d'astuce. Commençons par le cas le pire : le noyau est plus ancien que 2.6.30 (Non, non, nous ne sommes pas devins, 2.6.30 n'est pas encore sorti à ce jour, mais nous sommes à la source des modifications qu'il induit ici). Là, la configuration est simple ou presque. Le premier module chargé gagne et récupère la fonction de journalisation pour les protocoles auxquels il s'enregistre. Enfin, cela est vrai pour tous les modules sauf pour `nfnetlink_log`. Celui-ci a une fonction d'enregistrement et de désenregistrement qui lui permet de récupérer la fonction de journalisation en cours de fonctionnement. Des logiciels comme ulogd2 peuvent ainsi se rattacher au système de journalisation interne.

Pour les noyaux plus récents, le fonctionnement est plus simple. Le répertoire `/proc/sys/net/netfilter/nf_log` contient un fichier par protocole et le contenu de chaque fichier est le logger utilisé pour ce protocole. On peut ainsi changer de logger en injectant le nom d'un logger enregistré dans le fichier correspondant à un protocole. Par exemple, pour activer `ipt_LOG` sur IPv4, on peut taper :

```
# echo "ipt_LOG">/proc/sys/net/netfilter/nf_log/2
```

Pour savoir quels sont les modules activables en tant que logger, on peut afficher le fichier `/proc/net/netfilter/nf_log` qui indique entre parenthèses les modules enregistrés :

```
# cat /proc/net/netfilter/nf_log
 0 NONE (nfnetlink_log)
 1 NONE (nfnetlink_log)
 2 ipt_ULOG (ipt_LOG,nfnetlink_log,ipt_ULOG)
 3 NONE (nfnetlink_log)
 4 NONE (nfnetlink_log)
 5 NONE (nfnetlink_log)
 6 NONE (nfnetlink_log)
 7 NONE (nfnetlink_log)
 8 NONE (nfnetlink_log)
```

```
 9 NONE (nfnetlink_log)
10 NONE (nfnetlink_log)
11 NONE (nfnetlink_log)
12 NONE (nfnetlink_log)
```

1.2.3 Modules de suivi de connexions

Les modules de suivi de connexions par protocoles utilisent généralement ce système pour journaliser les paquets invalides. Comme les informations envoyées sont particulièrement techniques, elles ne sont pas compréhensibles par le commun des mortels (il faut au moins savoir se connecter à PostgreSQL) et la notification de ces événements a donc été désactivée par défaut. Pour l'activer, sur l'ensemble des protocoles, il faut faire :

```
# echo 255>/proc/sys/net/netfilter/nf_conntrack_log_invalid
```

Pour l'activer sur un protocole donné (par exemple TCP), il suffit d'écrire le numéro du protocole (6 pour TCP par exemple) dans le fichier **nf_conntrack_log_invalid**.

1.3 Événements de connexions

Les dernières versions de Netfilter (depuis 2.6.14) proposent un système de remontée des événements du suivi de connexions vers l'espace utilisateur. En langage compréhensible, un message est envoyé à l'espace utilisateur dès qu'une connexion est créée, change d'état ou est détruite. Ce système est utilisé pour répliquer les connexions entre deux pare-feu et permet une bascule d'une machine vers l'autre sans coupure. Au niveau de la journalisation, l'intérêt premier est d'enregistrer la création, la destruction des connexions établies à travers le pare-feu en ayant, de plus, des informations de volumétrie et de durée. Le message envoyé à l'espace utilisateur reprend l'ensemble des paramètres d'une connexion :

```
tcp 6 431911 ESTABLISHED src=192.168.22.176 dst=91.21.73.151 sport=44189
dport=993 packets=1573 bytes=124587 src=91.21.73.151 dst=213.144.21.78
sport=993 dport=44189 packets=2408 bytes=2677410 [ASSURED] mark=10003 use=1
```

On remarque ainsi que l'entrée contient le détail des transformations effectuées lors de la traduction d'adresse. Ici, une connexion de 192.168.22.176 vers le port 993 (imaps) de 91.21.73.151 devient une connexion de 213.144.21.78 vers 91.21.73.151 sur le port 993. Ce détail sur les opérations de traduction d'adresse est intéressant, puisqu'il contient la trace des connexions vues de l'intérieur et de l'extérieur. Il est donc possible sur requête d'un tiers (une personne costumée par exemple) qui n'a que la vision externe du trafic de déterminer depuis quelle adresse IP interne provenait une connexion (malicieuse, malveillante, illégale, [réponse au choix]) dirigée vers un serveur externe.

La remontée des événements du suivi de connexions vers l'espace utilisateur se fait au moyen de la bibliothèque **libnetfilter_conntrack** qui est chargée de l'interaction avec le suivi de connexion.

2 Présentation d'ulogd2

2.1 Architecture

Ulogd2 est le démon chargé de journaliser les paquets envoyés en espace utilisateur par Netfilter/Iptables, via les cibles **ULOG** et **NFLOG** et les connexions par **libnetfilter_conntrack**.

Ulogd2 est un système flexible et basé sur des modules. Les modules sont utilisés, par exemple, pour enregistrer un flux de données au format PCAP ou encore pour écrire dans une base de données, mais ils peuvent aussi servir de source de données ou encore de filtre. Un des intérêts évidents est de pouvoir utiliser une base de données pour faire l'analyse et l'extraction de statistiques avec une interface graphique, comme NuLog (http://software.inl.fr/trac/wiki/EdenWall/NuLog2) Nous allons détailler leur utilisation par la suite.

Grâce à cette architecture, il est très facile de créer un nouveau module pour tout besoin spécifique, par exemple l'importation de données depuis une source externe ou l'exportation vers d'autres systèmes.

2.2 Principe des stacks

Ulogd2 utilise les modules pour former une chaîne (appelée *stack*). Il existe trois types de modules :

- source ;
- filtre ;
- sortie.

Une chaîne doit absolument avoir une source, et une sortie (elle peut avoir plusieurs plugins de filtrage). Il est possible de définir plusieurs chaînes dans la configuration.

Chaque plugin possède un **type** (par exemple, PGSQL), et doit être instancié (en utilisant un nom, choisi par l'utilisateur). De cette manière, il est possible d'utiliser un même plugin dans différentes chaînes (avec différents paramètres de configuration), et, par exemple, d'envoyer les données vers différentes destinations, en utilisant plusieurs chaînes.

Les données d'entrée et de sortie de chaque module sont appelées des *clés*. Dans une chaîne, chaque module demande une liste de clés au module précédent, et le module de sortie peut utiliser les clés du module d'entrée, et des modules de filtrage.

Par exemple, la ligne suivante :

```
stack=log2:NFLOG,base1:BASE,ifi1:IFINDEX,ip2str1:IP2STR,mac2str1:HWHDR,
pgsql1:PGSQL
```

définit une chaîne avec les paramètres suivants :

- Le module d'entrée est **NFLOG**, donc il faudra utiliser la cible **-j NFLOG** d'iptables comme source.

- Les filtres sont **IFINDEX, IP2STR, HWHDR**.
- Les données seront enregistrées dans une base de données PostgreSQL.

Les différents modules d'entrées disponibles sont :

- **ULOG** : récupère des paquets depuis la source iptables **ULOG**.
- **NFLOG** : récupère des paquets depuis la source iptables **NFLOG** (apporte notamment le support IPv6 et ebtables).
- **NFCT** : récupère des paquets depuis le suivi de connexions Netfilter, en utilisant **libnetfilter_conntrack**.

Les modules de filtrage sont :

- **BASE** : convertit les données binaires du paquet en une structure utilisable par les autres modules.
- **HWHDR** : extrait les adresses MAC des en-têtes ethernet.
- **IFINDEX** : convertit un numéro d'interface réseau en nom d'interface (par ex. eth0).
- **IP2BIN** : extrait et normalise les adresses IP, sous forme binaire.
- **IP2STR** : extrait et normalise les adresses IP, sous forme de chaîne de caractères.
- **MARK** : ne laisse passer que les paquets dont la marque correspond à celle de la configuration.
- **PRINTFLOW** : convertit une connexion en chaîne de caractères.
- **PRINTPKT** : convertit un paquet en chaîne de caractères.
- **PWSNIFF** : essaye de détecter des séquences de login avec des mots de passe en clair.

Les modules de sortie :

- **LOGEMU** : enregistre les données dans un fichier texte.
- **NACCT** : exporte les données dans un format compatible nacct (*accounting*).
- **OPRINT** : enregistre les données dans un fichier texte, dans un format multiligne.
- **PCAP** : exporte les données au format PCAP.
- **SYSLOG** : enregistre les données en les envoyant à syslog.
- **IPFIX** : permet d'exporter les données au format IPFIX (*IP Flow Information Export*) [en cours de développement].
- Base de données : ulogd2 supporte de nombreuses bases de données (MySQL, PostgreSQL, etc.). Les bases supportées sont détaillées au chapitre suivant.

Chaque module est chargé dynamiquement lors du démarrage d'ulogd2. Un module dispose de :

- variables de configuration : elles paramètrent le comportement du module ;
- clés d'entrée (*input keys*) : elles définissent les valeurs (obligatoires ou pas) utilisées par le module pour pouvoir fonctionner ;
- clés de sortie (*output keys*) : les valeurs de sorties du module.

Chaque clé dispose d'un type, et peut avoir une valeur par défaut. Pour avoir une chaîne fonctionnelle, il faut obligatoirement un module d'entrée, et un ou plusieurs modules de filtrage dont l'ensemble des clés de sortie correspondra aux clés d'entrée du dernier module, le module de sortie.

Des informations sur un module, ses possibilités de configuration et ses clés d'entrée et de sortie, peuvent être obtenues en utilisant la commande **info** d'ulogd2 :

```
# /opt/ulogd2/sbin/ulogd --info /opt/ulogd2/lib/ulogd/ulogd_filter_
IP2STR.so
Name: IP2STR
Input keys:
        Key: oob.family (unsigned int 8)
        Key: oob.protocol (unsigned int 16)
        Key: ip.saddr (IP addr, optional)
        Key: ip.daddr (IP addr, optional)
        [...]
Output keys:
        Key: ip.saddr.str (string)
        Key: ip.daddr.str (string)
        [...]
```

La logique du module **IP2STR** est simple. Il convertit les champs de type **IP addr** en des champs de type **string**. Ainsi, si la clé **ip.saddr** est présente, il la convertit en une chaîne de caractères qu'il stocke dans **ip.saddr.str**. Cette clé peut alors être utilisée par les autres plugins de la stack.

Dans le cas d'un module de sortie, il n'y a pas de clés de sorties, mais cela ne l'empêche pas d'avoir des options de configuration :

```
# /opt/ulogd2/sbin/ulogd --info /opt/ulogd2/lib/ulogd/ulogd_output_
PCAP.so
Name: PCAP
Config options:
  Var: file (String, Default: /var/log/ulogd.pcap)
  Var: sync (Integer, Default: 0)
Input keys:
  Key: raw.pkt (unsigned int 32)
  Key: raw.pktlen (unsigned int 32)
  Key: ip.totlen (unsigned int 16)
  Key: oob.time.sec (unsigned int 32)
  Key: oob.time.usec (unsigned int 32)
Output keys:
  Output plugin, No keys
```

2.3 Bases de données

2.3.1 Schémas et insertion des données

Lors de l'utilisation de bases de données pour des volumes de données conséquents, le problème de conception suivant revient régulièrement : faut-il privilégier le design (modèle en nième forme normale, nombre important de tables) au détriment de la rapidité, ou faut-il utiliser un schéma avec un nombre de tables restreint, et éventuellement de la duplication de données, pour augmenter les performances ?

Cette question (à débattre lors de longues soirées) n'a pas de solution simple, c'est pourquoi nous avons décidé, en développant ulogd2... de ne pas décider ! Ulogd2 va utiliser une couche d'abstraction pour pouvoir enregistrer les données sans connaître le schéma SQL utilisé, et de cette manière laisser le choix à l'utilisateur.

Par défaut, ulogd2 utilise un schéma dans lequel les données sont réparties entre différentes tables. Les données communes (niveau IP) sont inscrites dans une table, les données TCP dans une autre, etc. Ce schéma assure donc une bonne cohérence des données. En revanche, il présente plusieurs difficultés :

- Un paquet contient des données à insérer dans plusieurs tables, qui dépendent des critères du paquet (protocole TCP ou UDP par exemple).
- Pour récupérer les données, une application devra extraire des données de plusieurs tables (en effectuant des jointures), également en fonction des critères du paquet. Ce point ne concerne pas directement ulogd2, mais il est préférable de prévoir l'utilisation des données pour pouvoir la simplifier.

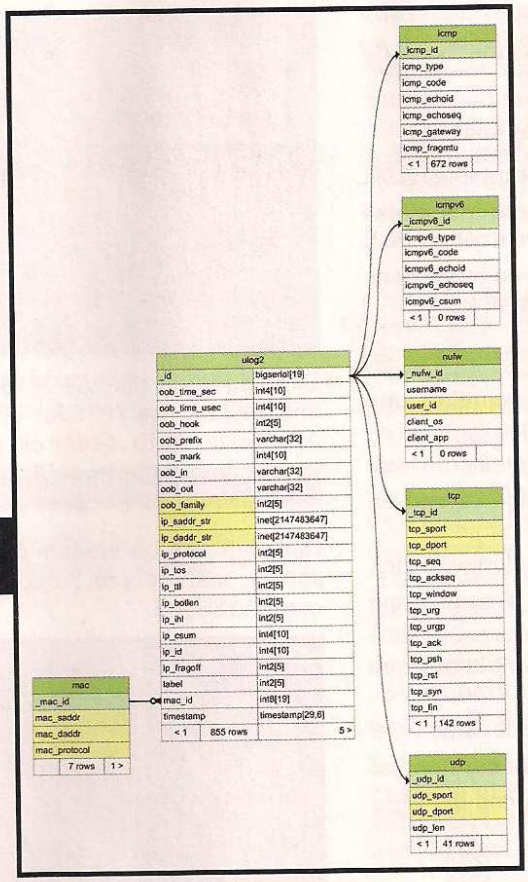


Schéma de la base de données PgSQL

La problématique de l'insertion va être résolue en utilisant des procédures stockées dans la base de données. Ulogd2 va appeler une procédure unique, qui sera chargée de faire les insertions dans les différentes tables. Voici un exemple de procédure, en utilisant le langage PL/pgSQL, propre à PostgreSQL :

```
CREATE OR REPLACE FUNCTION INSERT_PACKET_FULL(
            IN value1 integer,
            …)
RETURNS bigint AS $$
DECLARE
      t_id bigint;
DECLARE
        t_id := INSERT INTO tablename (field1,field2,…) VALUES ($1,$2,…);
        RETURN t_id;
END
$$ LANGUAGE plpgsql SECURITY INVOKER;
```

Cette procédure ne fait qu'insérer dans une table, mais elle pourrait également effectuer des opérations plus complexes.

Si l'on regarde ce qu'un module de sortie en base de données a dans le ventre, on peut être surpris (ça mange n'importe quoi un éléphant ou un dauphin...). Prenons le cas de PostgreSQL et *dumpons* les informations avec l'option **info** :

```
/opt/ulogd2/sbin/ulogd --info /opt/ulogd2/lib/ulogd/ulogd_output_PGSQL.so
Name: PGSQL
Config options:
  Var: table (String, Default: , Mandatory)
  Var: reconnect (Integer, Default: 2)
  Var: connect_timeout (Integer, Default: 0)
  Var: procedure (String, Default: , Mandatory)
  Var: db (String, Default: , Mandatory)
  Var: host (String, Default: )
  Var: user (String, Default: , Mandatory)
  Var: pass (String, Default: )
  Var: port (Integer, Default: 0)
  Var: schema (String, Default: public)
Input keys:
  No statically defined keys
Output keys:
  Output plugin, No keys
```

En ce qui concerne les options de configuration, il n'y a rien de bien exceptionnel à première vue. Ce qui est le plus bizarre concerne l'absence de clé d'entrée. Celles-ci sont en fait générées dynamiquement à partir des noms des champs de la table pointée par **table**. Les champs de la table sont convertis en remplaçant les *underscores* par des points, et ulogd cherche alors la correspondance avec les clés disponibles. S'il parvient à trouver toutes les clés, l'initialisation de la stack est réussie. L'ensemble ordonné de ces variables est alors utilisé comme liste des paramètres pour la procédure pointée par **procedure**. L'avantage de cette solution est de garantir l'adaptabilité des modules de bases de données à une large gamme d'opérations et une évolution des procédures stockées dans la base ne nécessite pas une recompilation d'ulogd, mais au plus un simple redémarrage de celui-ci si la liste des paramètres de la procédure a évolué.

2.3.2 Comparaison et utilisation des schémas

La lecture des données possède également une solution simple : il suffit d'encapsuler la lecture des données dans une vue, qui prendra en charge la collecte des données dans les différentes tables en utilisant des jointures. Par exemple :

```
CREATE OR REPLACE VIEW ulog AS
        SELECT _id,
        oob_time_sec,
        oob_time_usec,
        …
        FROM ulog2 LEFT JOIN tcp ON ulog2._id = tcp._tcp_id LEFT JOIN
udp ON ulog2._id = udp._udp_id
                LEFT JOIN sctp ON ulog2._id = sctp._sctp_id
                LEFT JOIN icmp ON ulog2._id = icmp._icmp_id
                LEFT JOIN mac ON ulog2.mac_id = mac._mac_id
                LEFT JOIN hwhdr ON ulog2._id = hwhdr._hw_id
                LEFT JOIN icmpv6 ON ulog2._id = icmpv6._icmpv6_id;
```

Ouf ! Nous avons donc réussi à transformer un schéma simple (qui utilisait des insertions) en quelque chose de plus lent et de plus complexe. Mais pourquoi ?

Avantages :

- Indépendance entre le code C et SQL : si le schéma SQL change, pas besoin de changer le code C.
- Le schéma SQL peut être spécifique à la base utilisée. Par exemple, PostgreSQL dispose de types de données spécifiques pour les adresses IP et MAC (**inet** et **macaddr**),

alors que MySQL ne dispose pas de ces types (il faut par exemple enregistrer les données en binaire).

- Le schéma SQL est facile à étendre pour de nouvelles applications : il suffit d'ajouter une table, et de la relier à la table principale en utilisant l'*id* du paquet.
- La récupération de données est plus simple pour les applications comme NuLog, puisque la complexité du schéma est masquée par les procédures et les vues.
- Cela permet d'avoir un contrôle plus strict sur les données et leur cohérence. Par exemple, les adresses MAC ne sont stockées qu'une seule fois dans ce schéma, ce qui représente un gain de place conséquent (dans le schéma monolithique, les adresses sont écrites pour chaque paquet).

Inconvénients :

- Lenteur : l'utilisation de procédures, ainsi que le fait d'avoir plusieurs tables, ralentissent fortement les insertions. Ce schéma n'est donc pas recommandé dans le cas où les performances sont importantes.
- Le schéma est plus difficile à lire.
- Certaines bases ne disposent pas de procédures stockées (sqlite3 par exemple). Ce point est gênant, mais là encore une solution a été trouvée.
- Un deuxième schéma est disponible, qui enregistre toutes les données dans une seule table (schéma *flat*). Ce schéma, orienté performance, n'a donc pas besoin d'utiliser de procédure, puisque les données peuvent être insérées directement.
- Nous avons donc étendu Ulogd2 pour qu'il accepte **INSERT** comme nom de procédure, auquel cas il insère les données directement, sans procédures.
- Grâce à cette souplesse sur le schéma, ulogd2 est donc capable de s'adapter à un grand nombre de systèmes différents, sans besoin de modifications.

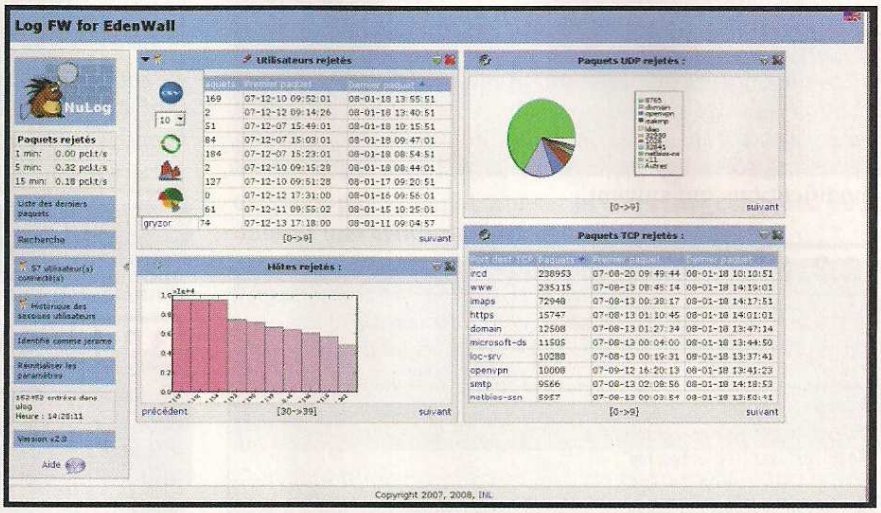
Ulogd2 supporte les bases de données suivantes :

- **MYSQL** : enregistre les données dans une base MySQL.
- **POSTGRESQL** : enregistre les données dans une base PostgreSQL.
- **SQLITE3** : enregistre les données dans une base sqlite.
- **DBI** : enregistre les données dans une base de données, en utilisant la couche d'abstraction **libdbi**. Cela permet notamment d'utiliser d'autres bases de données, comme Firebird, Sybase, MS-SQL ou encore Oracle et Ingres.

Note

Le module **dbi** aurait pu être utilisé pour MySQL et PostgreSQL. Cependant, cela empêcherait l'utilisation des fonctions spécifiques de l'API de chaque base, par exemple l'API asynchrone de PostgreSQL.

Certaines applications supportent déjà ulogd2, comme notre analyseur de logs NuLog :



3 Installation et configuration

Après cette longue présentation, vous êtes impatient de pouvoir utiliser ulogd2. Comme il s'agit d'un projet encore en développement, il n'existe pas encore de paquets pour les distributions... nous sommes donc obligés de passer par la case compilation.

3.1 Dépendances

Pour pouvoir compiler ulogd2, nous avons besoin de :

- Un compilateur (oui, elle était facile), nous prendrons donc gcc ;
- **libnfnetlink** (>= 0.39) ;
- **libnetfilter_log** (>= 0.0.15) ;
- **libnetfilter_conntrack** (>= 0.95) ;
- **mysql-dev** ;
- **postgresql-dev** ;
- **libdbi-dev** (optionnel) ;
- **libpcap-dev** (optionnel).

Après une série d'**apt-get install**, nous voilà donc prêts pour récupérer les sources d'ulogd2. Il n'existe pas encore de versions publiées d'ulogd2. Nous allons donc extraire les sources du dépôt *git* (que les âmes sensibles se rassurent, pas besoin d'apprendre la documentation de git).

```
git clone git://git.netfilter.org/ulogd2.git/
cd ulogd2
```

3.2 Compilation

Ulogd2 utilise les *autotools*. La compilation se fera donc de manière très classique.

```
./autogen.sh
./configure --prefix=/path/to/prefix
make
[sudo] make install
```

3.3 Configuration

La configuration d'ulogd2 est contenue dans le fichier **ulogd.conf**, qui est installé dans **$PREFIX/etc/**. La configuration est au format INI, et est séparée en différentes sections :

- la section globale, contenant la liste des modules à charger, la définition des chaînes, la configuration spécifique ulogd2, les options mémoire, etc.
- une section pour chaque instance de module, utilisée pour définir les paramètres propres à chaque instance.

La première étape est de choisir les modules d'entrée et de sortie, en fonction de votre système. **NFLOG** est présent dans tous les noyaux récents, et doit être préféré à **ULOG** chaque fois que c'est possible. Dans notre exemple, nous allons choisir les modules **NFLOG** et **POSTGRESQL**. La configuration doit être adaptée pour apporter les modifications qui suivent.

3.3.1 Section globale

Cette section permet de régler le comportement d'ulogd2, par exemple pour la gestion des ressources comme la mémoire.

```
[global]
# logfile for status messages
logfile="/var/log/ulogd.log"

# loglevel: debug(1), info(3), notice(5), error(7) or fatal(8)
loglevel=1

# socket receive buffer size (should be at least the size of the
# in-kernel buffer (ipt_ULOG.o 'nlbufsiz' parameter)
rmem=131071

# libipulog/ulogd receive buffer size, should be > rmem
bufsize=150000
```

3.3.2 Chargement des modules

Tous les modules utilisés par la suite doivent être chargés dans cette partie. La configuration des modules se fait dans des sections spécifiques.

```
plugin="/opt/ulogd/ulogd_inppkt_NFLOG.so"
plugin="/opt/ulogd/ulogd_output_PGSQL.so"
plugin="/opt/ulogd/ulogd_filter_IFINDEX.so"
plugin="/opt/ulogd/ulogd_filter_IP2STR.so"
plugin="/opt/ulogd/ulogd_filter_HWHDR.so"
```

3.3.3 Définition des chaînes

L'étape suivante est la définition des chaînes de modules. Pour fonctionner, ulogd2 doit disposer d'au moins une chaîne. La configuration pouvant parfois être assez astucieuse, le moyen le plus simple est de partir des chaînes données en exemple, en en décommentant une, et en l'adaptant au besoin.

Chaque module est utilisé sous la forme **instancename:MODULE** où **instancename** est défini par l'utilisateur, et **MODULE** est le nom d'un module chargé précédemment.

```
# IPv4 logging via PostgreSQL
stack=log1:NFLOG,base1:BASE,ifi1:IFINDEX,ip2str1:IP2STR,mac2str1:HWHDR,
pgsql1:PGSQL

# IPv4 logging via MySQL
#stack=log1:NFLOG,base1:BASE,ifi1:IFINDEX,ip2bin1:IP2BIN,mac2str1:HWHDR
,mysql1:MYSQL

# this is a stack for IPv4 packet-based logging via LOGEMU
#stack=log1:NFLOG,base1:BASE,ifi1:IFINDEX,ip2str1:IP2STR,print1:PRINTPK
T,emu1:LOGEMU

# this is a stack for IPv4 packet-based logging via LOGEMU with
filtering on MARK
#stack=log1:NFLOG,mark1:MARK,base1:BASE,ifi1:IFINDEX,ip2str1:IP2STR,pri
nt1:PRINTPKT,emu1:LOGEMU

# this is a stack for flow-based logging via LOGEMU
#stack=ct1:NFCT,ip2str1:IP2STR,print1:PRINTFLOW,emu1:LOGEMU

# this is a stack for flow-based logging via OPRINT
#stack=ct1:NFCT,op1:OPRINT
```

Il est possible de définir différentes chaînes, en ajoutant autant de directives **stack=** que nécessaire. Chaque module utilisant une chaîne devra donc faire usage d'une instance différente.

3.3.4 Configuration des modules

Chaque instance définie précédemment doit être configurée. Si parfois aucune configuration n'est nécessaire, la définition d'une section vide pour l'instance est obligatoirement une initialisation correcte du module.

```
# IPv4 logging through NFLOG
[log1]
# netlink multicast group (the same as the iptables --nflog-group
param)
group=1

# IPv6 logging through NFLOG
[log2]
group=2 # Group has to be different from the one used in log1

[emu1]
file="/var/log/ulogd_syslogemu.log"
sync=1

[sys2]
facility=LOG_LOCAL2

[pgsql1]
db="ulog2"
host="localhost"
user="ulog2"
table="ulog"
pass="ulog2_pass"
procedure="INSERT_PACKET_FULL"
```

3.4 Configuration de Postgresql

Il reste à créer la base de données, et l'utilisateur :

```
# su - postgres
$ createuser -P ulog2
$ createdb -O ulog2 ulog2
```

Pour pouvoir utiliser les procédures, le support du langage PL/pgSQL doit être ajouté à la base :

```
$ createlang plpgsql ulog2
```

Il ne reste plus qu'à insérer la structure initiale de la base (cette action ne se fait plus en tant qu'utilisateur **postgres**). Pour les non-habitués à PostgreSQL, on rappelle qu'il peut être nécessaire de modifier le fichier **pg_hba.conf** pour autoriser le mécanisme md5 en local.

```
$ psql -U ulog2 -h localhost ulog2 -f doc/pgsql-ulogd2.sql
```

Si tout se passe bien, vous n'avez plus qu'à démarrer ulogd2.

3.5 Démarrage d'ulogd2

Pour le moment, aucune donnée ne sera envoyée à ulogd2. Il reste à ajouter des règles iptables pour envoyer des données à **NFLOG**, et à démarrer ulogd2. Voici un exemple de règle iptables :

```
# iptables -A FORWARD -p tcp --dport 80 -m state --state NEW -j NFLOG \
--nflog-group 1
```

On peut vérifier à l'aide des compteurs que des paquets sont bien envoyés à **NFLOG** :

```
# iptables -vnL FORWARD
Chain FORWARD (policy ACCEPT 0 packets, 0 bytes)
 pkts bytes target     prot opt in     out     source               destination
 44582 2718K NFLOG      tcp  --  *      *       0.0.0.0/0            0.0.0.0/0
tcp dpt:80 flags:0x17/0x02 state NEW nflog-group 1
```

Nous pouvons donc démarrer ulogd2. Avant de le démarrer en mode définitif, nous allons commencer par l'utiliser en avant-plan et vérifier que tout fonctionne :

```
# /opt/sbin/ulogd
```

Et là, c'est le drame. Ulogd2 rend la main, en n'affichant que le message :

```
Fatal error, check logfile "/var/log/ulogd.log".
```

Comme nous n'avons rien d'autre à faire, nous écoutons sagement ulogd2, et regardons le fichier **/var/log/ulogd.log**.

Attention

Attention, surtout si vous utilisez la commande **tail**, il faut *scroller* plusieurs fois avant d'arriver aux messages intéressants...

```
Wed Mar 26 22:19:54 2008  ulogd.c:698 cannot find key `timestamp' in stack
Wed Mar 26 22:19:54 2008  ulogd.c:807 destroying stack
```

Après quelques tentatives (ici, l'erreur venait du module **BASE** qui n'était pas utilisé dans la chaîne), ulogd se lance correctement. En effectuant quelques connexions qui vérifient la règle iptables, on obtient quelques paquets dans la base SQL :

```
ulog2=> select count(*) from ulog2;
 count
-------
     3
(1 row)
```

On peut, par exemple, utiliser la vue **view_tcp_quad** pour obtenir une synthèse des connexions.

```
ulog2=> select * from view_tcp_quad;
 _id | ip_saddr_str | tcp_sport | ip_daddr_str  | tcp_dport
-----+--------------+-----------+---------------+-----------
   1 | 192.168.1.20 |     56035 | 91.121.73.151 |        80
   2 | 192.168.1.20 |     56037 | 91.121.73.151 |        80
   3 | 192.168.1.20 |     56038 | 91.121.73.151 |        80
(3 rows)
```

Tout fonctionne correctement, nous pouvons donc lancer ulogd2 en mode *daemon* :

```
# /opt/sbin/ulogd -d
```

4 Utilisation

4.1 Utilisation des labels

Nous avions évoqué au début de l'article la difficulté de différencier l'état (bloqué ou accepté) d'un paquet lors de la journalisation. Une solution partielle reposait sur l'utilisation de files de journalisation séparées grâce à l'option **--nflog-group**. Ulogd2 propose une option **numeric_label** dont la définition et l'usage sont laissés à l'administrateur. Une utilisation pratique est de définir une convention sur la décision prise sur le paquet, par exemple 0 pour bloqué, 1 pour accepté. Prenons comme cas d'exemple la journalisation des paquets acceptés et droppés par **NFLOG** dans une base PostgreSQL (base configurée comme décrit précédemment).

Commençons par définir deux stacks, la première pour les paquets droppés, la deuxième pour les paquets acceptés :

```
stack=logdrop:NFLOG,base1:BASE,ifi1:IFINDEX,ip2str1:IP2STR,mac2str1:HWH
DR,pgsql1:PGSQL
stack=logaccept:NFLOG,base1:BASE,ifi1:IFINDEX,ip2str1:IP2STR,mac2str1:H
WHDR,pgsql1:PGSQL
```

On a ainsi deux stacks qui diffèrent uniquement par leurs instances d'entrée dont voici la configuration :

```
[logdrop]
group=1
numeric_label=0

[logaccept]
group=1
numeric_label=1
```

Avec cette configuration, les paquets journalisés dans le groupe 1 doivent être les paquets droppés, et ceux journalisés dans le groupe 2 les paquets acceptés. On peut ainsi avoir le jeu de règles suivant.

```
iptables -A FORWARD -p tcp --dport 23 -j NFLOG --nflog-group 1 --nflog- \
prefix "Telnet packet"
iptables -A FORWARD -p tcp --dport 23 -j DROP
iptables -A FORWARD -p tcp --dport 22 -j NFLOG --nflog-group 2 --nflog- \
prefix "SSH packet"
iptables -A FORWARD -p tcp --dport 22 -j ACCEPT
```

Au niveau de la base de données, les paquets acceptés et droppés seront différenciés par le champ **label** qui vaudra 1 pour les paquets acceptés et 0 pour les paquets droppés.

4.2 Journalisation système

Comme vu au début de l'article, un certain nombre de modules utilisent une fonction de journalisation commune pour envoyer des paquets à l'espace utilisateur. Lorsque le module **NFLOG** est utilisé, la journalisation s'effectue sur le groupe 0. Pour simplifier les choses, lorsque ulogd2 détecte l'utilisation d'une instance d'entrée **NFLOG** paramétrée sur le groupe 0, il attribue les événements système à **NFLOG** pour l'ensemble des protocoles supportés.

La journalisation des paquets système dans un fichier par **LOGEMU** se fait donc simplement :

```
stack=log1:NFLOG,base1:BASE,ifi1:IFINDEX,ip2str1:IP2STR,print1:PRINTPKT
,emu1:LOGEMU

[log1]
group=0

[emu1]
file="/var/log/ulogd_syslogemu.log"
sync=1
```

On définit une stack prenant en entrée une instance **NFLOG** et débouchant sur une instance de **LOGEMU**. La configuration de **log1** est simple, puisque l'on se contente de la rattacher au groupe 0. Pour **emu1**, on spécifie le fichier de journalisation avec **file** et on force une journalisation synchrone avec l'option **sync**.

4.3 Journalisation des connexions dans PostgreSQL

On se propose de journaliser l'ensemble des connexions marquées 1 dans une base PostgreSQL. Nous allons donc devoir utiliser le module d'entrée **NFCT** et le module de sortie **PGSQL**. Pour filtrer sur la marque, nous allons utiliser le module **mark** qui interrompt le traitement des paquets par la stack si la marque du paquet ou de la connexion ne correspond pas à l'option spécifiée.

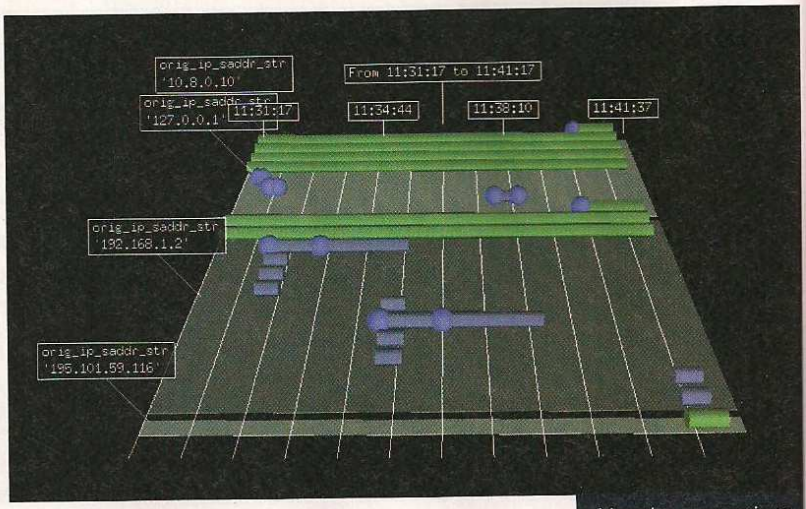
```
stack=ct1:NFCT,mark1:MARK,ip2str1:IP2STR,pgsql2:PGSQL
[ct1]

[pgsql2]
db="nulog"
host="localhost"
user="nupik"
table="ulog2_ct"
pass="changeme"
procedure="INSERT_CT"

[mark1]
mark = 1
```

Il y a peu d'originalité dans cette configuration. On notera seulement l'appel à la procédure **"INSERT_CT"** en lieu et place d'**"INSERT_PACKET_FULL"**. La modularité des modules de gestion de base rend en effet possible le passage d'une journalisation de paquets à une journalisation de connexions en changeant simplement les paramètres **procedure** et **table**.

4.4 nf3d, les journaux en 3D

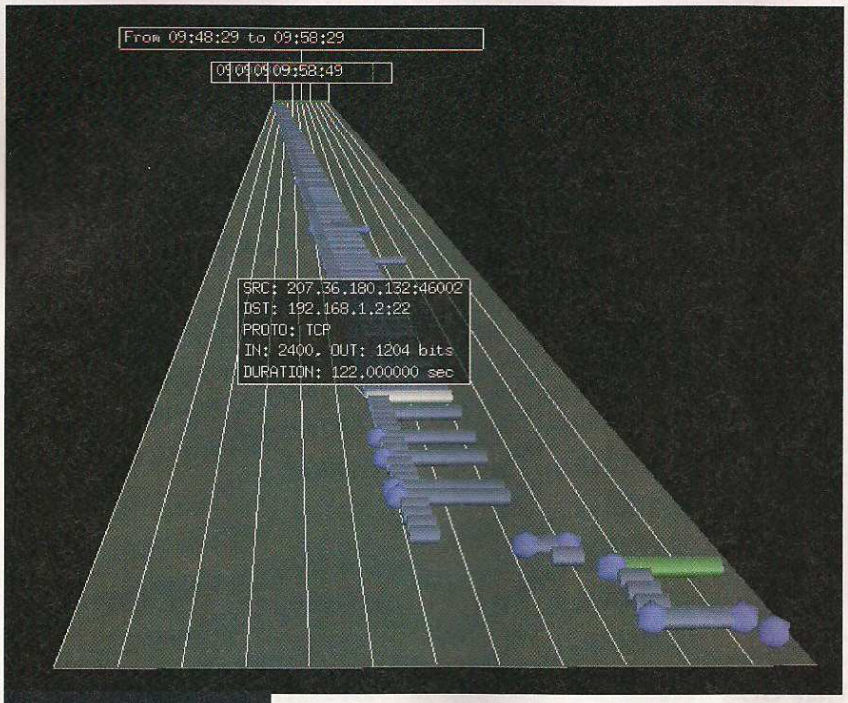
Ulogd2 journalise une quantité de données conséquente et, pour la première fois, les données sont de deux types. Le type connexion est le plus nouveau et sa représentation n'est pas triviale. Si les paquets sont souvent représentés par des points d'un espace comportant plus ou moins de dimensions, l'évidence d'une représentation similaire n'est pas totale pour les connexions. La notion de date de début et de fin sont dans le même composant et il est tentant de visualiser ces deux points sur un même axe. La représentation naturelle d'une connexion est donc un segment.

Une connexion présente des similitudes avec une notion plus courante, la tâche dans un projet. Dans les deux cas, on a un début et une fin et un ensemble d'autres paramètres. On a aussi une notion de dépendances avec, par exemple, une connexion TCP qui dépend d'une connexion DNS pour résoudre le serveur cible de la connexion TCP. Il nous a donc paru intéressant de tenter de représenter les connexions sous forme de diagramme de Gantt.



Vue des connexions

Les connexions sont représentées les unes en dessous des autres, la connexion ayant commencé en dernier est affichée au plus près et la plus ancienne dans le fond. Cette visualisation donne de bons résultats comme le montre la capture d'écran montrant une attaque brute force SSH.



Attaque brute force SSH

nf3d est construit au-dessus de python-visual et va chercher ses données dans la base PostgreSQL. Le *toolkit* python-visual est un formidable outil de *maquettage* d'interface 3D. Quelques lignes de codes suffisent à créer des objets simples et à définir les interactions avec la souris et le clavier.

nf3d réalise un lien entre une connexion journalisée et les paquets journalisés qui appartiennent à cette connexion. Il utilise pour cela une requête sur la base SQL en réalisant une jointure sur l'égalité des paramètres IPv4. Le principe est exprimé dans la requête suivante :

```
SELECT oob_time_sec+oob_time_usec/1000000 AS time, *
FROM ulog2
  JOIN tcp on _id=_tcp_id
  JOIN ulog2_ct ON ip_saddr_str=orig_ip_saddr_str AND
ip_daddr_str=orig_ip_daddr_str
      AND ip_protocol=orig_ip_protocol AND tcp_
sport=orig_l4_sport AND tcp_dport=orig_l4_dport
  WHERE tcp.tcp_dport = 22 AND ulogd.oob_time_sec >
$connexion_start
    AND ulogd_ct.oob_time_sec < $connexion_end
```

L'idée de cette requête est de constituer une jointure TCP en joignant la table **ulog2** et la table **tcp**, puis de joindre les données obtenues avec la table des connexions **ulog2_ct** en utilisant une égalité des paramètres IP et TCP. En s'assurant ensuite que les paquets sont arrivés dans l'intervalle de vie de la connexion, on trouve alors l'ensemble des paquets et leur connexion d'appartenance.

La puissance des bases de données est donc mise à profit pour extraire des informations qualifiées et cohérentes sur les paquets et les connexions. La modularité du schéma composite et la journalisation des paquets et des connexions confère une grande souplesse aux requêtes possibles.

5 Conclusion

Ulogd2 est actuellement en phase de bêta. Il bénéficie d'une conception solide et fiable et est donc d'une stabilité suffisante pour la plupart des installations. La configuration ainsi que les options de configuration ne devraient plus beaucoup changer lors des développements à venir.

Les évolutions principales concernent la finalisation du support IPFIX et l'incorporation d'un module d'entrée « Unix socket » qui permettra d'injecter des données depuis un programme tournant sur la même machine. Une des modifications à venir est le support des filtres d'événements en espace noyau proposé par les dernières versions de **libnetfilter_conntrack**. Cette évolution permettra de réduire le nombre de messages à traiter en espace utilisateur et augmentera les performances globales.

Ulogd2 est donc une réécriture complète d'ulogd qui apporte de nouvelles fonctionnalités intéressantes tant par le support de IPv6 que par celui de la journalisation des connexions. La version 1.0 du projet devrait être publiée au courant de l'année et elle permettra aux utilisateurs d'envisager sereinement un remplacement de leur ulogd vieillissant par cette nouvelle version.

Références

- ulogd : http://www.netfilter.org/projects/ulogd/index.html
- Nulog : http://software.inl.fr/trac/wiki/EdenWall/NuLog
- nf3d : http://software.inl.fr/trac/wiki/nf3d

Auteurs : Éric Leblond, Pierre Chifflier

Éric Leblond
Utilisateur GNU/Linux depuis 1996. Co-fondateur et directeur technique d'INL. Développeur principal de NuFW. Contributeur sur ulogd2 (code)

Pierre Chifflier
Utilisateur GNU/Linux depuis 1997. Responsable de l'équipe technique d'INL. Développeur Debian, contributeur NuFW. Contributeur sur ulogd2 (schémas et design SQL).

Netfilter et le filtrage du protocole

Bien que la création du protocole IPv6 date de 1998 et que l'exploitation commerciale ait débuté en 2001 (le FAI japonais NTT propose IPv6), le protocole est encore en évolution. Nous vous proposons ici un tour d'horizon du protocole et de son filtrage par Netfilter.

Auteurs

- Victor Stinner
- Eric Leblond

1 Netfilter et le filtrage du protocole IPv6

Si la pénurie d'adresses IPv4 est la première motivation de la définition d'IPv6, ce protocole intègre aussi de nouvelles fonctionnalités comme la découverte d'adresses, la fragmentation à la source, la généralisation du *multicast*, etc.

Cet article fait le point sur quelques-unes de ces nouveautés et explique les conséquences en termes de filtrage par Netfilter et iptables.

2 Petit rappel sur IPv6 et début de filtrage

2.1 Format des adresses IPv6

Le changement le plus visible est la largeur des adresses : 128 bits contre 32 bits pour IPv4. Pour une lecture plus facile, une adresse est notée avec 8 blocs de 16 bits (ex : `1:2:3:4:5:6:7:8`). On peut utiliser `::` pour remplacer une suite de zéros (une seule fois par adresse), exemple : `fe80:0:0:0:0:0:0:1` est abrégée `fe80::1`.

Comme IPv4, le début de l'adresse indique son type. Les préfixes les plus courants sont :

- `2000::/3` : Adresse globale (routable sur Internet), remplace de nombreux réseaux dispersés dans l'espace d'adressage IPv4.
- `fe80::/64` : Adresse valable uniquement sur le lien local, remplace le réseau `169.254.0.0/16` (autoconfiguration IPv4 **[RFC 3927]**)
- `ff02::/16` : Multicast local, remplace `224.0.0.0/24` **[RFC 3171]**.
- `::ffff:0:0/96` : Adresse IPv4 stockée dans une IPv6, exemple : `192.168.0.2` devient `::ffff:c0a8:0002`. On peut noter l'adresse IPv4 dans sa forme originale pour une meilleure lisibilité `::ffff:192.168.0.2`.

Le réseau `::1/128` qui ne désigne qu'une seule adresse désigne la boucle locale. Elle remplace le réseau `127.0.0.0/8`.

2.2 Création des adresses locales

Chaque interface réseau a sa propre adresse de lien dans le réseau `fe80::/64`. L'adresse est générée à partir de son adresse MAC selon l'algorithme décrit dans la `[RFC 2464]` (format EUI-64). Exemple : pour l'adresse MAC `00:16:76:ab:79:ed`, l'adresse IPv6 `fe80::216:76ff:feab:79ed` sera utilisée.

La génération d'une adresse fixe (à partir de l'adresse MAC) peut être perçue comme une atteinte à la vie privée. Sous Linux, il est possible d'utiliser une adresse temporaire (adresse aléatoire, **[RFC 4941]**) en écrivant 1 dans la clé sysctl `net.ipv6.conf.default.use_tempaddr` (adresse *temporaire*). C'est le choix par défaut sous Windows.

Note

À la création du protocole IPv6, il existait trois portées d'adresse : locale, site local et globale. Site local est déprécié depuis 2004 **[RFC 3879]**. Sa définition floue peut être source de problèmes dans la politique de sécurité.

2.3 Filtrage IPv6 sous Linux

Le filtrage d'IPv6 sous Netfilter n'est arrivé que récemment à maturité. C'est, en effet, dans le

noyau 2.6.15 (sorti le 03 janvier 2006) qu'est arrivé le suivi de connexion IPv6. Jusque-là, le filtrage IPv6 s'effectuait comme un filtre sans état.

Comme le laisse comprendre le paragraphe précédent, le filtrage IPv6 s'est construit après l'élaboration du filtrage IPv4. Une des conséquences est l'utilisation d'un programme dédié pour la gestion des règles Netfilter : **ip6tables**. Cet outil se comporte comme **iptables** dont il reprend quasi intégralement la syntaxe et l'organisation. Nous allons utiliser, dans cet article, **ip6tables** pour construire un jeu de règles strict. Pour commencer, fermons les portes :

```
ip6tables -P INPUT DROP
ip6tables -P OUTPUT DROP
ip6tables -P FORWARD DROP
```

En définissant la politique par défaut à DROP sur les trois chaînes, plus rien ne peut passer en IPv6. Nous allons maintenant ajouter quelques règles pour être un peu plus à l'aise. En premier lieu, avant d'autoriser explicitement l'ouverture de certains trafics nous allons accepter les flux établis et relatifs à une connexion en cours et autoriser le trafic local :

```
ip6tables -A INPUT -m state --state ESTABLISHED,RELATED -j ACCEPT
ip6tables -A INPUT -i lo -j ACCEPT # trafic local
ip6tables -A OUTPUT -m state --state ESTABLISHED,RELATED -j ACCEPT
ip6tables -A OUTPUT -o lo -j ACCEPT # trafic local
ip6tables -A FORWARD -m state --state ESTABLISHED,RELATED -j ACCEPT
```

Comme énoncé plus haut, un noyau supérieur à 2.6.15 est nécessaire pour que les filtres sur le suivi d'état (**--state**) soient utilisables. Pour être tout à fait strict, il est nécessaire d'ajouter maintenant une règle de filtrage sur les paquets détectés comme **INVALID** par le suivi de connexions pour chaque chaîne :

```
ip6tables -A INPUT -m state --state INVALID -j DROP
ip6tables -A OUTPUT -m state --state INVALID -j DROP
ip6tables -A FORWARD -m state --state INVALID -j DROP
```

3 Autoconfiguration sous IPv6

Le protocole DHCP est la solution qui s'est imposée sous IPv4 pour assurer l'attribution automatique d'adresses aux postes se connectant à un réseau. Consciente de l'intérêt de cette fonctionnalité, l'IETF a décidé de normaliser un protocole d'autoconfiguration assurant le même service pour IPv6.

3.1 Découverte du préfixe

Pour obtenir une adresse routable sur Internet, on envoie un message **Router Solicitation** (ICMPv6 type 133, **[RFC 2461]**) à destination de **ff02::2** (routeurs) en utilisant notre adresse locale comme source (**fe80::...**). Chaque routeur va répondre avec un message **Router Advertisement** (type 134) en utilisant son adresse locale comme source et **ff02::1** comme destination. Ce message contient un préfixe et le MTU.

Le préfixe le plus court est choisi parmi les réponses et on rajoute notre suffixe (généré à partir de l'adresse MAC ou un suffixe aléatoire). Exemple : à partir du préfixe **2a01:e35:8a04:25b0::/64** et de l'adresse MAC **00:16:76:ab:79:ed**, l'adresse **2a01:e35:8a04:25b0:216:76ff:feab:79ed** est générée. Dans notre cas, le suffixe (64 bits de poids faible) est identique à celui de l'adresse locale.

On peut désactiver l'autoconfiguration (envoi de **Router Solicitation**) en écrivant 0 dans la clé sysctl **net.ipv6.conf.eth0.autoconf**, et refuser les messages **Router Advertisement** en écrivant 0 dans **net.ipv6.conf.eth0.accept_ra**. Un routeur n'utilise pas l'autoconfiguration (n'envoie pas de **Router Solicitation** et refuse les **Router Advertisement**).

Les règles pour un poste client peuvent s'écrire :

```
# Router Solicitation / Advertisement
ip6tables -A OUTPUT -d ff02::/16 -p icmpv6 --icmpv6-type 133/0 -j ACCEPT
ip6tables -A INPUT  -s fe80::/64 -p icmpv6 --icmpv6-type 134/0 -j ACCEPT
```

Les règles pour un routeur :

```
# Router Solicitation / Advertisement
ip6tables -I INPUT -p icmpv6 --icmpv6-type 133/0 -j ACCEPT
ip6tables -A OUTPUT -p icmpv6 --icmpv6-type 134/0 -j ACCEPT
```

Le filtrage des paquets **invalides** (**-m state --state INVALID**) ne peut pas être utilisé en IPv6, car les paquets ICMPv6 vers/depuis le multicast local sont détectés comme invalides. Ce problème a été rapporté et fixé par vos serviteurs et le correctif a été accepté et devrait être inclus à partir de Linux 2.6.29. Il est donc nécessaire pour tous les noyaux précédents 2.6.29 d'aménager la règle de filtrage des paquets **INVALID** exprimée précédemment :

```
ip6tables -A INPUT ! -p icmpv6 -m state --state INVALID -j DROP
ip6tables -A OUTPUT ! -p icmpv6 -m state --state INVALID -j DROP
ip6tables -A FORWARD ! -p icmpv6 -m state --state INVALID -j DROP
```

Note

Ce défaut de Netfilter s'explique en grande partie par la nature sans état du mécanisme d'attribution des adresses. Ce qui peut être considéré comme un échange, envoi d'un **Router Solicitation** pour recevoir un **Router Advertisement** n'est pas en réalité considérable comme tel, tant d'un point de vue Netfilter que d'un point de vue protocolaire. Le **Router Advertisement** peut être émis indépendamment d'un message **Router Solicitation** et le routeur n'est pas obligé de répondre immédiatement à un message **Router Solicitation**. D'un point de vue Netfilter, le changement d'adresses source et destination entre le message initial et la réponse rend le protocole impossible à suivre par le suivi de connexions qui a besoin de connaître précisément le message.

3.2 Vérification de l'unicité d'une adresse

Pour éviter les doublons, on commence par envoyer un message **Multicast Listener Report Message v2** (ICMPv6 type 143, **[RFC 3810]**) à l'adresse `ff02::16` (groupe des routeurs MLDv2) avec comme la source `::`. L'adresse `::` est utilisée lorsqu'on n'a pas encore d'adresse, et ne peut être utilisé qu'en tant que source. Ensuite, on envoie un message **Neighbor Solicitation** (type 135, **[RFC 2461]**) avec l'adresse dont on veut tester l'unicité comme destination, et `::` comme source.

En cas de doublon, le propriétaire actuel de l'adresse envoie un message **Neighbor Advertisement** (type 136) à destination de `ff02::1` (ensemble des nœuds locaux). Si aucun message n'est reçu après un certain délai (1 seconde par défaut sous Linux), on considère qu'il n'y a pas de doublon. Le multicast local remplace le protocole ARP utilisé pour IPv4.

Règles pour un poste client :

```
# Multicast Listener Report Message v2
ip6tables -A OUTPUT -d ff02::/16 -p icmpv6 --icmpv6-type 143/0 -j ACCEPT

# Neighbor Solicitation/Advertisement
ip6tables -A INPUT  -p icmpv6 --icmpv6-type 135/0 -j ACCEPT
ip6tables -A INPUT  -p icmpv6 --icmpv6-type 136/0 -j ACCEPT
ip6tables -A OUTPUT -p icmpv6 --icmpv6-type 135/0 -j ACCEPT
ip6tables -A OUTPUT -p icmpv6 --icmpv6-type 136/0 -j ACCEPT
```

Pour un routeur, remplacez la première règle par :

```
# Multicast Listener Report Message v2
ip6tables -A INPUT -p icmpv6 --icmpv6-type 143/0 -j ACCEPT
```

Le critère `--icmpv6-type 136/0` filtre le type (136), mais aussi le code (0).

3.3 Découverte des voisins

On peut utiliser un *ping* (**Echo Request**, ICMPv6 type 128) vers le multicast local la découverte des voisins **[RFC 2461]**. La commande ping a besoin de l'option `-I interface` si on utilise une adresse multicast (la notation `ipv6%interface` est aussi acceptée) :

```
$ ping6 -I eth0 ff02::1
PING ff02::1(ff02::1) from fe80::216:76ff:feab:79ed eth0: 56 data bytes.
64 bytes from fe80::216:76ff:feab:79ed: icmp_seq=1 ttl=64 time=0.045 ms
64 bytes from fe80::221:85ff:fe11:6da0: icmp_seq=1 ttl=64 time=0.132 ms (DUP!)
64 bytes from fe80::215:c5ff:feaa:3538: icmp_seq=1 ttl=64 time=0.215 ms (DUP!)
64 bytes from fe80::207:cbff:fe3c:edd8: icmp_seq=1 ttl=64 time=1.04 ms (DUP!)
^C
```

Quatre nœuds qui ont répondu (`Echo Reply`, ICMPv6 type 129) y compris nous-mêmes. Il y a donc 3 voisins :

```
$ ip -6 neigh show
fe80::221:85ff:fe11:6da0 dev eth0 lladdr 00:21:85:11:6d:a0 DELAY
fe80::215:c5ff:feaa:3538 dev eth0 lladdr 00:15:c5:aa:35:38 DELAY
fe80::207:cbff:fe3c:edd8 dev eth0 lladdr 00:07:cb:3c:ed:d8 router STALE
```

Lorsqu'on découvre un voisin, il est d'abord dans l'état `DELAY`, c'est-à-dire non joignable et on attend quelques secondes avant de tester la connectivité. On lui envoie ensuite un test de connectivité (**Neighbor Solicitation**) et il passe dans l'état `PROBE`. Si l'on reçoit la réponse (**Neighbor Advertisement**), il passe dans l'état `REACHABLE`, sinon il est supprimé. Après un certain délai, les voisins dans l'état `REACHABLE` sont supprimés, sauf le routeur qui reste dans l'état `STALE` (non joignable, mais on ne teste pas la connectivité).

Règles pour autoriser le ping :

```
# Echo Request
ip6tables -A INPUT  -p icmpv6 --icmpv6-type 128/0 -j ACCEPT
ip6tables -A OUTPUT -p icmpv6 --icmpv6-type 128/0 -j ACCEPT
ip6tables -A FORWARD -p icmpv6 --icmpv6-type 128/0 -j ACCEPT
```

4 Internet et routage

4.1 Routage

Pour afficher la table de routage, on peut utiliser la commande `ip -6 route show` (ou `route -6`). Exemple abrégé :

```
$ ip -6 route show
2a01:e35:8a04:25b0::/64 dev eth0 expires 86171sec mtu 1480 hoplimit 4294967295
fe80::/64 dev eth0 expires 21333438sec mtu 1480 hoplimit 4294967295
ff00::/8 dev eth0 expires 21333438sec mtu 1480 hoplimit 4294967295
default via fe80::207:cbff:fe3c:edd8 dev eth0 expires 1566sec mtu 1480 hoplimit 64
```

La passerelle est enregistrée avec son adresse locale. Les routes ont une durée de vie.

Comme en IPv4, il existe aussi le message **Redirect** (ICMPv6 type 137) pour indiquer une route plus courte, et le message **Destination Unreachable** (type 1) pour indiquer qu'un hôte n'est pas joignable. On peut refuser les redirections en écrivant 0 dans les clés sysctl `net.ipv6.conf.*.accept_redirects`.

Note

Le routage par la source existe dans IPv6 **[RFC 2460]**, mais les en-têtes de routage de type 0 ont été dépréciés fin 2007 **[RFC 5095]** suite aux problèmes liés à la sécurité introduits par ce protocole **[IPv6 Routing Header Security]**. IPv4 possède aussi cette fonctionnalité (LSRR : **Loose Source and Record Route**, **[RFC 791]**), mais elle est désactivée sur la plupart des pare-feu, également pour des raisons de sécurité.

4.2 Internet

Contrairement à IPv4 qui utilise divers préfixes dispersés dans la plage d'adresse IPv4, IPv6 utilise un seul préfixe `2000::/3` pour les adresses *unicast* globales. Ce préfixe est découpé en :

- `2000::/16` : Ancien bloc utilisé avant 2001 (ouverture du registre officiel)
- `2001::/16` : Adresses permanentes ouvertes à la réservation
- `2002::/16` : Tunnels 6to4, utilise `192.88.99.0/24` **[RFC 3068]** en IPv4. En particulier, l'adresse `2002:c058:6301::` est une adresse *anycast* correspondant à `192.88.99.1` en IPv4.

Pour que les clients puissent utiliser l'autoconfiguration, les fournisseurs d'accès à Internet doivent distribuer un préfixe de 64 bits ou moins (la taille minimale des préfixes est fixée par RFC 3769). En France, depuis mars 2008, Free distribue des préfixes `/60`, ce qui permet d'utiliser 16 réseaux privés /64 différents.

Note

Attention, avec la disparition du NAT (ce n'est plus nécessaire), l'ensemble des machines est accessible directement depuis Internet si vous n'avez pas de pare-feu en coupure (ex : Freebox en mode routeur).

4.3 Diminution de la charge des routeurs

Contrairement à une idée reçue, IPv6 allège la charge des routeurs. L'en-tête IPv6 a une taille fixe (40 octets), les options étant écrites à la suite. Comme l'en-tête ne contient plus de somme de contrôle, un routeur qui décrémente le nombre de sauts (`hop limit`) n'a pas à la recalculer.

Le MTU de chemin est communiqué par le message **Router Advertisement** et est donc connu. Sa valeur minimale est de 1280 octets **[RFC 2460]**. La fragmentation des paquets est réalisée à la source en utilisant l'en-tête `Fragment header` (numéro 44), et n'est donc plus à la charge des routeurs.

En IPv4, les blocs d'adresses ont tendance à être disjoints (vue la rareté des blocs libres), ce qui a pour conséquence de créer de grandes tables de routage. En IPv6, comme un réseau « privé » peut contenir au minimum 2^64 machines (4 milliards de fois Internet IPv4), on peut se contenter d'un seul bloc. Il y a moins de préfixes et les tables de routage sont donc plus petites.

Enfin, la translation d'adresse et de port (NAT) disparaît également.

5 Utilisation courante du filtrage IPv6

5.1 Règles de filtrage standard

Pour pouvoir faire quelque chose avec notre machine bien protégée, il va falloir autoriser quelques flux. Autorisons par exemple sur un poste client les connexions SSH en entrée depuis les adresses globales :

```
ip6tables -A INPUT -s 2000::/3 \
  -p tcp --sport 1024:65535 --dport 22 \
  --syn -m state --state NEW -j ACCEPT
```

Pour autoriser DNS, HTTP et HTTPS en sortie :

```
ip6tables -A OUTPUT -d 2000::/3 \
  -p udp --sport 1024:65535 --dport 53 \
  -m state --state NEW -j ACCEPT
ip6tables -A OUTPUT -d 2000::/3 \
  -p tcp --sport 1024:65535 --dport 53 \
  --syn -m state --state NEW -j ACCEPT
ip6tables -A OUTPUT -d 2000::/3 \
  -p tcp --sport 1024:65535 --dport 80 \
  --syn -m state --state NEW -j ACCEPT
ip6tables -A OUTPUT -d 2000::/3 \
  -p tcp --sport 1024:65535 --dport 443 \
  --syn -m state --state NEW -j ACCEPT
```

Si l'on paramètre un routeur, voici un exemple de règles complémentaires pour autoriser DNS, HTTP et HTTPS depuis le réseau `2a01:e35:8a04:25b0::/64` vers Internet :

```
ip6tables -A FORWARD -m state --state RELATED,ESTABLISHED -j ACCEPT
ip6tables -A FORWARD -s 2a01:e35:8a04:25b0::/64 -d 2000::/3 \
```

```
  -p udp --sport 1024:65535 --dport 53 \
  -m state --state NEW -j ACCEPT
ip6tables -A FORWARD -s 2a01:e35:8a04:25b0::/64 -d 2000::/3 \
  -p tcp --sport 1024:65535 --dport 53 \
  --syn -m state --state NEW -j ACCEPT
ip6tables -A FORWARD -s 2a01:e35:8a04:25b0::/64 -d 2000::/3 \
  -p tcp --sport 1024:65535 --dport 80 \
  --syn -m state --state NEW -j ACCEPT
ip6tables -A FORWARD -s 2a01:e35:8a04:25b0::/64 -d 2000::/3 \
  -p tcp --sport 1024:65535 --dport 443 \
  --syn -m state --state NEW -j ACCEPT
```

5.2 Journalisation IPv6

5.2.1 Paramétrage de ulogd2

La journalisation des paquets IPv6 est possible avec la cible LOG qui envoie un message dans syslog pour chaque paquet journalisé. Si l'on souhaite effectuer une journalisation dans une sortie autre que syslog, la seule solution au moment de la rédaction de l'article est d'utiliser la cible **NFLOG** et ulogd2 pour effectuer la journalisation. La cible **ULOG** qui était utilisée pour la journalisation dans ulogd est uniquement compatible IPv4 et devient donc désuète.

Le démon de journalisation ulogd2 supporte nativement IPv6. Aucun paramétrage spécifique n'est nécessaire pour traiter les paquets IPv6. À titre d'exemple, nous allons journaliser les paquets droppés en fin de règles dans un fichier et dans PostgreSQL. Pour ce qui est des règles ip6tables, c'est très simple :

```
ip6tables -A INPUT -j NFLOG --nflog-prefix "INPUT DROP" --nflog-group 1
ip6tables -A OUTPUT -j NFLOG --nflog-prefix "OUTPUT DROP" --nflog-group 1
ip6tables -A FORWARD -j NFLOG --nflog-prefix "FWD DROP" --nflog-group 1
```

Après avoir compilé et installé ulogd2, comme décrit dans l'article **[ulogd2]**. Nous éditons le fichier de configuration **ulogd.conf**. Il faut définir deux *stacks* prenant en entrée une instance **NFLOG** avec l'une sortant vers LOGEMU et l'autre vers PGSQL. Il faut donc avoir parmi les lignes décommentées du fichier :

```
stack=log2:NFLOG,base1:BASE,ifi1:IFINDEX,ip2str1:IP2STR,print1:PRINTPKT,emu1:LOGEMU
stack=log2:NFLOG,base1:BASE,ifi1:IFINDEX,ip2str1:IP2STR,mac2str1:MAC2STR,pgsql1:PGSQL

[log2]
# netlink multicast group (the same as the iptables --nflog-group param)
group=1
bind=1

[emu1]
file="/var/log/ulogd_syslogemu.log"
sync=1

[pgsql1]
db="nulog"
host="localhost"
user="nupik"
table="ulog"
pass="changeme"
procedure="INSERT_PACKET_FULL"
```

La journalisation dans l'émulation syslog ressemblera à ceci :

```
Jan 13 01:07:02 ice-age  IN= OUT=br0 MAC= SRC=2a01:e35:1394:5bda:222
:15ff:fe45:52bd DST=2001:200:0:8002:203:47ff:fea5:3085 LEN=104 TC=0
HOPLIMIT=64 FLOWLBL=1610612736 PROTO=ICMPv6 TYPE=128 CODE=0 ID=21800
SEQ=4 UID=0 GID=1001 MARK=0
Jan 13 01:07:02 ice-age  IN=br0 OUT= MAC=00:22:15:45:52:bd:00:07:cb:9
0:f8:d1:86:dd SRC=2001:200:0:8002:203:47ff:fea5:3085 DST=2a01:e35:139
4:5bda:222:15ff:fe45:52bd LEN=104 TC=0 HOPLIMIT=46 FLOWLBL=1610612736
PROTO=ICMPv6 TYPE=129 CODE=0 ID=21800 SEQ=4 MARK=0
Jan 13 01:07:03 ice-age  IN= OUT=br0 MAC= SRC=2a01:e35:1394:5bda:222
:15ff:fe45:52bd DST=2001:200:0:8002:203:47ff:fea5:3085 LEN=104 TC=0
HOPLIMIT=64 FLOWLBL=1610612736 PROTO=ICMPv6 TYPE=128 CODE=0 ID=21800
SEQ=5 UID=0 GID=1001 MARK=0
```

Hormis le remplacement des adresses IPv4 par des adresses IPv6, on retrouve essentiellement des informations déjà présentes dans IPv4. Les nouveaux champs sont les suivants :

- **TC** : La classe de trafic, équivalente au champ **TOS** de Ipv4.
- **HOPLIMIT** : Équivalent IPv6 du TTL d'IPv4.
- **FLOWLABEL** : Identifiant de flux **[RFC3697]**, il permet la classification par les routeurs des paquets appartenant à un même flux. L'identifiant de flux peut être calculé à partir des coordonnées IP ou être fixé de manière plus spécifique.

5.2.2 Installation de nulog2

Nulog2 est une interface web d'analyse des journaux Netfilter. Elle est capable d'analyser des journaux IPv4 et IPv6. Nous allons décrire son installation afin de réaliser une analyse des informations collectées par ulogd2.

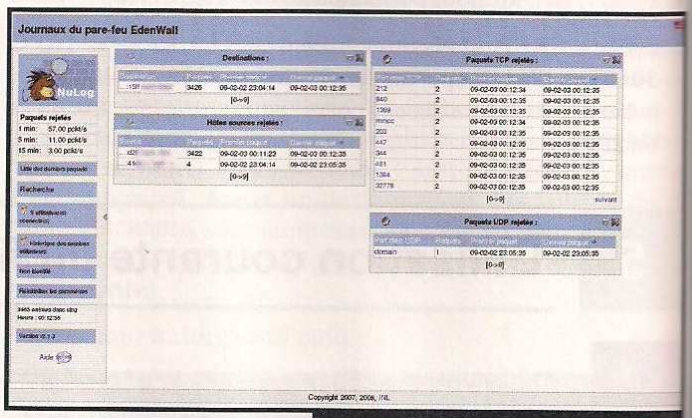


Figure 1 : Page d'accueil de Nulog

Note

Les heureux utilisateurs de Debian peuvent utiliser les paquets mis à disposition par INL sur le site **packages.inl.fr** pour installer Nulog avec un simple **apt-get install nulog**.

Les sources de Nulog sont disponibles à l'adresse **http://software.inl.fr/trac/wiki/EdenWall/NuLog#Download**. À noter qu'il vous faut au moins la version 2.1.4 de Nulog pour pouvoir s'interfacer avec une base ulogd2. Une fois récupérée, il suffit de les décompresser. Le fichier **INSTALL** contient

les instructions d'installation et les dépendances. Les dépendances sont **twisted**, **nevow**, **matplotlib**, **gettext**, **soappy**, **mysqldb**, **cairo**, **python-ipy**, **python-numpy**, **python-docutils** et, dans notre cas, **psycopg2**. Une fois les dépendances résolues, nous pouvons passer l'initialisation de la base MySQL qui ne nous concerne pas. Pour installer Nulog, il suffit alors de lancer :

```
./setup.py install
```

Pour pouvoir utiliser Nulog2 avec la base PostgreSQL remplie par ulogd2, il faut injecter une vue dans le schéma de la base. Cette vue émule une base Nulog classique et permet l'interopérabilité. Pour ajouter cette vue, il utiliser la définition **nulog.sql** fournit dans le répertoire **scripts** des sources de Nulog et il faut lancer :

```
postgres@ice-age:~$ psql -U ulog2 -h localhost ulog2 -f nulog.sql
```

Dans le répertoire **/etc/nulog**, il faut copier **default.core.conf** en **core.conf** pour pouvoir paramétrer le *backend* de Nulog. Hormis les paramètres de base, il faut spécifier le type de base ulogd2 et l'utilisation d'une base IPv6 :

```
# Database configuration
[DB]
# Hostname
host=localhost

# Database to use
db=ulog2

# Username
user=nulog

# Password
password=pupuce

# Type of database
# - mysql
# - pgsql
dbtype=pgsql

# Type of sql scheme
# - ulog: standard use of database, with ipv?.sql
# - ulogd2: the new ulogd2 database type (nulog-*-ulogd2.sql)
# - triggers: get perfs if you use triggers.py script
type=ulogd2

# IP version used in database
ip=6

# Main table
table=nulog
```

Si vous n'avez pas installé Nulog à partir des paquets Debian, il faut copier **default.wrapper.conf** en tant que **wrapper.conf** et éditer les chemins vers les modules si nécessaire. La dernière étape consiste à lancer **scripts/install_defconf.sh** pour créer le répertoire des préférences utilisateurs et installer la configuration par défaut.

```
$ cd scripts/
$ ./install_defconf.sh
```

La réponse aux questions et le paramétrage des chemins dans le fichier **wrapper.conf** finalisent l'installation.

Il suffit maintenant de démarrer Nulog. Cela se fait tout simplement en lançant la commande **nulog**. L'interface web est alors accessible depuis l'adresse http://localhost:8080/.

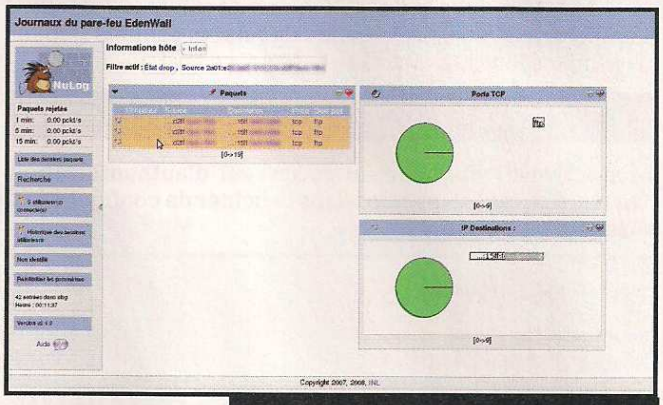


Figure 2. : Informations sur une IP source

5.3 NuFW en IPv6

La pare-feu authentifiant NuFW supporte IPv6 depuis la branche 2.2 (c'est aujourd'hui la branche stable). Pour l'installation de NuFW, je vous invite à vous référer à l'article paru dans **[Linux Mag 84]**. À des fins de concision, nous ne détaillerons ici que l'installation depuis les paquets Debian disponibles sur `packages.inl.fr` :

```
# apt-get install nufw nuauth
```

Pour activer IPv6 au niveau du serveur d'authentification `nuauth`, il suffit de spécifier dans le fichier de configuration `nuauth.conf` :

```
nuauth_client_listen_addr="[::]"
…
nuauth_push_to_client=0
nuauth_user_check_ip_equality=0
```

La première valeur fixe l'adresse d'écoute. Ici, on écoute en IPv6 uniquement, il faut rajouter `0.0.0.0` pour écouter aussi en IPv4. La deuxième et la troisième variables prennent en compte le fait que l'utilisateur doit pouvoir authentifier des paquets IPv6 et IPv4. `nuauth_push_to_client` passe `nuauth` dans un mode où il ne fait pas de requête de rafraîchissement aux clients connectés à l'IP source des paquets reçus. Dans ce mode, le client vérifie à intervalles réguliers si des paquets sont à authentifier. `nuauth_user_check_ip_equality` supprime, quant à lui, une vérification de cohérence entre l'adresse source du tunnel TLS et l'adresse source des paquets authentifiés. Cette deuxième variable est disponible depuis la version 2.2.21 de NuFW.

Les clients peuvent alors se connecter sur les adresses IPv6 globales du pare-feu :

```
$ /usr/local/nufw-2.2/bin/nutcpc -H 2a01:e35:1394:5bd0:222:15ff:fg45:5
2cd  -U user
...
SSL: certificate subject name (nuauth.inl.fr) does not match target
host name '2a01:e35:1394:5bd0:222:15ff:fe45:52bd', but continuing
(check is disabled by config)
Server Certificate OK
Enter password:
Using mechanism PLAIN
Authentication started...

nutcpc 2.2.21 $Revision: 5413 $ started (debug)
[+] Client is asked to send new connections.
```

Le client ainsi connecté authentifie alors l'ensemble des paquets IPv4 et IPv6 émis par l'utilisateur.

6 Conclusion

Le support du filtrage IPv6 est récent sous GNU/Linux et les outils de l'administrateur de pare-feu sont encore peu nombreux à supporter IPv6 notamment au niveau de la journalisation.

Le déploiement d'IPv6 se fait progressivement et le protocole continue d'évoluer pour corriger certains défauts et préparer l'avenir. Le développement d'IPv6 est encore très actif, notamment du côté d'anycast, la mobilité et le chiffrement (ESP et AH). Pour en savoir plus, consultez le livre **[IPv6 Théorie et Pratique]**, une référence en français dont le site Internet est une mine d'informations.

Les mécanismes mis en œuvre par ce protocole (comme l'autoconfiguration ou Mobile IPv6) peuvent se montrer difficiles à filtrer et IPv6 promet donc de nombreuses années de bonheur pour les développeurs de pare-feu. Les administrateurs de pare-feu s'amuseront eux avec la fin de la translation d'adresse.

Références

- **[IPv6 Théorie et Pratique]** CIZAULT (Gisèle), O'Reilly, http://livre.point6.net/
- **[IPv6 Routing Header Security]** BIONDI (Philippe) et EBALARD (Arnaud), CanSecWest 2007, http://www.natisbad.org/RH0/
- Nulog : http://software.inl.fr/trac/wiki/EdenWall/NuLog
- Paquets Debian INL : http://software.inl.fr/trac/wiki/packages
- **[RFC 791]** *Internet protocol*, 1981.
- **[RFC 2460]** *Internet Protocol, Version 6 (IPv6) Specification*, 1998.
- **[RFC 2461]** *Neighbor Discovery for IP Version 6 (IPv6)*, 1998.
- **[RFC 2464]** *Transmission of IPv6 Packets over Ethernet Networks*, 1998.
- **[RFC 3068]** *An Anycast Prefix for 6to4 Relay Routers*, 2001.
- **[RFC 3171]** *IANA Guidelines for IPv4 Multicast Address Assignments*, 2001.
- **[RFC 3697]** *IPv6 Flow Label Specification*, 2004.
- **[RFC 3810]** *Multicast Listener Discovery Version 2 MLDv2) for IPv6*, 2004.
- **[RFC 3879]** *Deprecating Site Local Addresses*, 2004.
- **[RFC 3927]** *Dynamic Configuration of IPv4 Link-Local Addresses*, 2005.
- **[RFC 4941]** *Privacy Extensions for Stateless Address Autoconfiguration in IPv6*, 2007.
- **[RFC 5095]** *Deprecation of Type 0 Routing Headers in IPv6*, 2007.
- **[ulogd2]** LEBLOND (Éric) et CHIFFLIER (Pierre), « Ulogd2, journalisation avancée avec Netfilter », *GNU/Linux Magazine HS* 41, page 30.
- **[Linux Mag 84]** LEBLOND (Éric), (Présentation de NUFW), *GNU/Linux Magazine* 84, 2006.

Auteurs : Victor Stinner, Éric Leblond

Victor Stinner
Développeur à INL. Auteur du Hachoir et membre actif de la communauté Python française.

Éric Leblond
Utilisateur GNU/Linux depuis 1996. Cofondateur et directeur technique d'INL. Développeur principal de NuFW.

Snort Inline

La détection d'intrusion réseau (NIDS) permet de vous alerter dans le cas d'une attaque. Une attaque peut très bien être une injection de code SQL sur un formulaire web, un parcours de répertoires pour remonter sur le fichier de mot de passe dans le cas d'un système mal configuré ou encore un scan de ports. Grâce à l'utilisation des bibliothèques que Netfilter fournit, Snort peut bloquer une attaque et fonctionner en mode IPS, c'est-à-dire la bloquer (Prevention). Nous allons voir, dans cet article, les principes de la détection d'intrusion réseau, puis mettre en œuvre une passerelle capable de bloquer les attaques arrivant sur votre réseau interne.

Auteur

- Sébastien Tricaud

1 Introduction

Les menaces à destination d'un réseau interne sont de plus en plus présentes à cause de la démocratisation d'internet, donc d'applications riches, donc de vecteurs d'attaques amplifiés et le tout automatisé par des vers ou virus. Malheureusement, ces attaques deviennent extrêmement complexes, à tel point qu'il n'est pas possible pour un être humain normalement constitué d'ingurgiter la masse d'information vue la rapidité de diffusion des vulnérabilités et des attaques.

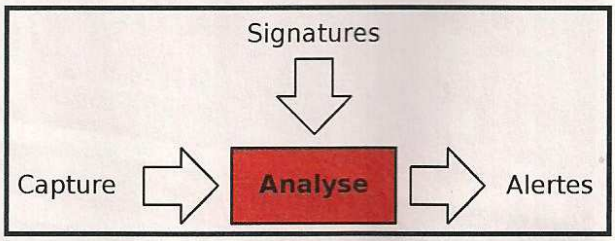
Heureusement, il existe un logiciel libre qui est capable de mettre en relation le flux réseau avec des signatures, de la même façon qu'un anti-virus va mettre en relation des signatures avec des binaires statiques. Ce logiciel s'appelle Snort et est disponible sur le site http://www.snort.org.

Mais attention, ne téléchargez pas cette version, car nous allons utiliser une variante qui s'appelle Snort Inline et qui est un projet à part entière, et a son propre site : http://snort-inline.sourceforge.net (là, vous pouvez télécharger).

Nous allons prendre les sources, version `snort_inline-2.6.1.5`.

Snort est découpé en 3 parties principales, comme le sera cet article. La première partie est celle de la capture du flux réseau :

- **Capture** : Alors que le Snort classique se base sur la `libpcap`, qui est la bibliothèque qui fût développée pour écrire le sniffeur tcpdump, Snort inline utilise la `libnetfilter queue`.
- **Signatures** : Il existe plusieurs milliers de signatures qui fonctionnent à plusieurs niveaux. Nous allons en voir quelques-unes pour comprendre comment en écrire, mais aussi comment elles peuvent se configurer dans le `snort_inline.conf`.
- **Alerte** : Ce qui est remonté à l'administrateur lorsqu'une attaque est détectée. Nous allons voir que des alertes nous sont remontées dans des cas où il n'y a pas forcément d'attaque. Cela s'appelle les « faux positifs ».



Nous allons mettre en place Snort Inline en mode pont, c'est-à-dire que la machine n'aura pas d'adresse IP.

Pour couper court à une guerre de religion récurrente sur la détection d'intrusions, la machine sera placée derrière le pare-feu, car seuls les paquets qui arrivent sur notre réseau nous intéressent. Il est insensé de mettre la machine devant le pare-feu, car internet regorge d'activités malveillantes (vers, scans, etc.) qui ne vous concernent pas forcément. Cela augmente votre charge de travail inutilement, car vous devrez filtrer les faux positifs de vraies attaques, et au final, vous n'utiliserez pas

l'outil. Le seul cas où c'est intéressant de le faire est pour le fun, pour constater en vrai ces activités malveillantes (et encore, difficile d'être mort de rire avec ça ! Ahahahah !).

Le but de cet article étant de pouvoir mettre en pratique sur votre réseau une installation de Snort Inline, nous irons à l'essentiel.

2 Installation

Nous prenons une Debian Lenny comme distribution de référence. Snort Inline peut se compiler sous toutes les grandes distributions sans problème. Il suffira d'adapter les *packages* par rapport à ce qui est disponible.

Dans un premier temps, il faut récupérer les sources, puis décompresser.

```
% wget http://ovh.dl.sourceforge.net/sourceforge/ \
snort-inline/snort_inline-2.6.1.5.tar.gz
% tar xf snort_inline-2.6.1.5.tar.gz
% cd snort_inline-2.6.1.5
```

On a besoin des bibliothèques de Netfilter et de la **libpcap** pour la capture :

```
% apt-get install libnetfilter-queue-dev libpcap0.8-dev
```

Comme Snort dépend de la **libpcre** (*Perl Compatible Regular Expression*) pour permettre l'emploi d'expressions régulières pour la recherche de motifs, on l'installe aussi :

```
% apt-get install libpcre3-dev
```

Enfin, Snort utilise la **libdumbnet** pour le premier décodage des paquets :

```
% apt-get install libdumbnet-dev
```

On récupère le patch qui va bien chez l'ami Pierre (parce que, sous Debian, on utilise **dumbnet** et pas **dnet**, qui n'a rien à voir) :

```
% wget http://www.wzdftpd.net/downloads/dumbnet.diff
% patch -p1 < dumbnet.diff
```

Puis, on lance la régénération du **configure.in** avec le patch **dumbnet**, puis le script **configure** qui va bien (support **nfnetlink**, utilisation de **dumbnet** et support des *pthreads*), pour enfin compiler, puis installer :

```
% ./autojunk.sh
% ./configure --enable-nfnetlink --with-dumbnet --enable-inline-init-
failopen
% make
# make install
```

On copie la configuration :

```
# mkdir /etc/snort_inline
# cp -r etc/* /etc/snort_inline
```

Quand Snort et sa configuration sont installés, malheureusement tout ne fonctionne pas comme prévu. Plutôt que de vous faire croire que nous sommes dans le monde des Pokémons comme n'hésitent pas à le faire certaines personnes machiavéliques et dont je tairais le nom, nous allons voir étape par étape comment comprendre l'erreur et la régler.

La capture se faisant via Netfilter, il faut positionner une règle lui disant d'aller envoyer les paquets à **nfqueue**. Dans notre cas, nous faisons une installation sur une passerelle. Tout passe par **FORWARD** :

```
# iptables -I FORWARD -j NFQUEUE
ip_tables: (C) 2000-2006 Netfilter Core Team
```

Du coup, nous pouvons démarrer Snort avec sa configuration par défaut :

```
# snort_inline -c /etc/snort_inline/snort_inline.conf
...
ERROR: OpenAlertFiles() => fopen() alert file /var/log/snort/snort_
inline-full: No such file or directory
Fatal Error, Quitting..
```

Le répertoire **/var/log/snort** n'existant pas, on le crée :

```
# mkdir /var/log/snort
```

On relance :

```
# snort_inline -c /etc/snort_inline/snort_inline.conf
...
ERROR: Unable to open rules file: /etc/snort_inline/drop-rules/
classification.config or /etc/snort_inline//etc/snort_inline/drop-
rules/classification.config
Fatal Error, Quitting..
```

Ceci est dû au fichier **classification.config** qui n'existe pas dans le répertoire voulu. On va d'abord configurer la variable **RULE_PATH** (ligne 48) :

```
var RULE_PATH /etc/snort_inline
```

Enfin, il ne nous reste plus qu'à commenter toutes les lignes incluant des signatures à partir de la ligne 666 dans le fichier **snort_inline.conf**.

Par défaut, les signatures ne sont pas installées. Il existe un projet communautaire de signatures Snort du nom d'*Emerging threats*, qui distribue ses signatures via le site web http://www.emergingthreats.net/.

Nous allons récupérer toutes les signatures dans un seul fichier, téléchargeable à l'adresse :

http://www.emergingthreats.net/rules/emerging-all.rules

```
# wget http://www.emergingthreats.net/rules/emerging-all.rules
# mv emerging-all.rules /etc/snort_inline/
```

À la fin du fichier **snort_inline.conf**, on rajoute la ligne pour inclure ce fichier de signatures :

```
include $RULE_PATH/emerging-all.rules
```

Et là, quand on lance Snort, on peut enfin voir un message qui fait plaisir :

```
Initializing Network Interface eth0
Decoding Ethernet on interface eth0
```

Ne vous inquiétez pas, il ne décode pas depuis **eth0**, mais depuis **NFQUEUE**. L'astuce consiste à traduire ce qui vient de Netfilter en PCAP, afin de toucher le moins possible au code de Snort.

Pour tester, vous pouvez lancer un **nmap** et regarder **/var/log/snort/alerts** l'alerte arriver :

```
[**] [122:1:0] (portscan) TCP Portscan [**]
```

Si vous ne voyez rien, faites bien attention à l'endroit où vous capturez avec Netfilter. Dans le pire des cas, essayez de rajouter la règle de capture sur la chaîne **INPUT** :

```
# iptables -I INPUT -j NFQUEUE
```

Cependant, faites attention de ne surtout pas vous couper l'accès à la machine ! Dans le même ordre d'idée, il vaut mieux ne pas filtrer les flux sur l'interface *loopback*.

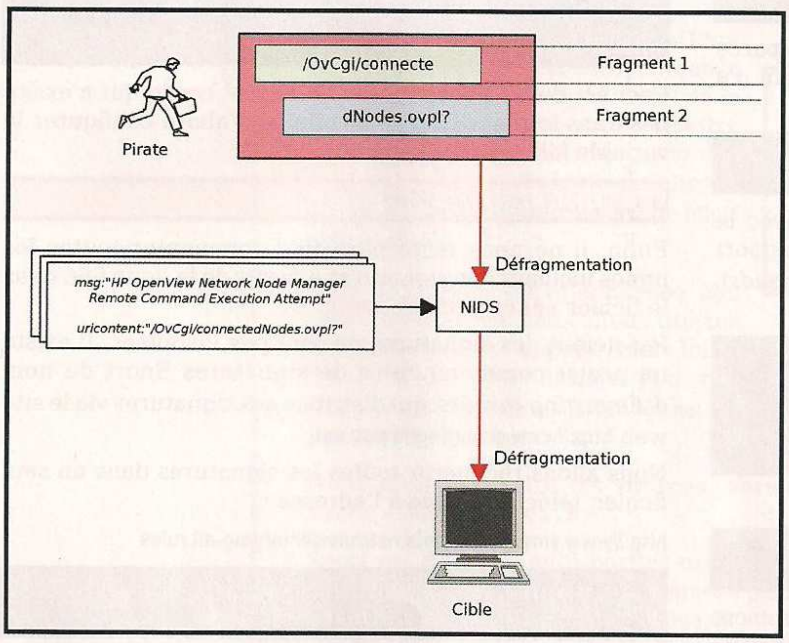
Évidemment, il reste encore des réglages à faire, que nous allons développer un peu plus loin. Mais, vous avez déjà une installation fonctionnelle vous permettant de tester.

3 La détection d'intrusion réseau

Maintenant que le paquet est arrivé, il y a toute une série d'opérations que Snort va faire afin d'empêcher une intrusion sur votre réseau.

3.1 La fragmentation des paquets

Si un paquet trop grand est émis sur le réseau (plus grand que le paramètre MTU), il va être fragmenté en plusieurs paquets, et envoyé en plusieurs morceaux. Cependant, le réassemblage ne se fera pas de la même façon suivant la pile du système d'exploitation sur lequel le NIDS fonctionne comparé au système cible. C'est-à-dire que le motif d'une attaque peut se retrouver dans plusieurs paquets qui ne se réassembleront pas pareil (en jouant sur l'*overlapping*), et ainsi l'attaquant contournera Snort.



Le problème avec la fragmentation provient du fait que lorsque la RFC sur IP a été transformée en code, certaines parties n'étaient pas suffisamment détaillées. Cela a eu pour résultat d'avoir les systèmes d'exploitation implémentant de manière différente certains mécanismes permettant d'ailleurs à **p0f** ou à **nmap** de déduire le système d'exploitation uniquement avec quelques tests actifs ou passifs sur la pile.

Il faut bien comprendre que le système de détection d'intrusion n'est pas le système d'exploitation cible et que certains outils comme Fragroute (http://monkey.org/~dugsong/fragroute/) exploitent cette faiblesse.

Il y a plusieurs manières d'empêcher ce problème : Snort, à travers son préprocesseur frag3, permet de créer des pseudo-paquets qui sont entièrement réassemblés à la manière du système d'exploitation cible configuré.

Une autre méthode consiste à empêcher complètement l'overlapping, car il n'est pas normal d'en avoir sur son réseau.

3.2 Le décodage

Une fois que nous sommes sûrs de récupérer les paquets réassemblés et que l'on arrive à capturer tous les flux réseau, il nous reste une étape importante : le décodage.

Il s'agit d'arriver à empêcher un attaquant de faire de l'évasion de signatures en exploitant les faiblesses des applications.

Par exemple, si nous prenons une signature faisant de la reconnaissance d'attaque en se basant sur une URL web, la signature contiendra le motif dans l'option **uricontent** comme ceci :

```
uricontent:"/cmd.php?"
```

Dans le cas où la machine ciblée par l'attaque n'utilise pas le serveur web Apache mais IIS, le signe backslash « \ » pourra être utilisé à la place de slash « / » et, du coup, la signature pourra être contournée simplement parce que le contenu des paquets ne correspondra pas à la signature.

Cela s'applique aussi à l'UTF-8, où les signatures sont en général écrites dans le jeu de caractères ISO-8859-1 et, du

coup, la correspondance ne se fera pas... sauf par l'application ciblée et pouvant elle décoder ces caractères et du coup subir l'attaque.

En langage Snort, les décodeurs sont appelés préprocesseurs et sont activables directement depuis la configuration. Ils ont aussi un sens d'activation logique, qui suit ce qui est défini dans cette configuration.

Il y a aussi un sens selon lequel ils doivent être activés. Regardons de plus près 3 préprocesseurs :

Nom	Description	Mémoire consommée	Performances
ac	Aho-Corasick complet	élevée	Le meilleur de tous
ac-std	Aho-Corasick Standard	moyenne	élevées
ac-bnfa	Aho-Corasick NFA	faible	élevées
acs	Aho-Corasick Sparse	faible	moyennes
ac-banded	Aho-Corasick Banded	faible	moyennes
ac-sparsebands	Aho-Corasick Sparse-Banded	faible	élevées
lowmem	Utilisation faible de la mémoire	faible	faibles

Nom	Lignes de code	Description
Frag3	4479	Réassembleur de flux, détection d'attaques concernant la fragmentation...
Stream5	11292	Suivi de connexion (tcp, udp et icmp), envoi d'alertes sur des attaques ciblant des anomalies sur les protocoles qu'il suit...
http_inspect	3497	Normalisation des URI, création du *buffer* uricontent pour récupérer depuis les signatures uniquement la requête. Envoi d'alertes ciblant http...

Le nombre de lignes de codes est donné à titre indicatif pour que vous compreniez la complexité de ces différents modules (et donc de leurs vulnérabilités potentielles ;)).

Le sens d'activation de ces différents décodeurs est important, tout simplement parce qu'il y a une chaîne de dépendances : on activera d'abord **frag3**, puis **stream5** et enfin **http_inspect**. Il n'est pas possible de faire correctement de la récupération d'URI si le flux n'a pas tout d'abord été réassemblé, puisque nous sommes dans l'état où nous recevons le **PUSH | ACK** dans le sens client->serveur.

Comme dit un peu plus haut, les décodeurs se configurent directement depuis le fichier de configuration de Snort Inline.

3.3 La recherche de motifs

Rechercher une chaîne de caractères n'est pas si simple. Il existe un tas d'algorithmes qui sont en général utilisés par les applications type éditeur de texte. Linux possède aussi une API de recherche de texte (TextSearch API) et, d'ailleurs, Netfilter peut manipuler des paquets réseau par rapport aux motifs qu'ils contiennent.

Il peut être utile sur une machine limitée de changer l'algorithme que Snort utilise. Par exemple, dans le fichier de configuration, on peut activer le mode de recherche utilisant le moins de mémoire possible en ajoutant la ligne :

```
config detection: search-method lowmem
```

Par défaut, Snort utilisera l'algorithme Aho-Corasick NFA (L'algo Aho-Corasick standard ayant pour complexité O(|X|+n)). Mais, il est possible d'en changer. Voici la liste des choix possibles :

Beaucoup de choix algorithmiques sont là tout simplement parce que, Snort étant un logiciel libre, c'est tout naturellement que les chercheurs en détection d'intrusion l'ont modifié et y ont contribué dans le cadre de leurs publications.

Dans une signature, cette recherche de motifs se fait à travers l'option **content**. Mais, Snort permet aussi de faire de la recherche de motifs grâce aux fameuses *Perl Compatible Regular Expressions* (PCRE), qui s'utilisent via l'option **pcre**.

Du coup, vient la question : « Comment Snort gère la recherche de texte entre les algorithmes standards et ceux de PCRE ? ».

Tout d'abord, faire des expressions rationnelles sur des paquets réseau est extrêmement coûteux en performances.

La recherche de motifs se fait dans le fichier **fpcreate.c** et Snort cherche d'abord à récupérer les motifs les plus longs dans les signatures (fonction **fpAddLongestContent()**). Le trafic passe ensuite dans une phase de préqualification appelée « MPSE ». Ici, les signatures sont testées de manière séquentielle.

Tant que le trafic n'a pas passé cette phase de MPSE, l'option **pcre** est ignorée. Une fois que la phase est passée, il ne reste que très peu de signatures à tester (généralement une ou deux) et on peut donc appliquer directement l'algorithme DFA ou NFA que PCRE appliquera.

Si les recherches de motifs efficaces vous passionnent, il vous faut alors un site passionnant sur ce sujet, et je ne peux que vous recommander d'aller plus loin grâce au travail formidable sur ce sujet de Thierry Lecroq (http://www-igm.univ-mlv.fr/~lecroq/).

3.4 Les alertes

Lorsque votre Snort tourne, il ne faut en général pas longtemps pour que vous ayez des alertes (et si ce n'est pas le cas, Nessus est votre ami).

Les alertes sont stockées dans un fichier plat, qui, sur beaucoup de configurations, est **/var/log/snort/alerts**, mais qui, pour le cas de la version de Snort que nous utilisons, est configuré par défaut pour aller dans **output alert_full: snort_inline-full**.

Dans le **snort_inline.conf**, tout est configuré par les directives **output** et deux modes sont activés :

```
output alert_full: snort_inline-full
output alert_fast: snort_inline-fast
```

Le mode **full** enregistrera tous les en-têtes des paquets, tandis que **fast** imprimera tout sur une seule ligne au moyen de quelques suppressions d'informations.

On retrouve ces fichiers dans **/var/log/snort/**.

Si on regarde les alertes générées, on retrouvera deux types d'alertes.

Les alertes de signatures :

```
[**] [1:2466:7] NETBIOS SMB-DS IPC$ unicode share access [**]
[Classification: Generic Protocol Command Decode] [Priority: 3]
02/01-08:42:08.819519 88.189.196.218:60398 -> 88.191.82.101:445
TCP TTL:59 TOS:0x0 ID:35278 IpLen:20 DgmLen:136 DF
***AP*** Seq: 0xECB71446  Ack: 0xD014FF37  Win: 0xFA90  TcpLen: 20
```

et les alertes des préprocesseurs :

```
[**] [122:1:0] (portscan) TCP Portscan [**]
[Priority: 3]
11/29-20:08:44.417053 80.93.212.186 -> 88.191.82.101
PROTO:255 TTL:0 TOS:0x0 ID:0 IpLen:20 DgmLen:163 DF
```

La différence entre les deux est que dans le cas de l'alerte par signature, si l'on estime que l'alerte est un faux positif, il suffit de repérer le Snort ID de la signature (sur la première ligne, c'est le deuxième nombre entre crochets ; dans notre cas, c'est le numéro 2466) et d'aller éditer les signatures pour rajouter un **#** devant la ligne de l'alerte.

Dans le deuxième cas, lorsqu'il s'agit d'une alerte de préprocesseur, on voit son nom apparaître sur la première ligne entre parenthèses, avant la description de l'attaque. Il s'agit ici du module de détection de scan de ports **portscan**.

Cependant, il faut faire très attention avant d'enlever les signatures que vous estimez ne pas être une attaque : il ne faut pas que ça joue contre vous et que, du coup, vous manquiez de réelles attaques. Il vaut mieux améliorer la signature et envoyer vos modifications au projet Emerging threats.

Ce qui peut être compliqué, c'est que, dans certains cas, la signature concerne bien une attaque, mais que, dans votre cas bien précis, ce ne soit qu'un mode de fonctionnement du réseau de votre entreprise (oui, toi qui me lis et qui transmets par tes applications le contenu de tes documents par URL et qui reçoit les alertes parce que l'URL est > 1024 caractères, je pense à toi !).

Si on regarde la signature de notre première alerte, nous avons ceci :

```
winiepot:~# grep -n sid:2466 /etc/snort/rules/*
/etc/snort/rules/netbios.rules:28:alert tcp $EXTERNAL_NET any ->
$HOME_NET 445 (msg:"NETBIOS SMB-DS IPC$ unicode share access";
flow:established,to_server; content:"|00|"; depth:1; content:"|FF|SMBu";
within:5; distance:3; byte_test:1,&,128,6,relative; pcre:"/^.
{27}/R"; byte_jump:2,7,little,relative; content:"I|00|P|00|C|00 24
00 00 00|"; distance:2; nocase; flowbits:set,smb.tree.connect.ipc;
classtype:protocol-command-decode; sid:2466; rev:7;)
```

La complexité de la règle nous montre qu'il y a vraiment peu de chances pour que ce soit un positif.

Pour tout vous dire, il s'agit de la machine principale du chapitre *honeynet* français, et un Nepenthes tourne dessus pour simuler des vulnérabilités Windows et récupérer ainsi des virus et vérifier le niveau de détection de ClamAV, ainsi que des évolutions des virus. Il est donc normal qu'il y ait ce type d'alertes.

En ce qui concerne la suppression de faux positifs sur les préprocesseurs, cela devient un peu plus compliqué. Il faut en général se renseigner dans la documentation sur le type d'alertes générées par le préprocesseur, et, dans le cas où nous sommes sûrs d'avoir un faux positif, le désactiver dans la configuration de Snort. Par exemple, sur **smtp**, la configuration est :

```
preprocessor smtp: \
  ports { 25 } \
  inspection_type stateful \
  normalize cmds \
  normalize_cmds { EXPN VRFY RCPT } \
  alt_max_command_line_len 260 { MAIL } \
  alt_max_command_line_len 300 { RCPT } \
  alt_max_command_line_len 500 { HELP HELO ETRN } \
  alt_max_command_line_len 255 { EXPN VRFY }
```

Chaque ligne est une option que l'on peut supprimer ou modifier en fonction de notre environnement : si jamais votre entreprise a un nom de domaine tellement long et des noms d'utilisateurs tellement longs aussi, vous aurez tout à gagner à modifier la valeur de **alt_max_command_line_len 300 { RCPT }**, sous peine d'avoir une alerte pour chaque courriel reçu.

Certes, ce cas-là est un peu extrême, mais songer d'abord à paramétrer les préprocesseurs correspondant aux alertes que vous recevez, afin d'écarter au maximum les faux positifs.

4 Les signatures

Si l'on regarde d'un peu plus près les signatures, elles ont un format bien spécifique. Prenons une signature pour l'exemple :

```
alert tcp $EXTERNAL_NET any -> $HTTP_SERVERS $HTTP_PORTS (msg:"ET WEB Cacti
SQL Injection Vulnerability tree.php leaf_id UPDATE"; flow:established,to_
server; uricontent:"tree.php?"; nocase; uricontent:"leaf_id="; nocase;
uricontent:"UPDATE"; nocase; pcre:"/.+UPDATE.+SET/Ui"; classtype:web-
application-attack; reference:cve,CVE-2008-0785; reference:bugtraq,27749;
sid:2007897; rev:2;)
```

Il s'agit ici de détecter une injection SQL sur l'application Cacti.

Une signature est divisée en deux grandes sections :

- l'en-tête : `alert tcp $EXTERNAL_NET any -> $HTTP_SERVERS $HTTP_PORTS`
- l'option : `(msg:...` et tout le reste

L'en-tête décrit à partir de quel moment il faut aller plus loin dans l'introspection du paquet. Par exemple, ici, on cherche une vulnérabilité qui sera exploitée sur le protocole `tcp` depuis le réseau externe (`$EXTERNAL_NET`), depuis n'importe quel port (le premier `any`) vers (`->`) les serveurs HTTP (`$HTTP_SERVERS`) écoutant sur les ports classiques HTTP (`$HTTP_PORTS`).

Ensuite, nous regardons le sens des flux. Il faut qu'il soit connecté au sens TCP du terme. En général, c'est quand on a les flags PUSH et ACK (`flow:established,to_server;`). Il faut que l'URL contienne `tree.php?`, `leaf_id=`, `UPDATE` et `UPDATE` avec `SET` plus loin (que l'on recherche en Aho-Corasick). Le `nocase` veut simplement dire que l'on cherche ce motif quelle que soit la casse.

Enfin, il ne nous reste plus qu'à classifier et mettre des informations. Il s'agit d'une attaque sur une application web, possédant un CVE et un numéro de *bugtraq* (vous permettant de retrouver un éventuel exploit pour tester vos applications).

Le `rev:2` nous montre qu'il y a eu deux révisions sur la règle.

Vous pouvez bien sûr vous inspirer d'autres signatures pour faire les vôtres, et participer au projet Emerging threats pour envoyer vos signatures.

Une fonctionnalité du mode Inline est de pouvoir supprimer les attaques, et cela ne peut se faire que grâce à la fonction `drop`, que l'on met à la place de `alert` en tout début de règle. Ainsi, Netfilter fera le choix de bloquer le paquet non plus sur des simples états, destinations où sources, mais sur de véritables intrusions qui n'atteindront même pas les machines cibles.

5 Pour aller plus loin

Maintenant que vous savez utiliser Snort et comprenez quelques-uns des rouages de la détection d'intrusion, je ne peux que vous recommander d'aller un petit peu plus loin avec la sélection des paquets et la gestion des alertes.

Pour la sélection des paquets, puisque Netfilter permet de faire de la sélection basée sur l'API TextSearch du noyau Linux, nous n'allons envoyer dans la `NFQUEUE` que les paquets qui auront été choisis parce que leur texte est disponible directement dans les signatures de Snort.

Le projet FWSnort (http://cipherdyne.org/fwsnort/) permet de traduire les signatures de Snort en règles Netfilter avec cette sélection de paquet. Malheureusement, il n'est pas possible d'avoir toutes les signatures par ce biais, mais c'est cependant très pratique quand on laisse le noyau faire la recherche de motifs à la place de Snort.

En ce qui concerne la gestion des d'alertes, il existe le projet Prelude IDS (http://www.prelude-ids.org) qui permet de normaliser les alertes que Snort envoie au format IDMEF (*Intrusion Detection Message Exchange Format*). Snort peut ainsi profiter de l'interface graphique Prewikka, et coupler ses informations à d'autres logiciels de détection d'intrusions comme OSSEC, voire d'autres Snort à différents endroits du réseau afin d'agréger les alertes et aussi de pouvoir les corréler et récupérer des alertes plus pertinentes.

6 Conclusion

Snort peut se mettre en place assez facilement, et rapidement détecter des intrusions comme des comportements sur votre réseau. D'autant plus qu'en mode Inline Snort peut bénéficier de tous les avantages de Netfilter. Cependant, il ne faut pas négliger l'affinage des règles pour arriver petit à petit à un système qui vous remonte des alertes de façon assez fiable.

Auteur : Sebastien Tricaud

Responsable du chapitre français du projet honeynet. Détecteur d'intrusions au sein d'INL.

Amon, le pare-feu de l'Éducation

Auteur

- Gwenaël Rémond

Amon est un pare-feu hautement configurable développé par et pour l'Éducation nationale et utilisé ailleurs (ministères, collectivités territoriales, entreprises). Il est basé sur la distribution GNU Linux Eole, qui propose un ensemble de solutions intégrées intranet-internet, (du serveur de fichiers dans les établissements scolaires jusqu'au concentrateur VPN inter-académique, en passant par la gestion de parc et la configuration automatisée de serveurs). Au sein de l'Éducation nationale, les questions de sécurité des réseaux se posent de manière cruciale. L'outil Era, « Éditeur de Règles pour Amon », conçu à l'origine comme un éditeur de règles de pare-feu, est devenu un framework de compilation et d'interprétation de directives de sécurité. Il a permis de mettre en place dans les établissements scolaires une politique globale de sécurité à l'échelle nationale.

1 Eole et Amon

1.1 Historique

Le pare-feu Amon est né au centre informatique du rectorat de l'académie de Dijon en 2000, sous l'égide de Luc Bourdot, actuellement ingénieur de recherche et chef de projet Eole. Au départ, il s'agissait d'installer des pare-feu dans quelques établissements scolaires de l'académie avec des outils libres. Amon était alors basé sur ipchains. Le passage à iptables est venu assez rapidement avec le noyau 2.4 un an plus tard.

Luc ne se doutait certainement pas à l'époque de l'ampleur qu'allait prendre ses travaux. Au début, la génération des règles iptables était un ensemble de scripts Bash. Il est devenu, ensuite, nécessaire de mieux gérer la pile d'instructions iptables. De plus, il était nécessaire de gérer les données de configuration des serveurs (notamment pour pouvoir les centraliser). Il a été décidé de tout enregistrer dans des fichiers XML.

Aujourd'hui, les différents fichiers de configuration, scripts et autres sont générés à partir d'interfaces graphiques codées en python-GTK (`gen_config`). Il est possible de l'exécuter sur le serveur au prompt, depuis des applications Web ou GTK décentralisées.

En complément, Bruno Boiget, ingénieur d'études au pôle développement travaille sur un système de *Single Sign On* (SSO) pour toutes les applications Web Eole. Les bibliothèques de programmation réseau de bas niveau utilisent `python-twisted` (le *framework* de programmation réseau Twisted Matrix).

Et comme tous les produits, Eole, c'est du libre, diffusé sous licence CECILL. Des paquets Ubuntu sont mis à disposition sur le dépôt Eole, et pendant que j'y suis, pour compiler des paquets, `eole-epack` est un outil graphique de compilation à partir d'un dépôt Subversion (merci à Jérôme et Joël). C'est assez pratique. Cet outil sera même bientôt utilisable depuis l'extérieur. Dans la ferme de compilation Eole, on génère une distrib entière un peu comme on génère une page Web, c'est-à-dire sans plus y penser (mais si, mais si ;)

Une grande partie des logiciels Eole sont des logiciels de configuration. À l'installation, ce qui prend le plus de temps, c'est la copie du CD sur le disque dur. Pour la configuration globale, la technologie Créole (pour « Création Eole ») est utilisée. Elle est rapide et permet l'industrialisation de la mise en place des serveurs.

nationale

1.2 Amon et l'outil Era

Revenons à Era. C'est un logiciel en deux parties installé par défaut sur les pare-feu Amon. La première partie du logiciel est graphique (GTK). La deuxième partie est un outil en ligne de commande, le compilateur-générateur de règles iptables, de QOS (qualité de service) et de règles authentifiantes (grâce à l'intégration de NuFW dans Amon). Era peut être lancé directement sur un Amon ou bien depuis n'importe quel poste de travail ou même... installé sur un poste Windows (...il n'y a pas de honte. Au labo, il y a des Vmware avec des Windows dedans rien que pour voir si ça s'installe bien ;).

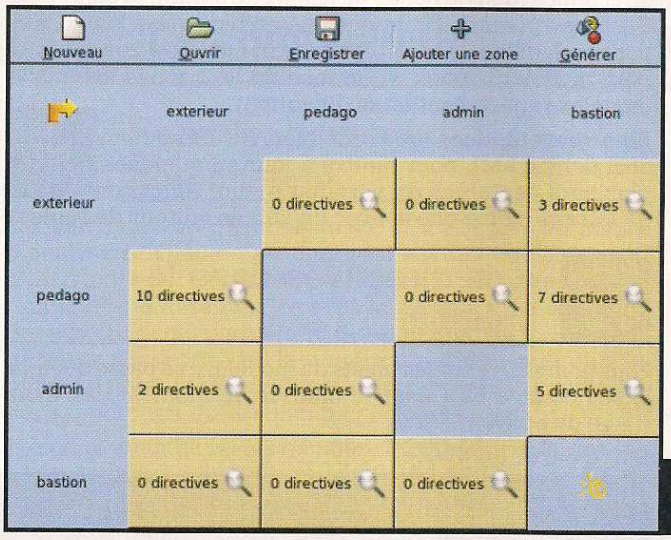


Fig. 1 : L'interface d'Era au lancement, la fenêtre principale représente le tableau des flux

À première vue, l'interface d'édition est déjà très différente de celle d'un éditeur de règles classique à la Firewall Builder. La configuration de pare-feu est orientée flux de directives. Nous y reviendrons.

Sur un Amon, après avoir édité et enregistré le XML dans l'interface Era, il faut lancer la ligne de commande en *root* pour appliquer les règles de pare-feu :

```
$ service bastion restart
```

Si vous êtes à distance, vous pouvez stocker la configuration XML sur un serveur de centralisation des configurations (le serveur Zephir). Ce serveur est utilisé dans le cadre de la gestion de parc pour déployer les modèles. Il est ainsi possible de reconfigurer plusieurs centaines de pare-feu en quelques minutes. Pratique en cas d'alerte de sécurité.

Sur un autre type de pare-feu qu'Amon, vous ne pouvez pas bénéficier de cette possibilité d'envoi de la configuration, mais vous pouvez simplement générer un script de règles iptables à lancer manuellement. Si vous êtes directement sur un Amon, vous pouvez en profiter pour réinstaller la configuration complète du système en tapant en root :

```
$ reconfigure
```

Voilà, quand même, c'est pas trop compliqué. Si vous voulez adapter le pare-feu à vos différents besoins, il va vous falloir comprendre un peu le fonctionnement de l'interface graphique.

2 La représentation du réseau avec Era

2.1 Les niveaux de sécurité

Voici l'idée principale : la problématique d'intégrité et de cohérence doit être assurée au moment de la configuration du pare-feu. Et pour raisonner sur un réseau, il est nécessaire de simplifier l'information, de sorte que l'observation du système fournisse une vue d'ensemble.

Le modèle est en fait une gestion par flux : je donne plus de droits à ce à quoi je fais le plus confiance (j'autorise du plus sécurisé vers le moins sécurisé) et je donne moins de droits à ce à quoi je fais le moins confiance. C'est le modèle de l'écoulement fluide : ça passe, sauf directive contraire (directive de type barrage), de ce qui est le plus sécurisé vers le moins sécurisé (flux descendant), et ça ne passe pas, sauf directive contraire (directive de type pont ou aqueduc), de ce qui est le moins sécurisé vers le plus sécurisé (flux montant).

Centrons-nous sur le pare-feu (c'est la zone la plus sécurisée). Autour, il y a internet (la zone extérieure, c'est la zone la moins sécurisée). Puis, d'autres cartes réseau ou tunnels (dans un établissement scolaire, il y a typiquement deux autres interfaces réseau : la zone pédagogique et la zone administrative et au moins un tunnel vers l'académie).

Affectons ensuite des niveaux de sécurité à chacune de ces zones.

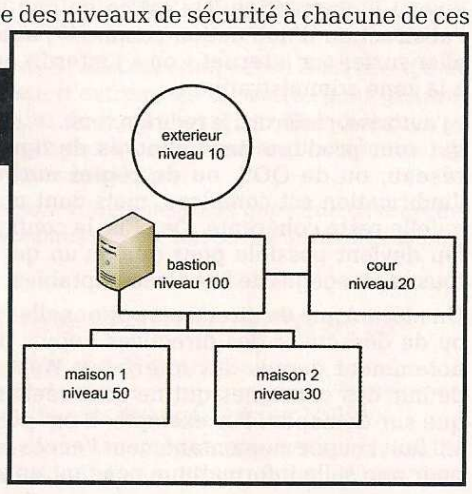


Fig. 2 : Les niveaux de sécurité

Ce choix détermine une vue par flux entre chaque zone. Par zone, entendons une carte réseau. À l'intérieur, vont être définies des directives de sécurité. Et,

enfin, une politique par défaut (par exemple, sauf directive contraire, ce qui vient de l'extérieur est peu sécurisé, donc doit être bloqué).

2.2 Le modèle devient exécutable

Au niveau du code, le modèle en mémoire après chargement du XML ressemble à ce qu'on peut reproduire ici au prompt Python après avoir importé les objets de la bibliothèque :

```
>>> from era import fwobjects
>>> z1 = Zone('z1', 10)
>>> z2 = Zone('z2', 50)
>>> assert z1 < z2
True
>>>
```

Pour créer une zone, il faut un nom et un niveau de sécurité (en plus de l'interface réseau). Les zones sont alors immédiatement ordonnées les unes par rapport aux autres, ce qui donne le tableau des flux montants et descendants une fois inséré dans l'objet matrice de flux :

```
>>> matrix = MatrixModel()
>>> matrix.add_zone(z1)
>>> matrix.add_zone(z2)
>>>  # deux flux sont créés :
[Flux : [z1 <===> z2], Flux : [z2 <===> z1]]
>>>
```

Le modèle exécutable apparaît comme bien plus pratique dans le cadre des méthodes agiles (en principe, au labo, on programme toujours en binôme, et les développements sont dirigés par les tests). Il dérive directement de la représentation que l'on se fait du réseau, la logique métier, sans avoir à être décrit dans un ou plusieurs langages de haut niveau ou un langage de configuration.

Bien sûr, ces créations d'objets sont transparentes. Elles se font naturellement depuis l'interface graphique (par exemple, la fenêtre d'édition des zones). Ce qu'il est important de constater, c'est que, derrière l'interface graphique, il y a un modèle objet qui évolue au fur et à mesure de la conception du pare-feu.

2.3 Les directives de sécurité

Au plan des décisions à prendre, il existe des niveaux d'analyse qu'il faut privilégier. La notion de directive représente ce niveau d'abstraction. Qu'est-ce qu'une directive ? C'est l'abstraction d'une action comme « j'autorise les profs à aller surfer sur internet » ou « j'interdis aux élèves l'accès à la zone administrative ».

« j'autorise, j'interdis, je redirige vers... », des actions simples qui vont produire des quantités de lignes d'instruction réseau, ou de QOS, ou de règles authentifiantes dont l'imbrication est complexe, mais dont nous sommes sûrs qu'elle reste cohérente. De plus, la configuration de pare-feu devient possible pour quelqu'un qui ne maîtrise pas tous ces langages de bas niveau (iptables, *trafic control*...).

Un mécanisme de directives optionnelles permet d'activer ou de désactiver des directives depuis diverses sources, notamment depuis des interfaces Web. Cela permet de définir des consignes qui ne sont réellement appliquées que sur demande. Par exemple, il est possible de décider s'il faut couper momentanément l'accès au Web ou au chat pour une salle informatique pendant un cours.

Les directives dépendent de l'échelle. À une grande échelle, (c'est-à-dire une petite surface, celle d'une salle info par exemple), il faut pouvoir rendre la configuration adaptable. À une plus petite échelle (une plus grande surface, l'échelle d'un établissement ou d'une académie), on peut choisir de rediriger systématiquement les élèves vers le proxy établissement.

En plus, un mécanisme de directives dites « cachées » est prévu : ces directives ne s'activent que si nécessaire. Par exemple, Amon peut détecter s'il dispose d'un proxy installé sur le serveur et ainsi l'activer en conséquence.

Enfin, à une échelle encore plus petite, l'échelle nationale, Eole fournit les moyens techniques pour appliquer de grandes orientations et préconisations de sécurité en proposant des modèles. Ce sont des fichiers XML, qu'on appelle « fichiers de modèles de pare-feu prédéfinis » (modèle 3 zones, 4 zones, 5 zones).

Ces modèles XML peuvent être « *templatisés* », c'est-à-dire que les valeurs des IP et des réseaux ne sont pas renseignées directement dans le fichier de manière à assurer la généricité. Il est aussi possible d'imbriquer hiérarchiquement les modèles pour en faciliter la maintenance (un modèle 4 zones hérite d'un modèle 3 zones, si on modifie le 3 zones les autres modèles répercutent la modification).

Bien souvent, dans les académies, il suffit de modifier un seul fichier XML pour mettre à jour en quelques minutes plusieurs centaines de pare-feu Amon. Allez donc tenter de faire la même chose avec un simple éditeur de règles !

2.4 Les directives dans le modèle

Pour poursuivre sur les objets du modèle, il se passe quelque chose comme ceci dans la matrice au moment de l'ajout d'une directive :

Soit à partir du XML :

```
>>> dir_node = '''<directive id='1' service='serv1' priority='1'
action='1' libelle='directive'>
<source name='extr1' /><source name='extr2' /><destination name='extr3' />
</directive>'''
>>> xmldoc = parseString(dir_node)
>>> directive = domparsers.instantiate_directive_from_dom(xmldoc)
```

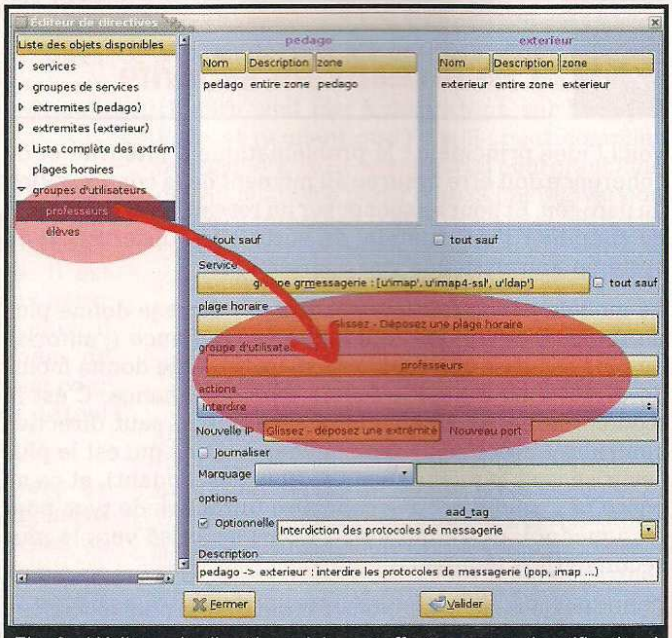


Fig. 3 : L'éditeur de directives. Ici pour affecter une authentification à la directive, il suffit de glisser-déposer le groupe d'utilisateurs choisi.

Soit depuis l'interface d'édition des directives, il se passe alors cela :

```
>>>directive = Directive([extremite1], [extremite2]], service1, ACTION_DENY, 1)
```

Puis, dans tous les cas, la directive est insérée dans la matrice au bon endroit dans les flux :

```
>>>matrix.add_directive(directive)
```

2.5 Utilisation de l'interface graphique

Voilà, maintenant que vous en savez plus sur le modèle, la prise en main de l'interface graphique sera plus évidente. Le but du jeu est de créer des directives. Au lancement, vous avez le tableau des flux. Vous pouvez alors ajouter ou supprimer des zones. Ce tableau a autant de lignes et de colonnes que de zones. En cliquant droit sur une zone, vous pouvez accéder à sa configuration et y créer des extrémités (ajout de machines ou de sous-réseaux).

Dans le tableau, il y a autant de cases que de flux. Cliquez sur une case pour ajouter une directive, vous obtiendrez la liste des directives existantes. Lors de l'ajout, vous accéderez à l'éditeur de directives fonctionnant par glisser-déposer d'extrémités, de services et/ou d'utilisateur.

Pour entrer un peu dans les détails, une extrémité est une IP, une plage d'IP ou un sous-réseau. Un service est un ensemble protocole, port ou plage de ports. L'action peut-être une autorisation, une interdiction, une redirection ou du DNAT ou SNAT. La matrice comptabilise les directives et leur priorité, et les ajoute aux bons emplacements dans les flux.

Pour plus de détails, je vous renvoie au manuel d'utilisation.

3 Le compilateur

3.1 Fonctionnement du compilateur

Du point de vue de la sécurité, il n'y a pas de niveau d'analyse privilégié : au final tout est ramené à une pile d'instructions réseau. Récapitulons : ces échelles sont toutes très différentes, ce sont le modèle et la définition des directives qui permettent de les unifier. Mais, comment garantir l'intégrité et la cohérence d'une pile entière d'instructions iptables ? Et entre les différents réseaux ? Et par rapport aux différentes technologies (ipsec, QOS...) ?

Le compilateur de règles récupère les directives, les classes par types (directives standards en ACCEPT ou en DROP, de redirection, de DNAT ou de SNAT) et génère les règles iptables, les règles de QOS et les règles authentifiantes correspondantes.

Regardons comment ça se passe au niveau du code et décomposons les différentes étapes :

Récupérons la matrice :

```
>>> from era.initialize import initialize_app
>>> matrix_model = initialize_app('3zones.xml')
```

Initialisons le compilateur iptables :

```
>>> from era.compiler import Compiler
>>> from era.iptwriter import IPTWriter
>>> compiler = Compiler(IPTWriter, sys.stdout)
```

Passons la matrice en paramètre au compilateur :

```
>>> compiler.compile(matrix_model)
```

Les règles iptables sont alors générées (dans ce cas, sur la sortie standard).

Il y a des processeurs de directives. Ce qui revient, lorsqu'on dispose d'une directive, à faire ceci :

```
>>> proc = era.processors.get_processor(directive)
>>> rules = proc.process()
>>> for rule in rules:
>>>     writer.append_rule(rule)
```

À un processeur, correspond un type de directive (DNAT, SNAT,...).

Observez la dernière ligne ci-dessus (l'itération sur la liste). Ce sont des objets modélisant une règle iptables.

3.2 La génération des règles iptables

Maintenant, nous arrivons au niveau des règles iptables. Au passage, le compilateur intègre une modélisation objet de bas niveau des règles iptables.

Là aussi, plutôt que de créer une syntaxe de haut niveau qui pourra ensuite être compilée en langage de plus bas niveau, avec Era, il y a un modèle exécutable, mais sans syntaxe (sans langage associé). Chacun sait que la définition d'un langage et de son arbre de syntaxe est lourde en temps et en moyens. Les projets comme HLFL (*high level firewalling language*) ou WallFire cherchent depuis des années à définir des syntaxes possibles.

Au niveau du compilateur, une directive n'est pas nécessairement associée à une unique règle iptables. Suivant le type de directive, le comportement du compilateur est différent. En effet, il se peut que le compilateur rajoute des règles implicitement, par exemple, dans le cas d'une redirection, une règle iptables de **FORWARD** doit être accompagnée d'une règle **INPUT** (si je redirige les élèves depuis internet sur le port 3128 d'un proxy de la dmz, je dois bien ouvrir ce port sur le bastion d'abord).

Une seule directive, si elle est composées d'un groupe de services, d'une liste d'extrémités ou autre, peut générer une grande quantité de lignes iptables. Une directive va générer autant de règles qu'elle compte d'extrémités, de services et des règles implicites.

Toutes ces règles sont indispensables pour que ça marche. Une seule action de haut niveau doit se traduire en plusieurs règles iptables.

3.3 Le filtrage authentifié et la qualité de service

Les règles authentifiantes sont gérées par NuFW.

Era récupère les directives authentifiantes au niveau du compilateur iptables et rajoute l'instruction **-J NFQUEUE**.

De plus, pour les redirections, un marquage est affecté. Cette marque est spécifique à chaque groupe d'utilisateur. Un fichier d'ACL est généré sous cette forme :

```
[directive_description]
proto = 6
srcport = 1024-65535
outdev = eth0
indev = eth2
gid = 10001
…
```

Le GID permet de déterminer à qui appartient cette ACL.

Quant à la QOS, elle est limitée à la patte externe. Un script est généré. Il représente une sous-partie du modèle global de la matrice de flux.

C'est réglable depuis l'interface graphique. Il s'agit de diverses poignées de déplacement qui permettent de répartir la bande passante vers les différentes zones. La taille allouée est proportionnelle et la vue correspond simplement à des pourcentages de la bande passante totale (que l'on définit).

3.4 La simulation et les tests

La sémantique d'Era (c'est-à-dire ce que fait effectivement le programme) est conforme à sa spécification (c'est-à-dire ce que l'on voulait que le programme fasse). Les techniques de tests de génération des règles sur des jeux de réseau ou de simulation sur un réseau physique passent difficilement à l'échelle : en bref, il est impossible de traiter tous les cas. Notre manière de vérifier a été de faire des tests, de constater que le programme correspond bien dans un grand nombre de cas donnés.

Il y a eu pas mal de tests et on aurait pu encore pousser ça avec des simulateurs comme nf-sim, un simulateur Netfilter dans l'espace utilisateur. En exécutant ou en simulant le programme dans un grand nombre d'environnements suffisamment représentatifs des exécutions possibles, il devient clair qu'on peut dormir sur ses deux oreilles.

3.5 La compilation dynamique

Récapitulons : une règle iptables générée aura beau être syntaxiquement correcte et bien positionnée, si elle ne s'inscrit pas dans un contexte elle pourra être sémantiquement fausse (la sémantique dans le sens où ce que fait effectivement la pile d'instructions réseau).

Le choix qui a été fait est une modélisation qui prend de la distance par rapport à une syntaxe. C'est l'interface graphique, donc le modèle, qui permet de fédérer les décisions, ainsi que le compilateur et son comportement de bas niveau.

Y a-t-il un moyen de faire du « reverse », de décompiler ? De passer d'un résultat de iptables -L ou iptables-save à un modèle objet ? C'est possible en l'état actuel du compilateur. C'est possible parce que le compilateur est statique, c'est-à-dire non contextuel.

Mais, si l'on veut dépasser les limitations liées au cumul des caractéristiques d'une directive (l'authentification, la qualité de service, le marquage, la journalisation…), le modèle lui-même doit devenir dynamique.

La modélisation par flux et les espaces d'objets qui constituent le modèle doivent devenir dynamiquement adaptables, contextuels. Un peu comme une syntaxe dont la grammaire serait contextuelle. Cela devient indispensable notamment si l'on veut approfondir les possibilités dans Era liées au filtrage authentifié.

4 Conclusion

Dans l'avenir, le compilateur devra modifier son comportement en fonction de la pile des instructions réseau qu'il rencontre. Si la compilation et les espaces d'objets dynamique vous intéressent, je vous suggère pour approfondir cette question d'aller faire un tour du côté de PyPy, le compilateur dynamique du langage dynamiquement typé Python, codé en Python lui-même, et de revenir lire le code d'Era dans… pas pour tout de suite en tout cas…

Que Patrick McHardy change nos habitudes en nous proposant nftables, peu importe. C'est très certainement mieux. Les règles en seront plus condensées, mais les problématiques d'accumulation resteront : il s'agit toujours d'une pile d'instructions réseau. Que les règles soient générées ou non, si on n'y prend pas garde, elles finissent par tourner au plat de spaghettis, se confondre, puis, au final, se contredire les unes les autres, d'où l'importance de disposer d'une vue d'ensemble.

Jusqu'à présent, les outils Eole se sont adaptés à une échelle de travail nationale sur des milliers de serveurs et avec des centaines de milliers d'utilisateurs (les élèves, les profs et les administratifs). Nous allons continuer nos efforts.

Auteur : Gwenaël Rémond

Consultant indépendant, développeur principal Era intervenant au pôle développement de l'équipe Eole à Dijon depuis 2001

Remerciements

Merci à toute l'équipe Eole, à son chef de projet ainsi qu'aux différents acteurs et contributeurs Era, tout particulièrement Samuel Morin, Klaas Tjebbes, Emmanuel Garette, Jérome Soyer, Joël Cuissinat, ainsi qu'à Bruno, Gaston, Laurent, David, Adrien…

Références

- Le projet Eole : http://eole.orion.education.fr
- La page d'accueil Era : http://eole.orion.education.fr/diff/article.php3?id_article=29
- Le wiki sur Era : http://eole.orion.education.fr/wiki/index.php/Era et la documentation sur la matrice de flux : http://eole.orion.education.fr/wiki/index.php/EraPresentation
- Une introduction à l'interprétation abstraite : http://www.di.ens.fr/~cousot/AI/IntroAbsInt.html
- L'interprétation abstraite dans PyPy et les espaces d'objets : http://codespeak.net/pypy/dist/pypy/doc/theory.html

NFqueue, la météo et autres applications

Le pare-feu Netfilter permet de faire tout ce qu'un pare-feu au niveau noyau est capable de faire : filtrer des paquets en fonction de critères tels que l'adresse source, le port de destination, etc. Si cela semble suffisant la plupart du temps, il peut être utile d'avoir accès à des bibliothèques pour accéder, par exemple, à un annuaire où à une base de données. Nous allons voir comment fonctionne le weatherwall, le pare-feu next-generation et d'autres applications utiles (ou pas).

Auteur

- Pierre Chifflier

1 Netfilter, vie et mort des paquets

Le travail de Netfilter consiste à aiguiller, en fonction de critères de décision, les paquets vers des actions (cibles) terminales. C'est assez semblable à notre compagnie de transport nationale, avec un brin de fiabilité et de prédictibilité en plus (quoique, le module `FUZZY` permet d'obtenir le même comportement...). Les cibles les plus classiques sont `DROP`, `REJECT` et `ACCEPT`.

Cependant, la liste des critères de décision est assez limitée, et il est très difficile de l'étendre : la programmation se fait en mode noyau. Il n'y a donc pas accès à des bibliothèques externes. Il faut *rebooter* assez fréquemment (la moindre erreur ne pardonne pas), et, pire encore, il faut programmer proprement. Pour résoudre ces problèmes, Netfilter a introduit plusieurs cibles terminales : `QUEUE` et `NFQUEUE`. Ces cibles permettent d'envoyer le paquet en espace utilisateur pour qu'une application puisse effectuer sa propre analyse et décider du verdict. Des cibles similaires existent pour le log : `LOG` et `NFLOG`.

Ces cibles sont déjà utilisées par plusieurs applications majeures, telles que **Snort-Inline** (IDS/IPS), **NuFW** (pare-feu authentifiant), **ulogd** (journalisation Netfilter, voir l'article dans ce magazine).

Cependant, la programmation se fait toujours en langage C. Ce n'est pas forcément un problème, mais ça ne facilite pas le développement rapide. Donc, on a troqué le vilain *kernel panic* contre un vilain *segmentation fault*.

L'objectif de cet article est de montrer comment, en combinant les fonctionnalités de la cible `NFQUEUE` et de `SWIG`, on peut créer des modules de développement rapide d'applications de filtrage, et comment ces modules peuvent conduire à écrire des applications pas toujours très sérieuses.

2 Swig

2.1 Swig, c'est quoi ?

SWIG (*Simplified Wrapper and Interface Generator* **[3]**) est un générateur d'interfaces : il permet de générer automatiquement la couche d'abstraction permettant d'utiliser des modules en C et C++ depuis des langages de haut niveau. SWIG supporte la génération pour de nombreux langages tels que Python, Perl, Ruby, Lua, et même Java ou C# (comme ça, tout le monde trouvera de quoi *troller*).

Par rapport à l'utilisation directe de l'API de chaque langage (par exemple perlxs ou ctypes), SWIG présente l'inconvénient d'être légèrement plus lent (et encore, ça reste à prouver). Par contre, il a l'avantage d'être générique (un seul code pour tous les *wrappers*), et d'être assez flexible pour nos besoins : il supporte l'utilisation de fonctions de conversion, et permet de créer un résultat orienté objet.

Cette dernière fonctionnalité est d'autant plus intéressante que, dans `nfqueue-bindings`,

nous n'avons pas envie d'exposer les fonctions de la même manière dans le langage de haut niveau, mais plutôt de les encapsuler de manière à les simplifier.

Grâce à SWIG, on va donc pouvoir troquer notre vilain *segmentation fault* contre un joli *traceback* python de 20 lignes.

2.2 Exemple simple

Mais assez parlé, passons à l'utilisation de SWIG. Imaginons que nous ayons le fichier **file.h** suivant :

```
#ifndef __MY_FILE__
#define __MY_FILE__

int    fact(int);
int    my_mod(int n, int m);

#endif
```

Le fichier **file.c** correspondant :

```
#include "file.h"

/* Compute factorial of n */
int fact(int n)
{
  if (n <= 1) return 1;
  else return n*fact(n-1);
}

/* Compute n mod m */
int my_mod(int n, int m)
{
  return(n % m);
}
```

Nous souhaitons pouvoir utiliser ces fonctions dans un langage de haut niveau (ici, on prend comme exemple Python). Nous allons donc créer un fichier d'interface de SWIG, **example.i**, pour pouvoir ensuite générer un module Python. Normalement, on devrait déclarer toutes les fonctions dans le fichier SWIG, mais il y a beaucoup plus simple : il suffit d'inclure le fichier **header**.

```
/* File : example.i */
%module example
%{
/* Put headers and other declarations here */
#include "file.h"
%}

%include "file.h"

%extend my_struct_t {
        int my_func(int);
}
```

La directive **%module** permet de préciser le nom sous lequel le module pourra être importé dans Python. On invoque ensuite SWIG en utilisant la commande **swig**, et en précisant le langage de destination :

```
swig -python example.i
```

Cette commande va créer deux fichiers, **example.py** et **example_wrap.c**. Ce dernier doit être compilé avec nos autres fichiers (**file.c**) pour former une bibliothèque dynamique, qui sera chargée par le module Python. On compile donc le tout :

```
gcc -c -fpic file.c example_wrap.c -I/usr/include/python2.5/
gcc -shared -o _example.so file.o example_wrap.o -lpython2.5
```

Notez que le nom du fichier de sortie est celui du module, préfixé du caractère « _ » (souligné). Le module est prêt, on peut l'utiliser :

```
$ python
>>> import example
>>> example.fact(5)
120
```

2.3 Code orienté objet

Le code précédent aurait été suffisant pour pouvoir encapsuler l'intégralité des fonctions de **libnetfilter_queue**. Mais, nous ne souhaitons pas « juste » proposer les fonctions à Python. Ça n'aurait pas beaucoup d'intérêt (et c'est déjà possible en utilisant des choses comme **ctypes**). L'autre objectif est d'encapsuler les fonctions pour pouvoir faire de la programmation objet, et SWIG va nous aider.

SWIG est capable de convertir du code C++, en conservant la notion d'objet dans le langage de destination. Mais, cela nous aurait obligé à écrire une couche d'abstraction C++ à la main, donc le gain aurait été faible. Nous allons plutôt utiliser des structures C, parce que SWIG offre la possibilité assez sympathique d'étendre ces structures en leur ajoutant des fonctions, qui deviennent donc des fonctions membres.

On ajoute maintenant la structure suivante au fichier **header** :

```
struct my_struct_t {
 int var;
};
```

SWIG va générer un objet Python opaque (de type **proxy**, pour transférer tous les appels sur l'objet Python vers l'objet C correspondant, en convertissant les arguments si besoin est).

```
>>> s = example.my_struct_t()
>>> print s
<example.my_struct_t; proxy of <Swig Object of type 'my_struct_t *' at 0x85cc440> >
```

Pour ajouter des fonctions membres, il faut utiliser l'instruction **%extend** de SWIG, dans le fichier interface :

```
%extend my_struct_t {
        int my_func(int);
}
```

Cette instruction permet d'ajouter des fonctions membres à l'objet correspondant à la structure. La fonction C correspondante doit exister, et sa signature doit être du type :

```
code_retour <nom_structure>_<nom_fonction>(type_structure *self,
autres_arguments);
```

Dans notre exemple, la signature sera la suivante :

```
int my_struct_t_my_func(struct my_struct_t *self, int n)
{
  if (self == NULL)
    return;

  self->var += n;
  return self->var;
}
```

C'est fini ! Après recompilation, nous voilà capable d'accéder à la fonction comme pour tout objet python :

```
>>> s = example.my_struct_t()
>>> s.var = 0
```

```
>>> s.my_func(42)
42
```

```
>>> print s.var
42
```

3 nfqueue-bindings

3.1 Cmake !

Après avoir utilisé (et bataillé avec) automake et autoconf, chacun se fera son opinion sur ces outils. De mon côté, le choix a été vite fait : **Cmake** !

CMake va nous servir pour détecter les bibliothèques nécessaires à la compilation, pour créer les fichiers Makefile, qui serviront à gérer la compilation et l'installation.

Pour les pré-requis, la détection sera extrêmement simple : le minimum est SWIG, et `libnetfilter_queue` :

```
FIND_PACKAGE(SWIG REQUIRED)
INCLUDE(${SWIG_USE_FILE})

INCLUDE(UsePkgConfig)
PKGCONFIG(libnetfilter_queue LIBNFQ_INCLUDE_DIR LIBNFQ_LINK_DIR LIBNFQ_
LINK_FLAGS LIBNFQ_CFLAGS)

ADD_SUBDIRECTORY(python)
ADD_SUBDIRECTORY(perl)
```

Voilà, CMake se charge de trouver et d'exporter les variables requises. Pour les variables spécifiques, la recherche se fera dans le répertoire associé (`perl` ou `python`). Par exemple (pour Python) :

```
FIND_PACKAGE(PythonInterp)
FIND_PACKAGE(PythonLibs)
```

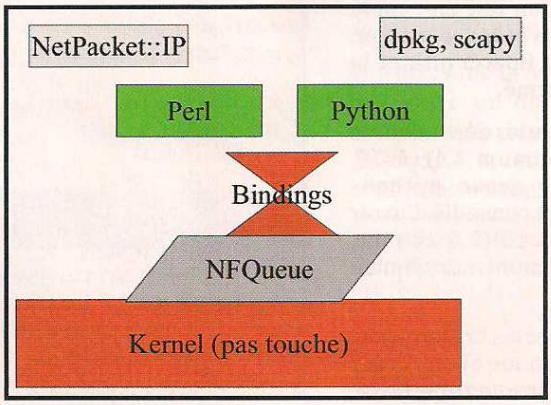
Le principal de la configuration se fait en précisant le nom du fichier interface SWIG, et le nom des fichiers C :

```
SET(SOURCES ../nfq.c ../nfq_common.c ../nfq_utils.c)

SWIG_ADD_MODULE(nfqueue python libnetfilter_queue.i ${SOURCES})
SWIG_LINK_LIBRARIES(nfqueue ${PYTHON_LIBRARIES} ${LIBNFQ_LINK_FLAGS})
```

3.2 Fonctionnement

`nfqueue_bindings` n'a pas pour vocation de copier l'API de `libnetfilter_queue`, donc au lieu d'exposer toutes les fonctions de l'API, seul un objet est directement visible : `queue`. Cette objet contient les méthodes nécessaires pour créer une file et s'enregistrer auprès du noyau.

La récupération des données des paquets se fera également sous forme de données brutes, ce qui permettra soit d'accéder directement aux données, soit d'utiliser un module standard (par exemple `NetPacket::IP` pour Perl ou `dpkt` et `scapy` pour Python).



Cependant, un problème apparaît assez vite : `libnetfilter_queue` fonctionne en utilisant une fonction de rappel (*callback*) en C, alors que nous ne pouvons que fournir une fonction de haut niveau (Python). Et là, c'est le drame : après avoir cherché un bon moment, la chose n'apparaît pas du tout simple avec SWIG. Il a donc été décidé d'utiliser directement l'API des langages de haut niveau.

La solution la plus simple (et aussi la moins propre) est de créer une fonction qui permettra de stocker une référence sur la fonction Python, dans un pointeur générique (`void *`). Ce n'est pas très élégant, mais cela simplifiera les choses pour la suite.

```
int set_callback(PyObject *pyfunc)
{
        self->_cb = (void*)pyfunc;
        Py_INCREF(pyfunc);
        return 0;
}
```

La fonction de callback nfqueue a une signature imposée par nfqueue. Le contenu de cette fonction devra effectuer trois actions : extraire les données, construire un objet Python, appeler la fonction.

1. Extraction des données (code simplifié) :

```
int  swig_nfq_callback(struct nfq_q_handle *qh, struct nfgenmsg *nfmsg,
                       struct nfq_data *nfad, void *data)
{
  struct nfqnl_msg_packet_hdr *ph;
  int id;
  char *payload_data;
  int payload_len;

  ph = nfq_get_msg_packet_hdr(nfad);
  id = ntohl(ph->packet_id);
  payload_len = nfq_get_payload(nfad, &payload_data);
```

2. Construction de l'objet Python :

```
  func = (PyObject *) data;
  p = malloc(sizeof(struct payload));
  p->data = payload_data;
  p->len = payload_len;
  p->id = id;
  payload_obj = SWIG_NewPointerObj((void*) p, SWIGTYPE_p_payload, 1);
  arglist = Py_BuildValue("(i,O)", 42, payload_obj);
```

La dernière ligne correspond à la conversion d'une structure C en son objet Python « proxifié » par SWIG. Pour comprendre le fonctionnement, il suffit de regarder le code source du fichier C généré par SWIG. Le chiffre 42, lui, ne sert à rien.

Note

L'utilisation de l'API Python (en particulier l'appel de fonctions) n'est pas *thread-safe*. Il faut donc entourer le code des macros `SWIG_PYTHON_THREAD_BEGIN_ALLOW` et `SWIG_PYTHON_THREAD_END_ALLOW`.

3. Appel de la fonction

```
result = PyEval_CallObject(func,arglist);
result = PyErr_Occurred();
if (result) {
  /* callback failure */
  PyErr_Print();
}
```

Tout fonctionne. Enfin, presque : on a oublié d'appliquer le verdict ! Pas de panique, ce point est assez simple à régler, il suffit d'appeler la fonction **nfq_set_verdict**. Cette fonction doit être appelée pendant la fonction de callback. On l'encapsule donc dans SWIG comme toute autre fonction C. Dans le fichier d'interface SWIG, on ajoute :

```
/* taken from /usr/include/linux/netfilter.h */
#define NF_DROP   0
#define NF_ACCEPT 1
#define NF_STOLEN 2
#define NF_QUEUE  3
#define NF_REPEAT 4
#define NF_STOP   5
#define NF_MAX_VERDICT NF_STOP

%extend queue {
int set_verdict(int d) {
        return nfq_set_verdict(self->qh, self->id, d, 0, NULL);
}
};
```

C'est tout pour le code C : le reste n'est que de la gestion d'erreurs, et de l'encapsulation de fonctions standards permettant de régler les paramètres nfqueue.

3.3 Compilation

nfqueue-bindings est disponible au téléchargement sur le site http://software.inl.fr. Seuls les paquets source sont disponibles. Les paquets binaires sont pris en charge directement par les distributions Linux.

À l'heure actuelle, la version stable est la version **0.1**. Ce numéro est assez faible non pas par manque de stabilité, mais plus par manque de maturité (l'API pourrait encore changer), et par le fait que certaines fonctions C ne sont pas encore encapsulées.

Le dépôt git contient des corrections effectuées après la *release*. Il est donc parfois préférable de l'utiliser. C'est ce que nous allons faire ici.

```
$ git clone git://git.inl.fr/git/nfqueue-bindings.git
```

Cette commande récupère le dépôt git, en fait une copie locale, et extrait la dernière version dans la branche *master*. N'ayez crainte, tout cela est très rapide (merci git), et le suffixe **.git** est automatiquement supprimé.

Avant de pouvoir compiler, assurez-vous que les dépendances sont bien installées : **CMake** (version minimum 2.4), **SWIG**, **pkg-config**, **libnfnetlink**, **libnetfilter_queue**, **python-dev** et **libperl-dev**. Il est aussi fortement conseillé d'avoir un noyau pas trop ancien, minimum 2.6.18 (2.6.20 pour pouvoir régler la taille de la file de communication entre le noyau et l'espace utilisateur).

Pour ne pas polluer le répertoire source avec les fichiers issus de la configuration et de la compilation, nous allons créer un sous-répertoire **build**, et compiler dans ce répertoire.

Si quelque chose ne fonctionne pas comme voulu, il suffira de l'effacer et de recommencer. Commençons par invoquer CMake, en précisant un répertoire d'installation.

```
$ cd nfqueue-bindings
$ mkdir build
$ cd build
$ cmake -DCMAKE_INSTALL_PREFIX=/usr/local ..
```

Si tout se passe bien, un message doit vous indiquer que la configuration est finie, et que les fichiers Makefile ont été générés. On continue ensuite par la très classique compilation :

```
$ make
$ [sudo] make install
```

3.4 Exemple simple

Nous disposons donc maintenant de deux modules (Perl ou Python). Si vous avez installé ces modules dans des répertoires non standards, ils ne sont pas utilisables directement : l'interpréteur se plaint qu'il ne peut pas charger le module.

Pour Python, la solution est soit de modifier la variable d'environnement **PYTHONPATH**, soit d'ajouter quelques lignes au début du programme :

```
from sys import path
path.append('/path/to/python/bindings')
import nfqueue
```

Pour Perl, le plus simple est d'ajouter le chemin vers le module dans la variable **@INC**, mais en faisant attention de le faire avant l'exécution du reste du code (donc dans la directive **BEGIN**) :

```
BEGIN {
  push @INC,"/path/to/perl/bindings";
};

use nfqueue;
```

L'exemple qui suit va effectuer un décodage très basique des paquets, en utilisant le module Python **dpkt** (**[4]**). La fonction de callback doit extraire les données brutes du *payload*, puis effectuer une conversion vers un objet **dpkt.ip.IP**. Une connaissance du format des paquets IP peut être utile (sans pour autant avoir besoin de lire les RFC), et, dans tous les cas, il ne faut pas oublier que certaines informations sont dépendantes du protocole. Par exemple, la notion de port de destination n'a de sens que pour TCP ou UDP.

```
def cb(i,payload):
  print "python callback called !", i

  print "payload len ", payload.get_length()
  data = payload.get_data()
  pkt = ip.IP(data)
  print "proto:", pkt.p
  print "source: %s" % inet_ntoa(pkt.src)
  print "dest: %s" % inet_ntoa(pkt.dst)
  if pkt.p == ip.IP_PROTO_TCP:
    print "  sport: %s" % pkt.tcp.sport
    print "  dport: %s" % pkt.tcp.dport
  payload.set_verdict(nfqueue.NF_DROP)

  return 1
```

Dans le cas où la fonction de callback ne renvoie pas de décision (**ACCEPT** ou **DROP**), un verdict par défaut est appliqué (**ACCEPT**). Si la fonction de verdict est appelée plusieurs fois, c'est le premier appel qui gagne. Cela permet également de renvoyer la décision le plus rapidement possible, et de pouvoir continuer le traitement si besoin est.

La création de la file **NFQUEUE** permet de préciser à quelle file on souhaite se connecter. Cette file est identifiée par un numéro, et doit correspondre au paramètre **--queue-num** de la règle iptables (nous reviendrons sur ce point après). Il faut également positionner la fonction de callback avant d'appeler la boucle d'évènements, qui est une boucle infinie, jusqu'à une interruption (clavier par exemple) ou une erreur.

```
q = nfqueue.queue()
q.open()
q.bind();
q.set_callback(cb)
q.create_queue(0)
try:
  q.try_run()
except KeyboardInterrupt, e:
  print "interrupted"
q.unbind()
q.close()
```

La fonction **fast_open** a récemment été ajoutée pour effectuer toutes les opérations d'initialisation :

```
q = nfqueue.queue()
q.set_callback(cb)
q.fast_open(0)
```

Il reste à ajouter une règle avec **iptables** pour envoyer les paquets (suivant un critère, inutile de tout envoyer) vers la cible **NFQUEUE**.

```
# iptables -A OUTPUT -p tcp --destination 192.168.33.181 -m state \
--state ESTABLISHED -j ACCEPT
# iptables -A OUTPUT -p tcp --destination 192.168.33.181 --dport 80 -m \
state --state NEW --syn -j NFQUEUE --queue-num 0
```

On lance notre programme de test en utilisant la commande **sudo**. En effet, l'opération de connexion (*bind*) demande des privilèges administrateur, et on teste l'envoi d'un paquet (*telnet* ou *nc* feront l'affaire).

```
$ sudo ./examples/example.py
open
bind
setting callback
creating queue
trying to run
setting copy_packet mode
python callback called ! 42
payload len  60
proto: 6
source: 192.168.33.145
dest: 192.168.33.181
  sport: 38247
  dport: 80
python callback call: 0 sec 552 usec
```

D'autres exemples sont disponibles dans le répertoire **examples** des sources.

4 Applications

4.1 Weatherwall

Maintenant que nos *bindings* sont prêts, nous allons pouvoir les utiliser. La première application sera une illustration des possibilités qui s'offrent par le fait de ne pas être en mode noyau : nous allons créer un pare-feu dont le critère de filtrage sera non pas l'adresse source ou destination, mais plutôt la **météo** de la ville source ou destination.

Tout d'abord, nous allons utiliser les bindings pour récupérer l'adresse de destination des paquets. Grâce aux bindings (et aux modules Perl), ce point est réglé assez rapidement :

```
my $ip_obj = NetPacket::IP->decode($payload->get_data());
my $dst_ip = $ip_obj->{dest_ip};
```

L'étape suivante consiste à récupérer le nom de la ville, et le pays correspondant à l'adresse IP de destination. Nous allons utiliser pour cela l'extension Perl **Geo::IP**. Avant de pouvoir l'utiliser, il faut télécharger une base contenant des informations sur tous les blocs IP.

```
wget http://www.maxmind.com/download/geoip/database/GeoLiteCity.dat.gz
gunzip GeoLiteCity.dat.gz
```

On utilise donc **Geo::IP** pour récupérer la ville et le pays. Cette récupération peut parfois échouer (toutes les adresses IP du monde ne sont pas enregistrées ou encore l'adresse est celle d'un réseau privé, etc.), donc il ne faut pas oublier la gestion des erreurs.

```
use Geo::IP;
my $record = get_city($dst_ip);
$record or return;
my $city = $record->city . ", " . $record->country_name;
```

Le nom du pays est mis dans un format **"Ville, Pays"** qui sera ensuite utilisé pour récupérer la météo.

La récupération de la météo est à peine plus compliquée. La difficulté se situe surtout dans le choix des modules : il en existe de nombreux (basés sur METAR ou encore **weather.com**). Certains de ces modules demandent de s'enregistrer (ce qui est parfois payant). Ils ont donc été écartés. Après une recherche sur le net, nous choisissons donc d'utiliser **Weather::Underground** (**[5]**), qui semble fonctionner assez bien.

En utilisant la ville et le pays récupéré juste avant, on récupère un objet contenant les paramètres météo : température, humidité, et conditions actuelles (pluie, etc.). Là encore, la récupération de la météo peut échouer, donc il faut faire attention.

```
use Weather::Underground;
my $weather = get_weather($city);
if (not $weather) {
  print "unable to fetch weather for $city\n";
  return;
}
```

Note

La récupération de la météo se fait en ouvrant une connexion. Attention de ne pas renvoyer cette connexion à **NFQUEUE**, sinon votre programme ne sera qu'une nouvelle sorte de boucle infinie !

Nous y sommes presque ! L'objet contient toutes les données qui nous intéressent, il est facile de les lire :

```
print "Place: ", $weather->{place}, "\n";
print "Conditions: ", $weather->{conditions}, "\n";
print "Temperature: ", $weather->{temperature_celsius}, "\n";
print "Humidity: ", $weather->{humidity}, "\n";
```

Et c'est là où la syntaxe Perl est appréciable :

```
if ($weather->{conditions} =~ m/[Rr]ain/)
  and $city neq "Rennes")
{
  $payload->set_verdict($nfqueue::NF_DROP);
  return;
}
$payload->set_verdict($nfqueue::NF_ACCEPT);
```

4.2 Personal Firewall

Vous en avez assez que Netfilter filtre des paquets sans rien vous dire ? Vous souhaitez pouvoir effectuer un contrôle plus poussé ? Pas de problème, nous allons créer une application permettant de demander à l'utilisateur de valider les paquets !

On utilise le même système que précédemment, en enregistrant une fonction de rappel sur la réception d'un paquet. La première étape consiste à extraire les valeurs qui nous intéressent du paquet.

```
text_dst = None
if pkt.p == ip.IP_PROTO_TCP:
        print "  sport: %s" % pkt.tcp.sport
        print "  dport: %s" % pkt.tcp.dport
        text = "%s:%s => %s:%s" % (inet_ntoa(pkt.src),pkt.tcp.sport,
inet_ntoa(pkt.dst), pkt.tcp.dport)
        text_dst = "%s:%s" % (inet_ntoa(pkt.dst), pkt.tcp.dport)
```

Nous disposons maintenant d'une variable **text** qui contient le quadruplet TCP de la connexion, par exemple **192.168.1.12:59568 => 192.168.1.1:80**.

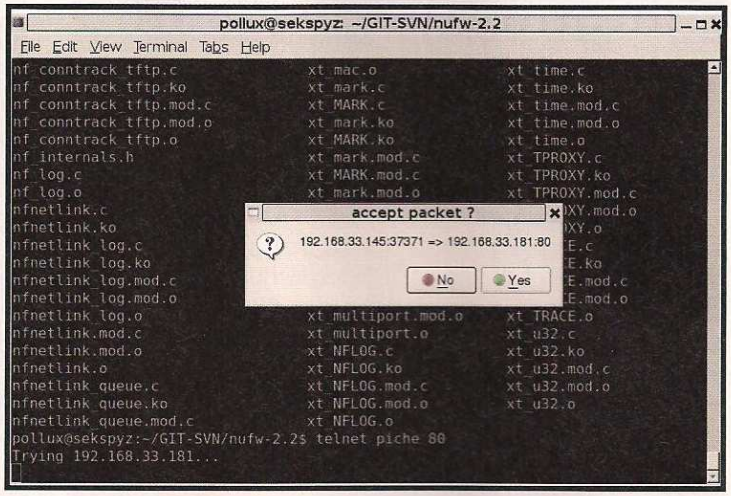
Il ne reste plus qu'à présenter ce texte à l'utilisateur, en lui demandant de valider (ou pas) la connexion. Pour des raisons d'heure tardive et d'urgence sur les délais de réalisation, l'implémentation a été faite avec la solution la plus rapide, en Python-Qt :

```
reply = QMessageBox.question(None,'accept packet ?',text,QMessageBox.
Yes, QMessageBox.No)

if reply == QMessageBox.Yes:
        decision = nfqueue.NF_ACCEPT
        payload.set_verdict(nfqueue.NF_ACCEPT)
else:
        decision = nfqueue.NF_DROP

payload.set_verdict(decision)
```

Pour pouvoir tester, il faut ajouter une règle iptables qui envoie les paquets à la cible **NFQUEUE** (mais on ne veut que les paquets SYN). On ajoute donc deux règles, une pour les connexions établies, et une pour les paquets SYN.

```
# iptables -A OUTPUT -p tcp --destination 192.168.33.181 -m state
--state ESTABLISHED -j ACCEPT
# iptables -A OUTPUT -p tcp --destination 192.168.33.181 --dport 80 -m
state --state NEW --syn -j NFQUEUE
```

Voila ! On dispose maintenant d'une application presque aussi utile et professionnelle que ses équivalents sous un système propriétaire.



Pour accélérer un peu les performances (et ne pas trop ennuyer l'utilisateur), le programme a été amélioré pour ne filtrer que les paquets SYN d'un part, et pour mettre en cache les décisions d'autre part. Cette partie est extrêmement simple : avant de poser la question, on regarde si le quadruplet est déjà dans le cache (pour faire sale, on stocke directement le texte concernant l'adresse et le port de destination dans le cache), et on applique la décision si c'est le cas :

```
if text_dst and cache_decisions.has_key(text_dst):
        print "shortcut: %s (%d)" % (text_dst,cache_decisions[text_
dst])
        return payload.set_verdict(cache_decisions[text_dst])
```

Il ne reste plus qu'à remplir le cache, en ajoutant après l'application de la décision :

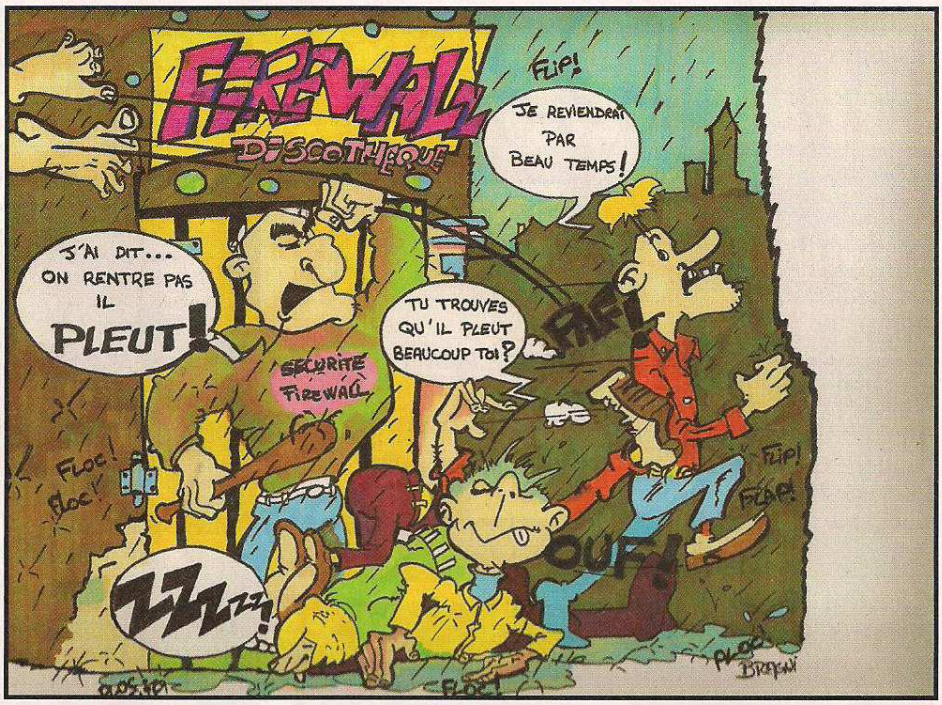
```
cache_decisions[text_dst] = decision
```

4.3 Filtrage du contenu des paquets

Maintenant que nous savons décoder les paquets, il serait intéressant de pouvoir lire les données, par exemple le contenu des requêtes HTTP. Un peu de connaissance TCP/IP s'impose : une connexion TCP est composée de paquets de contrôle (qui servent à assurer que les données ont effectivement été transmises) et d'un seul paquet qui contiendra réellement les données. La différence entre ces paquets se fait grâce à l'utilisation de drapeaux (*flags*) : SYN, ACK, RST, FIN et PSH. Chaque paquet envoyé est marqué d'un ou plusieurs drapeaux, par exemple le premier paquet d'un connexion est forcément un SYN.

Le paquet contenant les données est marqué du flag PSH. Il est important de restreindre autant que possible le filtrage au niveau d'iptables pour n'envoyer en espace utilisateur que le minimum de données possibles : la copie coûte cher (en temps), et Linux n'est pas forcément idéal sur ce point (pas de *zero-copy*, donc avant d'arriver en espace utilisateur, un paquet est copié plusieurs fois).

L'exemple suivant est fait en Perl (pour changer). On procède, comme précédemment, mais cette fois on doit d'abord décoder le couche IP du paquet, puis la couche TCP pour accéder aux données. Nous allons utiliser **nfqueue-bindings** pour écrire un programme qui vérifie qu'une connexion sur le port 80 contient vraiment des données correspondant au protocole HTTP (de manière assez rudimentaire). On adapte la fonction de callback :




```
if($ip_obj->{proto} == IP_PROTO_TCP) {
  # decode the TCP header
  my $tcp_obj = NetPacket::TCP->decode($ip_obj->{data});

  if ($tcp_obj->{flags} & NetPacket::TCP::PSH) {
    print "TCP data:\n";
    print "*" x 50 . "\n";
    print $tcp_obj->{data};
    print "*" x 50 . "\n";
    if ($tcp_obj->{dest_port} == 80) {
      _check_http($tcp_obj->{data}) or return $payload->set_
verdict($nfqueue::NF_DROP);
    }
  }
}
```

Il reste à écrire la fonction **_check_http**. Cette fonction est censée s'assurer que les données vérifient bien le protocole http. Donc, il faudrait lire la RFC correspondante et effectuer un nombre assez grand de vérifications. À titre d'exemple, on va uniquement vérifier que les données contiennent deux paramètres classiques dans les requêtes HTTP : une ligne commençant par **GET** (le nom de l'objet demandé) et une autre ligne commençant par **User-Agent** (information contenant le nom du navigateur). Ces lignes peuvent apparaître dans le désordre. Donc, nous allons faire au plus simple : une expression rationnelle que l'on applique sur l'intégralité des données, en mode multi-ligne.

```
my @http_checks = (
  "^GET ",
  "^User-Agent",
);

sub _check_http
{
  my $data = shift;

  foreach my $check (@http_checks) {
    return 0 unless ($data =~ /$check/moi);
  }

  return 1;
}
```

Si une au moins des vérifications échoue, la fonction renverra 0 et le paquet sera rejeté.

4.4 Modification de paquets

Vous en avez assez que vos utilisateurs passent leur temps à faire de la messagerie instantanée ou de l'IRC ? On va les aider à se faire des amis en modifiant leurs paquets dynamiquement !

La modification de paquets est une fonctionnalité moins connue de **libnetfilter_queue**. Puisque nous disposons du paquet, il serait bon de pouvoir le modifier, et le réinjecter. Eh bien, c'est possible ! Après avoir modifié les différents champs, il va falloir ré-assembler chaque couche du protocole en recalculant la somme de contrôle (*checksum*), en commençant par la couche TCP, puis la couche IP, et de le renvoyer au noyau via la fonction **nfq_set_verdict**. La fonction suivante est déclarée dans le fichier d'interface SWIG :

```
int set_verdict_modified(int d, char *new_payload, int new_len) {
        return nfq_set_verdict(self->qh, self->id, d, new_len, new_payload);
}
```

Tout d'abord, on ne souhaite filtrer que les connexions TCP, et, dans ces connexions, uniquement les paquets qui contiennent des données (pas les acquittements). On identifie à nouveau ces paquets grâce au flag TCP PSH (*Push*). La partie ré-assemblage du paquet et calcul du checksum est entièrement gérée par le module **NetPacket::IP**. Il suffit donc d'appeler la fonction **encode**.

```
my $ip_obj = NetPacket::IP->decode($payload->get_data());
if($ip_obj->{proto} == IP_PROTO_TCP) {

  # Desassemblage du paquet
  my $tcp_obj = NetPacket::TCP->decode($ip_obj->{data});
```

```
if ($tcp_obj->{flags} & NetPacket::TCP::PSH &&
  length($tcp_obj->{data})) {

 # Modification
 $tcp_obj->{data} =~ s/love/hate/m;
 print "**********\n";
 print $tcp_obj->{data};
 print "**********\n";

 # Re-assemblage
 $ip_obj->{data} = $tcp_obj->encode($ip_obj);
 my $modified_payload = $ip_obj->encode();

 # Envoi
 $payload->set_verdict_modified($nfqueue::NF_ACCEPT,$modified_
payload,length($modified_payload));
}
```

La règle **iptables** doit être modifiée pour ne filtrer que les paquets **PSH** :

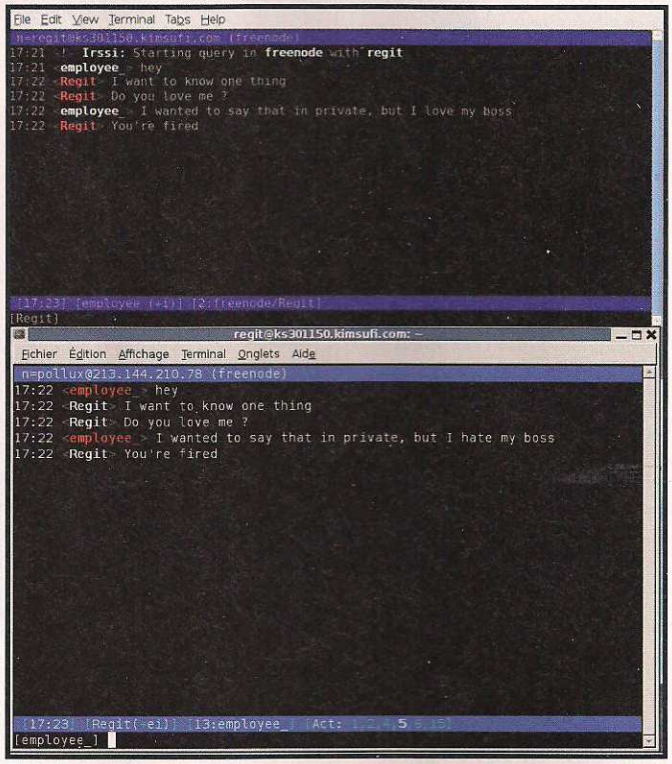
```
iptables -A OUTPUT -p tcp --destination 192.168.33.181 --tcp-flags psh \
psh -j NFQUEUE --queue-num 0
```

En regardant la sortie (assez verbeuse) du programme, on voit que la substitution a bien été faite :

```
TCP src_port: 33242
TCP dst_port: 6667
TCP flags   : 24
TCP data    : PRIVMSG regit :I wanted to say that in private, but I love my boss

**********
PRIVMSG regit :I wanted to say that in private, but I hate my boss
**********
192.168.xx.xxx => 213.219.xxx.xx 6
TCP src_port: 33242
TCP dst_port: 6667
```

```
data length: 120
ret: 157
perl callback call: 0 sec 1693 usec
```

Il faut faire attention cependant aux performances. Le fait de ré-assembler les paquets et de les envoyer prend environ trois fois plus de temps qu'un simple **ACCEPT**.



5 Conclusion

Voila, c'est tout pour cette introduction à **nfqueue-bindings**. De nombreuses autres applications sont envisageables. À vous d'inventer la vôtre ! Par exemple, **nfqueue-bindings** permet également de modifier la marque associée à une connexion, peut utiliser la cible **NF_REPEAT** pour renvoyer le paquet au début de la chaîne de filtrage, et bien d'autres choses encore.

nfqueue-bindings est disponible sur http://software.inl.fr/trac/wiki/nfqueue-bindings, et est packagé pour Debian par votre serviteur.

Les évolutions prévues sur **nfqueue-bindings** concernent surtout l'ajout de fonctions permettant d'accéder aux métadonnées d'un paquet (interface d'entrée et de sortie, etc.), à l'ajout de fonctions qui ne sont actuellement pas accessibles, à l'amélioration des performances, et à une meilleure gestion des erreurs. L'ajout du support d'autres langages est également envisagé.

Un projet similaire **nflog-bindings** **[6]** a été créé, pour la cible **NFLOG** (au lieu de **NFQUEUE**), ce qui permet d'avoir accès aux mêmes fonctionnalités sauf les fonctions de modification et de verdict sur les paquets. L'intérêt est évidemment de ne pas bloquer le paquet pendant que la fonction de callback est appelée.

Enfin, je tiens à remercier Éric pour l'idée originale du pare-feu météo :)

Références

- **[1]** nfqueue-bindings : http://software.inl.fr/trac/wiki/nfqueue-bindings
- **[2]** La présentation faite au SSTIC : http://www.wzdftpd.net/downloads/rump-sstic-2008-nfqueue-bindings.pdf
- **[3]** SWIG : http://www.swig.org
- **[4]** dpkt : http://code.google.com/p/dpkt/
- **[5]** Weather::Underground : http://search.cpan.org/~mnaguib/Weather-Underground-3.02/Underground.pm
- **[6]** NFLog-bindings : http://software.inl.fr/trac/wiki/nflog-bindings

Auteur : Pierre Chifflier

Utilisateur GNU/Linux depuis 1997. Responsable de l'équipe technique d'INL. Développeur Debian, contributeur NuFW.