OpenBSD

Langues : [fr] [de] [en] [es] [id] [ja] [pl]

[Index de La FAQ] [Suivant : Principes de Base]

PF : Le Filtre de Paquets d'OpenBSD

Table des Matières

Packet Filter (que nous appelerons désormais PF) est le système utilisé par OpenBSD pour filtrer le trafic TCP/IP et effectuer des opérations de Traduction d'Adresses IP (NAT). PF est aussi capable de normaliser, de conditionner le trafic TCP/IP, et de fournir des services de contrôle de bande passante et gestion des priorités des paquets. PF fait partie du noyau GENERIC d'OpenBSD depuis la version 3.0. Les précédentes versions d'OpenBSD utilisaient un ensemble logiciel pare-feu/NAT différent qui n'est plus supporté.

PF fût initialement développé par Daniel Hartmeier. Il est maintenant et développé et maintenu par Daniel et le reste de l'équipe OpenBSD.

L'ensemble des documents listés dans la table des matières est destiné à servir d'introduction générale au pare-feu PF tel qu'il est

livré avec OpenBSD. C'est un supplément aux pages du manuel. Il n'a pas vocation de les remplacer. Toutes les fonctionnalités majeures de PF sont couvertes. Cependant, pour une couverture complète et approfondie de ce que peut faire PF, nous vous conseillons de lire d'abord la page du manuel pf(4).

De même que le reste de la FAQ, ce document est orienté vers les utilisateurs d' OpenBSD 3.4 (la version la plus récente disponible à ce jour). Etant donné que PF continue sans cesse de s'améliorer, il existe un certain nombre de changements et d'améliorations entre la version fournie avec 3.4-release et la version fournie avec OpenBSD-current. Nous vous recommandons de lire les pages du manuel de la version d'OpenBSD que vous utilisez.

[Index de La FAQ] [Suivant : Principes de Base]


www@openbsd.org

Originally [OpenBSD: index.html,v 1.14]
$Translation: index.html,v 1.8 2004/02/18 20:55:13 saad Exp $
$OpenBSD: index.html,v 1.8 2004/02/19 11:43:44 jufi Exp $

[Index] [Suivant : Listes et Macros]

PF : Principes de Base

Table des Matières

Activation

Pour activer PF, éditez le fichier `/etc/rc.conf` et modifiez la ligne PF comme suit :

```
pf=YES
```

Redémarrez votre système pour que les modifications soient prises en compte.

Vous pouvez aussi activer et désactiver PF en utilisant le programme pfctl(8) :

```
# pfctl -e
# pfctl -d
```

La première commande active PF (l'option "-e" correspond à "enable"). La seconde commande désactive PF (l'option "-d" correspond à "disable"). Ces commandes ne causent pas le chargement d'une base de règles. Cette dernière doit être chargée séparemment avant ou après l'activation de PF.

Configuration

PF lit les règles de configuration à partir du fichier `/etc/pf.conf` au démarrage de la machine. Ces règles sont chargées au démarrage par les scripts rc. Il est à noter que `/etc/pf.conf` est le fichier par défaut chargé par ces scripts. C'est un fichier texte chargé et interprété par pfctl(8) puis inséré dans pf(4). Pour certaines applications, d'autres bases de règles peuvent être chargées à partir de fichiers différents durant la phase de démarrage. Comme n'importe quelle application Unix bien conçue, PF offre une grande flexibilité.

Le fichier `pf.conf` est composé de sept parties :

- **Macros :** Variables définies par l'utilisateur pouvant contenir des adresses IP, des noms d'interfaces, etc.
- **Tables :** Structures conçues pour contenir des listes d'adresses IP.
- **Options :** Diverses options pour contrôler le fonctionnement de PF.
- **Scrub :** Retraitement des paquets pour les normaliser et les défragmenter.

- **Gestion de La Bande Passante :** Gère la bande passante et la priorité des paquets.
- **Traduction ("Translation") :** Contrôle la Traduction d'Adresses IP et la redirection de paquets.
- **Règles de Filtrage :** Permet le filtrage séléctif ou le bloquage des paquets lors de leur traversée de n'importe quelle interface.

A l'exception des macros et des tables, chaque section doit apparaître dans l'ordre précité dans le fichier de configuration. Cependant toutes les sections ne sont pas obligatoires.

Les lignes vides sont ignorées et les lignes commençant par # sont considérées comme des commentaires.

Contrôle

Après le démarrage, PF peut être contrôlé grâce au programme pfctl(8). Voici des exemples de commandes :

```
# pfctl -f /etc/pf.conf     Charge le fichier pf.conf
# pfctl -nf /etc/pf.conf    Analyse le fichier mais ne le charge pas
# pfctl -Nf /etc/pf.conf    Charge uniquement les règles NAT à partir du fichier
# pfctl -Rf /etc/pf.conf    Charge uniquement les règles de filtrage à partir du fichier

# pfctl -sn                 Affiche les règles NAT actuelles
# pfctl -sr                 Affiche les règles de filtrage actuelles
# pfctl -ss                 Affiche la table d'état actuelle
# pfctl -si                 Affiche les statistiques et les compteurs de filtrage
# pfctl -sa                 Affiche TOUT ce qu'on est capable d'afficher
```

Pour une liste complète de commandes, veuillez lire la page du manuel pfctl(8).

[Index] [Suivant : Listes et Macros]

---

www@openbsd.org
Originally [OpenBSD: config.html,v 1.10]
$Translation: config.html,v 1.4 2004/02/19 20:31:29 saad Exp $
$OpenBSD: config.html,v 1.4 2004/02/20 06:33:11 jufi Exp $

OpenBSD

[Précédent : Principes de Base] [Index] [Suivant : Tables]

PF : Listes et Macros

Table des Matières

Listes

Une liste permet d'utiliser plusieurs critères similaires dans une même règle PF. Par exemple, plusieurs protocoles, plusieurs numéros de ports, plusieurs adresses IP, etc. Ainsi, au lieu d'écrire une règle de filtrage pour chaque adresse IP qui doit être bloquée, une seule règle peut être écrite en créant une liste avec les adresses IP à bloquer. Les listes sont définies en mettant des éléments entre accolades `{ }`.

Quand pfctl(8) rencontre une liste durant le chargement de la base de règles, il crée de multiples règles : une règle pour chaque élément de la liste. Par exemple :

```
block out on fxp0 from { 192.168.0.1, 10.5.32.6 } to any
```

devient :

```
block out on fxp0 from 192.168.0.1 to any
block out on fxp0 from 10.5.32.6 to any
```

Plusieurs listes peuvent être utilisées dans une même règle. Elles ne sont pas uniquement limitées aux règles de filtrage :

```
rdr on fxp0 proto tcp from any to any port { 22 80 } -> \
   192.168.0.6
block out on fxp0 proto { tcp udp } from { 192.168.0.1, \
   10.5.32.6 } to any port { ssh telnet }
```

Les virgules entre les éléments d'une même liste sont optionnelles.

Macros

Les macros sont des variables définies par l'utilisateur et pouvant contenir des adresses IP, des numéros de ports, des noms

d'interfaces, etc. Les macros peuvent réduire la complexité d'une base de règles PF et facilitent la maintenance d'une base de règles.

Les noms de macros doivent commencer par une lettre et peuvent contenir des lettres, des chiffres, et des barres de soulignement "_". Les noms des macros ne doivent pas être des noms réservés tels que `pass`, `out`, ou `queue`.

```
ext_if = "fxp0"

block in on $ext_if from any to any
```

L'exemple précédent montre comment créer une macro nommée `ext_if`. Quand une macro est appelée après avoir été créée, son nom est précédé d'un caractère `$`.

Les macros peuvent aussi représenter des listes tel que le montre l'exemple suivant :

```
friends = "{ 192.168.1.1, 10.0.2.5, 192.168.43.53 }"
```

Elles peuvent être définies de manière récursive. Vu qu'elles ne sont pas traduites dans des quotes `""`, la syntaxe suivante doit être employée :

```
host1 = "192.168.1.1"
host2 = "192.168.1.2"
all_hosts = "{" $host1 $host2 "}"
```

La macro `$all_hosts` correspond maintenant à la liste d'adresses IP {192.168.1.1 192.168.1.2}.

[Précédent : Principes de Base] [Index] [Suivant : Tables]


www@openbsd.org

Originally [OpenBSD: macros.html,v 1.9]
$Translation: macros.html,v 1.3 2004/02/19 20:31:29 saad Exp $
$OpenBSD: macros.html,v 1.3 2004/02/20 06:33:11 jufi Exp $

OpenBSD

[Précédent : Listes et Macros] [Index] [Suivant : Règles de Filtrage]

PF : Tables

Table des Matières

Introduction

Une table est une structure utilisée pour regrouper un ensemble d'adresses IPv4 et/ou IPv6. Chercher une adresse IP dans une table est une opération très rapide qui consomme moins de mémoire et de temps processeur qu'une recherche dans une liste. C'est pourquoi une table est idéale pour contenir un grand ensemble d'adresses IP vu que le temps nécessaire pour consulter le contenu d'une table de 50 000 adresses prend un tout petit peu plus de temps que la consultation d'une table de 50 adresses. Les tables peuvent être utilisées d'une des manières suivantes :

- adresse source et/ou destination dans les règles de filtrage, scrub, NAT, et redirection.
- adresse de traduction dans les règles NAT.
- adresse de redirection dans les règles de redirection.
- adresse de destination dans les options des règles de filtrage `route-to`, `reply-to`, et `dup-to`.

Une table peut être créée dans `pf.conf` ou en utilisant pfctl(8).

Configuration

Dans `pf.conf`, les tables sont créées en utilisant la directive `table`. Les attributs suivants peuvent être spécifiés pour chaque table :

- `const` - le contenu de la table ne peut être modifié une fois la table créée. Lorsque cet attribut n'est pas spécifié, pfctl(8) peut être utilisé pour ajouter ou supprimer des adresses de la table à n'importe quel moment, même lorsque le système utilise un securelevel(7) de deux ou supérieur.
- `persist` - conduit le noyau à garder la table en mémoire même lorsqu'aucune règle ne s'y réfère. Sans cet attribut, le noyau supprimera automatiquement la table lorsque la dernière règle s'y référant est supprimée.

Exemple :

```
table <goodguys> { 192.0.2.0/24 }
table <rfc1918> const { 192.168.0.0/16, 172.16.0.0/12, \
   10.0.0.0/8 }
table <spammers> persist

block in on fxp0 from { <rfc1918>, <spammers> } to any
pass  in on fxp0 from <goodguys> to any
```

Les adresses peuvent aussi être spécifiées en utilisant l'opérateur de négation (ou "not"). Voici un exemple :

```
table <goodguys> { 192.0.2.0/24, !192.0.2.5 }
```

La table `goodguys` contient alors toutes les adresses dans le réseau 192.0.2.0/24 excepté 192.0.2.5.

Notez que les noms de tables sont toujours entourées des caractères < et >.

Les tables peuvent aussi être peuplées à partir de fichiers texte contenant une liste d'adresses et de réseaux IP :

```
table <spammers> persist file "/etc/spammers"

block in on fxp0 from <spammers> to any
```

Le fichier `/etc/spammers` doit contenir une liste d'adresses IP et/ou des blocs réseau en notation CIDR, une entrée par ligne. Toute ligne commençant par # est traitée comme un commentaire. Elle est donc ignorée.

Manipulation des Tables avec `pfctl`

Les tables peuvent être manipulées à la volée en utilisant pfctl(8). Par exemple, pour ajouter des entrées à la table <spammers> précédemment créée :

```
# pfctl -t spammers -Tadd 218.70.0.0/16
```

Ceci aura aussi pour effet de créer la table <spammers> si elle n'existe pas encore. Pour afficher le contenu d'une table :

```
# pfctl -t spammers -Tshow
```

L'argument `-v` peut aussi être utilisé avec `-Tshow` pour afficher des statistiques pour chaque entrée de la table. Pour supprimer des adresses de la table :

```
# pfctl -t spammers -Tdelete 218.70.0.0/16
```

Pour plus d'informations concernant la manipulation des tables avec `pfctl`, veuillez consulter pfctl(8).

Spécification des Adresses

Les hôtes peuvent être spécifiés par adresse IP ou par nom. Lorsque le nom est résolu en adresse IP, toutes les adresses IPv6 et IPv4 résultantes sont placées dans la table. Les adresses IP peuvent aussi être saisies dans la table en spécifiant le nom d'une

interface valide ou le mot-clé `self` dans quel cas toutes les adresses IP assignées à l'interface ou aux interfaces configurées sur le pare- feu (si le mot-clé `self` est utilisé) seront insérées dans la table.

Recherche des Adresses Correspondantes

La recherche d'une adresse dans une table retournera l'entrée la plus proche de cette adresse. Ceci permet la création de tables telles que :

```
table <goodguys> { 172.16.0.0/16, !172.16.1.0/24, 172.16.1.100 }

block in on dc0 all
pass  in on dc0 from <goodguys> to any
```

Tout paquet arrivant vers `dc0` verra son adresse source recherchée dans la table `<goodguys>`:

- 172.16.50.5 - l'entrée la plus proche est 172.16.0.0/16; le paquet a une entrée correspondante dans la table donc il passe
- 172.16.1.25 - l'entrée la plus proche est !172.16.1.0/24; le paquet a une entrée correspondante mais celle-ci est en négation (l'opérateur "!" est utilisé); le paquet n'a donc aucune entrée correspondante et sera bloqué
- 172.16.1.100 - l'entrée la plus proche est 172.16.1.100; le paquet a une entrée correspondante dans la table donc il passe
- 10.1.4.55 - n'a aucune adresse correspondante dans la table. Il sera bloqué

[Précédent : Listes et Macros] [Index] [Suivant : Filtrage de Paquets]


www@openbsd.org

Originally [OpenBSD: tables.html,v 1.9]
$Translation: tables.html,v 1.4 2004/02/19 20:31:29 saad Exp $
$OpenBSD: tables.html,v 1.4 2004/02/20 06:33:11 jufi Exp $

[Previous: Tables] [Contents] [Next: Network Address Translation]

PF: Packet Filtering

Table of Contents

Introduction

Packet filtering is the selective passing or blocking of data packets as they pass through a network interface. The criteria that pf(4) uses when inspecting packets is based on the Layer 3 (IPv4 and IPv6) and Layer 4 (TCP, UDP, ICMP, and ICMPv6) headers. The most often used criteria are source and destination address, source and destination port, and protocol.

Filter rules specify the criteria that a packet must match and the resulting action, either block or pass, that is taken when a match is found. Filter rules are evaluated in sequential order, first to last. Unless the packet matches a rule containing the `quick` keyword, the packet will be evaluated against *all* filter rules before the final action is taken. The last rule to match is the "winner" and will dictate what action to take on the packet. There is an implicit `pass all` at the beginning of a filtering ruleset meaning that if a packet does not match any filter rule the resulting action will be `pass`.

Rule Syntax

The general, *highly simplified* syntax for filter rules is:

```
action direction [log] [quick] on interface [af] [proto protocol] \
   from src_addr [port src_port] to dst_addr [port dst_port] \
   [tcp_flags] [state]
```

action
: The action to be taken for matching packets, either `pass` or `block`. The `pass` action will pass the packet back to the kernel for further processing while the `block` action will react based on the setting of the `block-policy` option. The default reaction may be overridden by specifying either `block drop` or `block return`.

direction
: The direction the packet is moving on an interface, either `in` or `out`.

`log`
: Specifies that the packet should be logged via pflogd(8). If the rule specifies the `keep state`, `modulate state`, or `synproxy state` option, then only the packet which establishes the state is logged. To log all packets regardless, use `log-all`.

`quick`
: If a packet matches a rule specifying `quick`, then that rule is considered the last matching rule and the specified *`action`* is taken.

`interface`
: The name of the network interface that the packet is moving through.

af
: The address family of the packet, either `inet` for IPv4 or `inet6` for IPv6. PF is usually able to determine this parameter based on the source and/or destination address(es).

protocol
: The Layer 4 protocol of the packet:
  - `tcp`
  - `udp`
  - `icmp`
  - `icmp6`
  - A valid protocol name from `/etc/protocols`
  - A protocol number between 0 and 255
  - A set of protocols using a list.

src_addr, *dst_addr*
: The source/destination address in the IP header. Addresses can be specified as:
  - A single IPv4 or IPv6 address.
  - A CIDR network block.
  - A fully qualified domain name that will be resolved via DNS when the ruleset is loaded. All resulting IP addresses will be substituted into the rule.
  - The name of a network interface. Any IP addresses assigned to the interface will be substituted into the rule.
  - The name of a network interface followed by */netmask* (i.e., `/24`). Each IP address on the interface is combined with the netmask to form a CIDR network block which is substituted into the rule.
  - The name of a network interface in parentheses `( )`. This tells PF to update the rule if the IP address(es) on the named interface change. This is useful on an interface that gets its IP address via DHCP or dial-up as the ruleset doesn't have to be reloaded each time the address changes.
  - The name of a network interface followed by the `:network` or `:broadcast` keywords. The resulting CIDR network (i.e., 192.168.0.0/24) or broadcast address (i.e., 192.168.0.255) will be substituted into the rule when the ruleset is loaded.
  - A table.
  - Any of the above but negated using the `!` ("not") modifier.
  - A set of addresses using a list.
  - The keyword `any` meaning all addresses
  - The keyword `all` which is short for `from any to any`.

src_port, *dst_port*
: The source/destination port in the Layer 4 packet header. Ports can be specified as:
  - A number between 1 and 65535
  - A valid service name from `/etc/services`
  - A set of ports using a list
  - A range:
    - `!=` (not equal)
    - `<` (less than)
    - `>` (greater than)
    - `<=` (less than or equal)
    - `>=` (greater than or equal)
    - `><` (range)
    - `<>` (inverse range)

    The last two are binary operators (they take two arguments) and do not include the arguments in the range.

tcp_flags
: Specifies the flags that must be set in the TCP header when using `proto tcp`. Flags are specified as `flags` *`check/mask`*. For example: `flags S/SA` - this instructs PF to only look at the S and A (SYN and ACK) flags and to match if the SYN flag is "on".

state
: Specifies whether state information is kept on packets matching this rule.
  - `keep state` - works with TCP, UDP, and ICMP.
  - `modulate state` - works only with TCP. PF will generate strong Initial Sequence Numbers (ISNs) for packets matching this rule.
  - `synproxy state` - proxies incoming TCP connections to help protect servers from spoofed TCP SYN floods. This option includes the functionality of `keep state` and `modulate state`.

Default Deny

The recommended practice when setting up a firewall is to take a "default deny" approach. That is, to deny *everything* and then selectively allow certain traffic through the firewall. This approach is recommended because it errs on the side of caution and also makes writing a ruleset easier.

To create a default deny filter policy, the first two filter rules should be:

```
block in  all
block out all
```

This will block all traffic on all interfaces in either direction from anywhere to anywhere.

Passing Traffic

Traffic must now be explicitly passed through the firewall or it will be dropped by the default deny policy. This is where packet criteria such as source/destination port, source/destination address, and protocol come into play. Whenever traffic is permitted to pass through the firewall the rule(s) should be written to be as restrictive as possible. This is to ensure that the intended traffic, and only the intended traffic, is permitted to pass.

Some examples:

```
# Pass traffic in on dc0 from the local network, 192.168.0.0/24,
# to the OpenBSD machine's IP address 192.168.0.1. Also, pass the
# return traffic out on dc0.
pass in  on dc0 from 192.168.0.0/24 to 192.168.0.1
pass out on dc0 from 192.168.0.1 to 192.168.0.0/24


# Pass TCP traffic in on fxp0 to the web server running on the
# OpenBSD machine. The interface name, fxp0, is used as the
# destination address so that packets will only match this rule if
# they're destined for the OpenBSD machine.
pass in on fxp0 proto tcp from any to fxp0 port www
```

The `quick` Keyword

As indicated earlier, each packet is evaluated against the filter ruleset from top to bottom. By default, the packet is marked for passage, which can be changed by any rule, and could be changed back and forth several times before the end of the filter rules. **The last matching rule "wins".** There is an exception to this: The `quick` option on a filtering rule has the effect of canceling any further rule processing and causes the specified action to be taken. Let's look at a couple examples:

Wrong:

```
block in on fxp0 proto tcp from any to any port ssh
pass  in all
```

In this case, the `block` line may be evaluated, but will never have any effect, as it is then followed by a line which will pass everything.

Better:

```
block in quick on fxp0 proto tcp from any to any port ssh
pass  in all
```

These rules are evaluated a little differently. If the `block` line is matched, due to the `quick` option, the packet will be blocked, and the rest of the ruleset will be ignored.

Keeping State

One of Packet Filter's important abilities is "keeping state" or "stateful inspection". Stateful inspection refers to PF's ability to track the state, or progress, of a network connection. By storing information about each connection in a state table, PF is able to quickly determine if a packet passing through the firewall belongs to an already established connection. If it does, it is passed through the firewall without going through ruleset evaluation.

Keeping state has many advantages including simpler rulesets and better packet filtering performance. PF is able to match packets moving in *either* direction to state table entries meaning that filter rules which pass returning traffic don't need to be written. And, since packets matching stateful connections don't go through ruleset evaluation, the time PF spends processing those packets can be greatly lessened.

When a rule has the `keep state` option, the first packet matching the rule creates a "state" between the sender and receiver. Now, not only do packets going from the sender to receiver match the state entry and bypass ruleset evaluation, but so do the reply packets from receiver to sender. For example:

```
pass out on fxp0 proto tcp from any to any keep state
```

This allows any outbound TCP traffic on the `fxp0` interface and also permits the reply traffic to pass back through the firewall. While keeping state is a nice feature, its use significantly improves the performance of your firewall as state lookups are dramatically faster than running a packet through the filter rules.

The `modulate state` option works just like `keep state` except that it only applies to TCP packets. With `modulate state`, the Initial Sequence Number (ISN) of outgoing connections is randomized. This is useful for protecting connections initiated by certain operating systems that do a poor job of

choosing ISNs.

Keep state on outgoing TCP, UDP, and ICMP packets and modulate TCP ISNs:

```
pass out on fxp0 proto tcp from any to any modulate state
pass out on fxp0 proto { udp, icmp } from any to any keep state
```

Another advantage of keeping state is that corresponding ICMP traffic will be passed through the firewall. For example, if `keep state` is specified for a TCP connection and an ICMP source-quench message referring to this TCP connection arrives, it will be matched to the appropriate state entry and passed through the firewall.

It's important to note that stateful connections are limited to the interface they were created on. This is particularly important on routers and firewalls running PF, especially when a "default deny" policy is implemented as outlined above. If a firewall is keeping state on all outgoing connections on the external interface, those packets must still be explicitly passed through the internal interface.

Note that `nat`, `binat`, and `rdr` rules implicitly create state for matching connections as long as the connection is passed by the filter ruleset.

Keeping State for UDP

One will sometimes hear it said that, "One can not create state with UDP as UDP is a stateless protocol!" While it is true that a UDP communication session does not have any concept of state (an explicit start and stop of communications), this does not have any impact on PF's ability to create state for a UDP session. In the case of protocols without "start" and "end" packets, PF simply keeps track of how long it has been since a matching packet has gone through. If the timeout is reached, the state is cleared. The timeout values can be set in the options section of the `pf.conf` file.

TCP Flags

Matching TCP packets based on flags is most often used to filter TCP packets that are attempting to open a new connection. The TCP flags and their meanings are listed here:

- **F** : FIN - Finish; end of session
- **S** : SYN - Synchronize; indicates request to start session
- **R** : RST - Reset; drop a connection
- **P** : PUSH - Push; packet is sent immediately
- **A** : ACK - Acknowledgement
- **U** : URG - Urgent
- **E** : ECE - Explicit Congestion Notification Echo
- **W** : CWR - Congestion Window Reduced

To have PF inspect the TCP flags during evaluation of a rule, the `flags` keyword is used with the following syntax:

```
flags check/mask
```

The *`mask`* part tells PF to only inspect the specified flags and the *`check`* part specifies which flag(s) must be "on" in the header for a match to occur.

```
pass in on fxp0 proto tcp from any to any port ssh flags S/SA
```

The above rule passes TCP traffic with the SYN flag set while only looking at the SYN and ACK flags. A packet with the SYN and ECE flags would match the above rule while a packet with SYN and ACK or just ACK would not.

Note: in previous versions of OpenBSD, the following syntax was supported:

```
. . . flags S
```

This is no longer true. A mask must now *always* be specified.

Flags are often used in conjunction with `keep state` rules to help control the creation of state entries:

```
pass out on fxp0 proto tcp all flags S/SA keep state
```

This would permit the creation of state on any outgoing TCP packet with the SYN flag set out of the SYN and ACK flags.

One should be careful with using flags -- understand what you are doing and why, and be careful with the advice people give as a lot of it is bad. Some people have suggested creating state "only if the SYN flag is set and no others". Such a rule would end with:

```
. . . flags S/FSRPAUEW  bad idea!!
```

The theory is, create state only on the start of the TCP session, and the session should start with a SYN flag, and no others. The problem is some sites are starting to use the ECN flag and any site using ECN that tries to connect to you would be rejected by such a rule. A much better guideline is:

```
. . . flags S/SAFR
```

While this is practical and safe, it is also unnecessary to check the FIN and RST flags if traffic is also being scrubbed. The scrubbing process will cause PF to drop any incoming packets with illegal TCP flag combinations (such as SYN and FIN or SYN and RST). It's highly recommended to always `scrub` incoming traffic:

```
scrub in on fxp0
.
.
.
pass in on fxp0 proto tcp from any to any port ssh flags S/SA \
   keep state
```

TCP SYN Proxy

Normally when a client initiates a TCP connection to a server PF will pass the handshake packets between the two endpoints as they arrive. PF has the ability, however, to proxy the handshake. With the handshake proxied, PF itself will complete the handshake with the client, initiate a handshake with the server, and then pass packets between the two. The benefit of this process is that no packets are sent to the server before the client completes the handshake. This eliminates the threat of spoofed TCP SYN floods affecting the server because a spoofed client connection will be unable to complete the handshake.

The TCP SYN proxy is enabled using the `synproxy state` keywords in filter rules. Example:

```
pass in on $ext_if proto tcp from any to $web_server port www \
   flags S/SA synproxy state
```

Here, connections to the web server will be TCP proxied by PF.

Because of the way `synproxy state` works, it also includes the same functionality as `keep state` and `modulate state`.

The SYN proxy will not work if PF is running on a bridge(4).

Blocking Spoofed Packets

Address "spoofing" is when an malicious user fakes the source IP address in packets they transmit in order to either hide their real address or to impersonate another node on the network. Once the user has spoofed their address they can launch a network attack without revealing the true source of the attack or attempt to gain access to network services that are restricted to certain IP addresses.

PF offers some protection against address spoofing through the `antispoof` keyword:

```
antispoof [log] [quick] for interface [af]
```

`log`
: Specifies that matching packets should be logged via pflogd(8).

`quick`
: If a packet matches this rule then it will be considered the "winning" rule and ruleset evaluation will stop.

`interface`
: The network interface to activate spoofing protection on. This can also be a list of interfaces.

`af`
: The address family to activate spoofing protection for, either `inet` for IPv4 or `inet6` for IPv6.

Example:

```
antispoof for fxp0 inet
```

When a ruleset is loaded, any occurrences of the `antispoof` keyword are expanded into two filter rules. Assuming that interface `fxp0` has IP address 10.0.0.1 and a subnet mask of 255.255.255.0 (i.e., a /24), the above `antispoof` rule would expand to:

```
block in on ! fxp0 inet from 10.0.0.0/24 to any
block in inet from 10.0.0.1 to any
```

These rules accomplish two things:

- Blocks all traffic coming from the 10.0.0.0/24 network that does *not* pass in through `fxp0`. Since the 10.0.0.0/24 network is on the `fxp0` interface, packets with a source address in that network block should never be seen coming in on any other interface.
- Blocks all incoming traffic from 10.0.0.1, the IP address on `fxp0`. The host machine should never send packets to itself through an external interface, so any incoming packets with a source address belonging to the machine can be considered malicious.

NOTE: The filter rules that the `antispoof` rule expands to will also block packets sent over the loopback interface to local addresses. These addresses should be passed explicitly. Example:

```
pass quick on lo0 all

antispoof for fxp0 inet
```

Usage of `antispoof` should be restricted to interfaces that have been assigned an IP address. Using `antispoof` on an interface without an IP address will result in filter rules such as:

```
block drop in on ! fxp0 inet all
block drop in inet all
```

With these rules there is a risk of blocking *all* inbound traffic on *all* interfaces.

IP Options

By default, PF blocks packets with IP options set. This can make the job more difficult for "OS fingerprinting" utilities like nmap. If you have an application that requires the passing of these packets, such as multicast or IGMP, you can use the `allow-opts` directive:

```
pass in quick on fxp0 all allow-opts
```

Filtering Ruleset Example

Below is an example of a filtering ruleset. The machine running PF is acting as a firewall between a small, internal network and the Internet. Only the filter rules are shown; queueing, `nat`, `rdr`, etc., have been left out of this example.

```
ext_if  = "fxp0"
int_if  = "dc0"
lan_net = "192.168.0.0/24"

# scrub incoming packets
scrub in all

# setup a default deny policy
block in  all
block out all

# pass traffic on the loopback interface in either direction
pass quick on lo0 all

# activate spoofing protection for the internal interface.
antispoof quick for $int_if inet

# only allow ssh connections from the local network if it's from the
# trusted computer, 192.168.0.15. use "block return" so that a TCP RST is
# sent to close blocked connections right away. use "quick" so that this
```

```
# rule is not overridden by the "pass" rules below.
block return in quick on $int_if proto tcp from ! 192.168.0.15 \
   to $int_if port ssh flags S/SA

# pass all traffic to and from the local network
pass in  on $int_if from $lan_net to any
pass out on $int_if from any to $lan_net

# pass tcp, udp, and icmp out on the external (Internet) interface.
# keep state on udp and icmp and modulate state on tcp.
pass out on $ext_if proto tcp all modulate state flags S/SA
pass out on $ext_if proto { udp, icmp } all keep state

# allow ssh connections in on the external interface as long as they're
# NOT destined for the firewall (i.e., they're destined for a machine on
# the local network). log the initial packet so that we can later tell
# who is trying to connect. use the tcp syn proxy to proxy the connection.
pass in log on $ext_if proto tcp from any to { !$ext_if, !$int_if } \
   port ssh flags S/SA synproxy state
```

[Previous: Tables] [Contents] [Next: Network Address Translation]


www@openbsd.org

$OpenBSD: filter.html,v 1.18 2004/04/27 17:56:57 henning Exp $

[Previous: Packet Filtering] [Contents] [Next: Traffic Redirection (Port Forwarding)]

PF: Network Address Translation (NAT)

Table of Contents

Introduction

Network Address Translation (NAT) is a way to map an entire network (or networks) to a single IP address. NAT is necessary when the number of IP addresses assigned to you by your Internet Service Provider is less than the total number of computers that you wish to provide Internet access for. NAT is described in RFC 1631.

NAT allows you to take advantage of the reserved address blocks described in RFC 1918. Typically, your internal network will be setup to use one or more of these network blocks. They are:

```
10.0.0.0/8        (10.0.0.0 - 10.255.255.255)
172.16.0.0/12     (172.16.0.0 - 172.31.255.255)
192.168.0.0/16    (192.168.0.0 - 192.168.255.255)
```

An OpenBSD system doing NAT will have at least two network adapters, one to the Internet, the other to your internal network. NAT will be translating requests from the internal network so they appear to all be coming from your OpenBSD NAT system.

How NAT Works

When a client on the internal network contacts a machine on the Internet, it sends out IP packets destined for that machine. These packets contain all the addressing information necessary to get them to their destination. NAT is concerned with these pieces of information:

- Source IP address (for example, 192.168.1.35)
- Source TCP or UDP port (for example, 2132)

When the packets pass through the NAT gateway they will be modified so that they appear to be coming from the NAT gateway itself. The NAT gateway will record the changes it makes in its state table so that it can a) reverse the changes on return packets and b) ensure that return packets are passed through the firewall and are not blocked. For example, the following changes might be made:

- Source IP: replaced with the external address of the gateway (for example, 24.5.0.5)
- Source port: replaced with a randomly chosen, unused port on the gateway (for example, 53136)

Neither the internal machine nor the Internet host is aware of these translation steps. To the internal machine, the NAT system is simply an Internet gateway. To the Internet host, the packets appear to come directly from the NAT system; it is completely unaware that the internal workstation even exists.

When the Internet host replies to the internal machine's packets, they will be addressed to the NAT gateway's external IP (24.5.0.5) at the translation port (53136). The NAT gateway will then search the state table to determine if the reply packets match an already established connection. A unique match will be found based on the IP/port combination which tells PF the packets belong to a connection initiated by the internal machine 192.168.1.35. PF will then make the opposite changes it made to the outgoing packets and forward the reply packets on to the internal machine.

Translation of ICMP packets happens in a similar fashion but without the source port modification.

NAT and Packet Filtering

NOTE: Translated packets must still pass through the filter engine and will be blocked or passed based on the filter rules that have been defined. The *only* exception to this rule is when the `pass` keyword is used within the `nat` rule. This will cause the NATed packets to pass right through the filtering engine.

Also be aware that since translation occurs *before* filtering, the filter engine will see the *translated* packet with the translated IP address and port as outlined in How NAT Works.

IP Forwarding

Since NAT is almost always used on routers and network gateways, it will probably be necessary to enable IP forwarding so that packets can travel between network interfaces on the OpenBSD machine. IP forwarding is enabled using the sysctl(3) mechanism:

```
# sysctl -w net.inet.ip.forwarding=1
# sysctl -w net.inet6.ip6.forwarding=1 (if using IPv6)
```

To make this change permanent, the following lines should be added to `/etc/sysctl.conf`:

```
net.inet.ip.forwarding=1
net.inet6.ip6.forwarding=1
```

These lines are present but commented out (prefixed with a #) in the default install. Remove the # and save the file. IP forwarding will be enabled when the machine is rebooted.

Configuring NAT

The general format for NAT rules in `pf.conf` looks something like this:

```
nat [pass] on interface [af] from src_addr [port src_port] to \
   dst_addr [port dst_port] -> ext_addr [pool_type] [static-port]
```

`nat`
: The keyword that begins a NAT rule.

`pass`
: Causes translated packets to completely bypass the filter rules.

`interface`
: The name of the network interface to translate packets on.

`af`
: The address family, either `inet` for IPv4 or `inet6` for IPv6. PF is usually able to determine this parameter based on the source/destination address(es).

`src_addr`
: The source (internal) address of packets that will be translated. The source address can be specified as:
  - A single IPv4 or IPv6 address.
  - A CIDR network block.
  - A fully qualified domain name that will be resolved via DNS when the ruleset is loaded. All resulting IP addresses will be substituted into the rule.
  - The name of a network interface. Any IP addresses assigned to the interface will be substituted into the rule at load time.
  - The name of a network interface followed by */`netmask`* (e.g. `/24`). Each IP address on the interface is combined with the netmask to form a CIDR network block which is substituted into the rule.
  - The name of a network interface followed by the `:network` keyword. The resulting CIDR network (e.g. 192.168.0.0/24) will be substituted into the rule when the ruleset is loaded.
  - A table.
  - Any of the above but negated using the `!` ("not") modifier.
  - A set of addresses using a list.
  - The keyword `any` meaning all addresses

`src_port`
: The source port in the Layer 4 packet header. Ports can be specified as:
  - A number between 1 and 65535
  - A valid service name from `/etc/services`
  - A set of ports using a list
  - A range:
    - `!=` (not equal)
    - `<` (less than)
    - `>` (greater than)
    - `<=` (less than or equal)
    - `>=` (greater than or equal)
    - `><` (range)
    - `<>` (inverse range)

    The last two are binary operators (they take two arguments) and do not include the arguments in the range.

  The `port` option is not usually used in `nat` rules because the goal is usually to NAT all traffic regardless of the port(s) being used.

`dst_addr`
: The destination address of packets to be translated. The destination address is specified in the same way as the source address.

`dst_port`
: The destination port in the Layer 4 packet header. This port is specified in the same way as the source port.

`ext_addr`
: The external (translation) address on the NAT gateway that packets will be translated to. The external address can be specified as:
  - A single IPv4 or IPv6 address.
  - A CIDR network block.
  - A fully qualified domain name that will be resolved via DNS when the ruleset is loaded.
  - The name of the external network interface. Any IP addresses assigned to the interface will be substituted into

the rule at load time.
- The name of the external network interface in parentheses `( )`. This tells PF to update the rule if the IP address(es) on the named interface changes. This is highly useful when the external interface gets its IP address via DHCP or dial-up as the ruleset doesn't have to be reloaded each time the address changes.
- The name of a network interface followed by the `:network` keyword. The resulting CIDR network (e.g. 192.168.0.0/24) will be substituted into the rule when the ruleset is loaded.
- A set of addresses using a list.

`pool_type`
: Specifies the type of address pool to use for translation.

`static-port`
: Tells PF not to translate the source port in TCP and UDP packets.

This would lead to a most basic form of this line similar to this:

```
nat on tl0 from 192.168.1.0/24 to any -> 24.5.0.5
```

This rule says to perform NAT on the `tl0` interface for any packets coming from 192.168.1.0/24 and to replace the source IP address with 24.5.0.5.

While the above rule is correct, it is not recommended form. Maintenance could be difficult as any change of the external or internal network numbers would require the line be changed. Compare instead with this easier to maintain line (`tl0` is external, `dc0` internal):

```
nat on tl0 from dc0/24 to any -> tl0
```

The advantage should be fairly clear: you can change the IP addresses of either interface without changing this rule.

When specifying an interface name for the translation address as above, the IP address is determined at pf.conf *load* time, not on the fly. If you are using DHCP to configure your external interface, this can be a problem. If your assigned IP address changes then NAT will continue translating outgoing packets using the old IP address. This will cause outgoing connections to stop functioning. To get around this, you can tell PF to automatically update the translation address by putting parentheses around the interface name:

```
nat on tl0 from dc0/24 to any -> (tl0)
```

There is one major limitation to doing this: Only the first IP alias on an interface is evaluated when the interface name is placed in parentheses.

Bidirectional Mapping (1:1 mapping)

A bidirectional mapping can be established by using the `binat` rule. A `binat` rule establishes a one to one mapping between an internal IP address and an external address. This can be useful, for example, to provide a web server on the internal network with its own external IP address. Connections from the Internet to the external address will be translated to the internal address and connections from the web server (such as DNS requests) will be translated to the external address. TCP and UDP ports are never modified with `binat` rules as they are with `nat` rules.

Example:

```
web_serv_int = "192.168.1.100"
web_serv_ext = "24.5.0.6"

binat on tl0 from $web_serv_int to any -> $web_serv_ext
```

Translation Rule Exceptions

Exceptions can be made to translation rules by using the `no` keyword. For example, if the NAT example above was modified to look like this:

```
no nat on tl0 from 192.168.1.10 to any
nat on tl0 from 192.168.1.0/24 to any -> 24.2.74.79
```

Then the entire 192.168.1.0/24 network would have its packets translated to the external address 24.2.74.79 except for 192.168.1.10.

Note that the first matching rule wins; if it's a `no` rule, then the packet is not translated. The `no` keyword can also be used with `binat` and `rdr` rules.

Checking NAT Status

To view the active NAT translations pfctl(8) is used with the `-s state` option. This option will list all the current NAT sessions:

```
# pfctl -s state
TCP 192.168.1.35:2132 -> 24.5.0.5:53136 -> 65.42.33.245:22 TIME_WAIT:TIME_WAIT
UDP 192.168.1.35:2491 -> 24.5.0.5:60527 -> 24.2.68.33:53   MULTIPLE:SINGLE
```

Explanations (first line only):

TCP
: The protocol being used by the connection.

192.168.1.35:2132
: The IP address (192.168.1.35) of the machine on the internal network. The source port (2132) is shown after the address. This is also the address that is replaced in the IP header.

24.5.0.5:53136
: The IP address (24.5.0.5) and port (53136) on the gateway that packets are being translated to.

65.42.33.245:22
: The IP address (65.42.33.245) and the port (22) that the internal machine is connecting to.

TIME_WAIT:TIME_WAIT
: This indicates what state PF believes the TCP connection to be in.

[Previous: Packet Filtering] [Contents] [Next: Traffic Redirection (Port Forwarding)]


www@openbsd.org

$OpenBSD: nat.html,v 1.13 2004/01/01 04:16:17 nick Exp $

[Précédent : Traduction d'Adresses Réseau] [Index] [Suivant : Raccourcis pour la Création de Jeux de Règles]

PF : Redirection de Trafic ("Forwarding" de Ports)

Table des Matières

Introduction

Quand vous utilisez la NAT, les machines sur votre réseau peuvent accéder à la totalité des ressources Internet. Qu'en est-il lorsqu'une de vos machines derrière la passerelle NAT a besoin d'être accédée depuis Internet ? la redirection est utilisée pour résoudre ce genre de problèmes. Elle permet au trafic entrant d'être acheminé vers une machine derrière la passerelle NAT.

Prenons un exemple :

```
rdr on tl0 proto tcp from any to any port 80 -> 192.168.1.20
```

La ligne ci-dessus redirige le trafic à destination du port TCP 80 (serveur web) vers une machine sur le réseau d'adresse IP 192.168.1.20. Ainsi, même si cette machine est derrière votre passerelle, le monde externe peut y avoir accès.

La partie `from any to any` de la ligne `rdr` précitée peut être assez utile. Si vous voulez restreindre les adresses ou les sous-réseaux autorisés à avoir accès au serveur web sur le port 80, vous pouvez le faire de la façon suivante :

```
rdr on tl0 proto tcp from 27.146.49.0/24 to any port 80 -> \
   192.168.1.20
```

Ceci aura pour effet de rediriger uniquement le trafic en provenance du sous-réseau spécifié. Notez que cela implique la possibilité de rediriger différentes machines en entrée vers des machines derrière la passerelle. Une telle fonctionnalité est utile. Par exemple, vous pouvez autoriser des utilisateurs sur des sites distants à accèder à leur propre machine dans la mesure où vous connaissez leur adresse IP :

```
rdr on tl0 proto tcp from 27.146.49.14 to any port 80 -> \
   192.168.1.20
rdr on tl0 proto tcp from 16.114.4.89 to any port 80 -> \
   192.168.1.22
rdr on tl0 proto tcp from 24.2.74.178 to any port 80 -> \
   192.168.1.23
```

Redirection et Filtrage de Paquets

REMARQUE : Les paquets redirigés doivent encore être autorisés par le moteur de filtrage. Ils seront bloqués ou autorisés selon les règles de filtrage qui ont été définies. La *seule* exception à cette règle est lorsque le mot-clé `pass` est utilisé dans la règle `rdr`; dans ce cas, les paquets redirigés seront automatiquement autorisés par le moteur de filtrage.

Notez aussi que la traduction est effectuée *avant* le filtrage. Le moteur de filtrage verra le paquet *traduit*, ç.à.d. après que son adresse IP destination et/ou port de destination soient modifiés pour correspondre à l'adresse/au port de redirection spécifié(e) dans la règle `rdr`. Considérons le cas suivant :

- 192.0.2.1 - hôte sur Internet
- 24.65.1.13 - adresse IP externe du routeur OpenBSD
- 192.168.1.5 - adresse IP interne du serveur web

Règle de redirection :

```
rdr on tl0 proto tcp from 192.0.2.1 to 24.65.1.13 port 80 \
   -> 192.168.1.5 port 8000
```

Avant que la règle de redirection ne soit utilisée, le paquet est traité :

- Adresse source : 192.0.2.1
- Port source : 4028 (choisi de manière arbitraire par le système d'exploitation)
- Adresse de destination : 24.65.1.13
- Port de destination : 80

Puis la règle de redirection est évaluée :

- Adresse source : 192.0.2.1
- Port source : 4028
- Adresse de destination : 192.168.1.5
- Port de destination : 8000

Le moteur de filtrage verra le paquet IP tel qu'il apparaît après la traduction effectuée par le mécanisme de redirection.

Implications au niveau de la Sécurité

La redirection a des implications au niveau de la sécurité. La création d'une ouverture pour autoriser le trafic à destination du réseau interne protégé peut causer la compromission d'une machine sur ce dernier. Par exemple, si le trafic est redirigé vers un serveur web interne et par malheur, une vulnérabilité est découverte dans le service web ou dans un script CGI executé par le serveur, alors la machine peut être compromise par un intrus provenant d'Internet. Une fois cette compromission réalisée, l'intrus peut rebondir à partir de cette machine vers le réseau interne ce qui est bien entendu permis par le pare-feu.

Ces risques peuvent être minimisés en confinant de manière stricte le système accédé depuis l'extérieur à un réseau séparé. Ce

réseau est souvent appelé zone démilitarisée (DMZ) ou réseau de service privé (Private Service Network, PSN). De cette façon, si le serveur web est compromis, les effets peuvent être limités au réseau DMZ/PSN en filtrant soigneusement le trafic autorisé entre DMZ/PSN et vos autres réseaux.

Redirection et Réflexion

Souvent, les règles de redirection sont utilisées pour faire suivre des connexions Internet entrantes à un serveur local disposant d'une adresse privée sur le réseau interne comme le montre l'exemple suivant :

```
server = 192.168.1.40

rdr on $ext_if proto tcp from any to $ext_if port 80 -> $server \
   port 80
```

Mais lorsque la règle de redirection est testée par un client du LAN, elle ne fonctionne pas. C'est normal car les règles de redirection ne s'appliquent qu'aux paquets qui passent à travers l'interface spécifiée dans chacune d'elles (dans l'exemple précité, c'est l'interface externe `$ext_if`). Cependant, une connexion émanant d'un client interne et à destination de l'adresse externe du pare-feu n'implique pas que les paquets vont passer à travers l'interface externe du pare-feu. La pile TCP/IP sur le pare- feu compare l'adresse de destination des paquets entrants avec ses propres adresses et aliases et détecte les connexions qui lui sont destinées lorsque les paquets constituant celles-ci passent son interface interne. De tels paquets ne passent pas physiquement à travers l'interface externe, et de toute façon la pile ne simule pas ce passage. Ainsi, PF ne voit jamais ces paquets sur l'interface externe. Du coup, la règle de redirection sur l'interface externe n'est jamais appliquée.

Que doit-on faire alors ? ajouter une seconde règle de redirection sur l'interface interne ? non, ça ne fonctionnera pas mieux. Lorsque le client local se connecte à l'interface externe du pare-feu, le paquet initial de l'échange TCP atteint le pare-feu via son interface interne. La nouvelle règle de redirection s'applique et l'adresse de destination est remplacée par celle du serveur interne. Le paquet et alors redirigé par l'interface interne du pare-feu vers le serveur interne. Mais l'adresse source n'a pas été modifiée. Elle contient l'adresse du client local. Le serveur envoie alors directement les réponses au client. Le pare-feu ne voit jamais le retour et n'a aucune chance de traduire correctement les paquets de retour. Le client reçoit une réponse à partir d'une source inattendue. Il met alors fin à cette connexion.

Cependant, il est souvent souhaitable que les clients sur le LAN se connectent au même serveur interne que les clients externes et de manière aussi transparente. Il existe plusieurs solutions à ce problème :

DNS en "Split-Horizon"

Il est possible de configurer les serveurs DNS pour fournir une réponse différente selon la provenance de la requête : clients internes ou externes. Ainsi les clients internes recevront l'adresse interne du serveur en réponse à leur demande de résolution de nom. Ils pourront alors se connecter directement au serveur local sans impliquer le pare-feu. Ce dernier sera alors moins sollicité.

Déplacer le Serveur Vers un Réseau Local Séparé

Une autre solution consiste à ajouter une carte réseau au pare-feu et à déplacer le serveur local vers un réseau dédié de type DMZ. Les connexions des clients locaux seront alors redirigées de la même manière que les connexions en provenance d'Internet. L'utilisation de réseaux séparés a plusieurs avantages. Entre autres, une amélioration du niveau de sécurité en isolant le serveur des machines internes. Si le serveur (qui, nous le rappelons, est joignable d'Internet) est compromis, il ne peut se connecter à des machines internes directement vu que toutes ces connexions doivent passer à travers le pare-feu.

Mandater les Connexions TCP ("TCP Proxying")

Un proxy TCP générique peut être mis en place sur le pare-feu. Ce proxy devra écouter sur le port de redirection ou accepter les connexions sur l'interface interne et redirigées sur le port sur lequel il écoute. Lorsqu'un client se connecte sur le pare-feu, le proxy

accepte la connexion, établit une seconde connexion au serveur interne, et véhicule les données entre les deux connexions.

Des proxies simples peuvent être crées avec inetd(8) et nc(1). L'entrée suivante dans le fichier `/etc/inetd.conf` crée une socket d'écoute rattachée à l'adresse de loopback (127.0.0.1) et le port 5000. Les connexions sont redirigées sur le port 80 du serveur 192.168.1.10.

```
127.0.0.1:5000 stream tcp nowait nobody /usr/bin/nc nc -w \
   20 192.168.1.10 80
```

La règle de redirection suivante redirige le port 80 sur l'interface interne du proxy :

```
rdr on $int_if proto tcp from $int_net to $ext_if port 80 -> \
   127.0.0.1 port 5000
```

Combinaison RDR et NAT

A l'aide d'une règle NAT supplémentaire sur l'interface interne, la traduction manquante d'adresse source décrite plus haut peut être réalisée.

```
rdr on $int_if proto tcp from $int_net to $ext_if port 80 -> \
   $server
no nat on $int_if proto tcp from $int_if to $int_net
nat on $int_if proto tcp from $int_net to $server port 80 -> \
   $int_if
```

Ceci aura pour effet d'effectuer une autre traduction d'adresse du paquet initial envoyé par le client lorsque celui-ci est redirigé à travers l'interface interne. Cette seconde opération de traduction remplacera l'adresse source du client par l'adresse interne du pare-feu. Le serveur interne répondra alors à l'adresse interne du pare-feu, qui effectuera les opérations de NAT et de RDR inverses avant de faire suivre le paquet au client. Cette méthode est relativement complexe. Elle crée deux états séparés pour chaque connexion redirigée. Il faut prendre des précautions pour éviter que la règle de NAT s'applique au reste du trafic tel que les connexions en provenance de hôtes externes (via d'autres redirections) ou en provenance du pare-feu lui-même. Il est à noter que la règle `rdr` précitée fera apparaître à la pile TCP/IP des paquets en provenance du réseau interne et à destination de ce dernier.

Cependant, nous recommandons de privilégier plutôt les solutions précédemment mentionnées.

[Précédent : Traduction d'Adresses Réseau] [Index] [Suivant : Raccourcis pour la Création de Jeux de Règles]


www@openbsd.org

Originally [OpenBSD: rdr.html,v 1.14]
$Translation: rdr.html,v 1.5 2004/03/23 20:29:05 saad Exp $
$OpenBSD: rdr.html,v 1.4 2004/03/26 08:40:52 jufi Exp $

[Précédent : Redirection de Trafic ("Forwarding" de Ports)] [Index] [Suivant : Options de Fonctionnement]

PF : Raccourcis pour la Création de Jeux de Règles

Table des Matières

Introduction

PF offre plusieurs moyens pour simplifier un jeu de règles. Quelques bons exemples sont les macros et les listes. De plus, le langage d'écriture des jeux de règles, ou grammaire, offre aussi quelques raccourcis pour rendre un jeu de règles plus simple. En règle générale, plus un jeu de règles est facile plus il est facile de le comprendre et le maintenir.

Utiliser les Macros

Les macros sont utiles car elles fournissent une alternative au codage en dur des adresses, des numéros de ports, des noms d'interfaces etc. dans un jeu de règles. Est-ce que l'adresse IP d'un serveur a été modifiée ? Pas de problème, il suffit de mettre à jour la macro; nul besoin de retoucher les règles de filtrage que vous avez mis du temps et de l'énergie à construire selon vos besoins.

Une convention usuelle dans l'écriture des jeux de règles PF est de définir une macro pour chaque interface réseau. Si une carte réseau doit être remplacée par une autre carte qui utilise un pilote différent (en changeant une 3Com par une Intel par exemple), la macro est mise à jour et les règles de filtrage fonctionneront comme avant. Un autre bénéfice est le déploiement du même jeu de règles sur plusieurs machines. Certaines de ces machines peuvent avoir des cartes réseau différentes. L'utilisation de macros pour définir les cartes réseau permet d'installer les jeux de règles avec un minimum d'édition. L'utilisation de macros pour définir des informations dans un jeu de règles sujet à changements, telles que les numéros de ports, les adresses IP, et les noms d'interfaces, est une pratique recommandée.

```
# définition de macros pour chaque interface réseau
IntIF = "dc0"
ExtIF = "fxp0"
DmzIF = "fxp1"
```

Une autre convention usuelle est d'utiliser les macros pour définir des adresses IP et les blocs réseau. Ceci aura pour effet de réduire de manière significative les opérations de maintenance d'un jeu de règles lorsque les adresses IP y figurant changent.

```
# définition de nos réseaux
IntNet = "192.168.0.0/24"
ExtAdd = "24.65.13.4"
DmzNet = "10.0.0.0/24"
```

Si le réseau interne doit être étendu ou modifié, il suffit de mettre à jour la macro :

```
IntNet = "{ 192.168.0.0/24, 192.168.1.0/24 }"
```

Une fois le jeu de règles rechargé, tout fonctionnera comme avant.

Utiliser les Listes

Prenons comme exemple quelques bonnes règles à inclure dans vos jeux de règles pour gérer les adresses RFC 1918 qui ne doivent pas être routées sur Internet et qui d'habitude, sont utilisées dans un but malicieux :

```
block in  quick on tl0 inet from 127.0.0.0/8 to any
block in  quick on tl0 inet from 192.168.0.0/16 to any
block in  quick on tl0 inet from 172.16.0.0/12 to any
block in  quick on tl0 inet from 10.0.0.0/8 to any
block out quick on tl0 inet from any to 127.0.0.0/8
block out quick on tl0 inet from any to 192.168.0.0/16
block out quick on tl0 inet from any to 172.16.0.0/12
block out quick on tl0 inet from any to 10.0.0.0/8
```

Simplifions maintenant les règles ci-dessus :

```
block in  quick on tl0 inet from { 127.0.0.0/8, 192.168.0.0/16, \
   172.16.0.0/12, 10.0.0.0/8 } to any
block out quick on tl0 inet from any to { 127.0.0.0/8, \
   192.168.0.0/16, 172.16.0.0/12, 10.0.0.0/8 }
```

Le jeu de règles a été réduit de six lignes : on est passé de huit lignes à deux. On peut encore simplifier en utilisant des macros en conjonction d'une liste :

```
NoRouteIPs = "{ 127.0.0.0/8, 192.168.0.0/16, 172.16.0.0/12, \
   10.0.0.0/8 }"
ExtIF = "tl0"
block in  quick on $ExtIF from $NoRouteIPs to any
block out quick on $ExtIF from any to $NoRouteIPs
```

Notez que les macros et les listes simplifient le fichier `pf.conf` mais que les lignes sont en réalité interprétées par pfctl(8) en plusieurs lignes. Ainsi l'exemple précédent est interprété comme suit :

```
block in  quick on tl0 inet from 127.0.0.0/8 to any
block in  quick on tl0 inet from 192.168.0.0/16 to any
block in  quick on tl0 inet from 172.16.0.0/12 to any
block in  quick on tl0 inet from 10.0.0.0/8 to any
block out quick on tl0 inet from any to 10.0.0.0/8
```

```
block out quick on tl0 inet from any to 172.16.0.0/12
block out quick on tl0 inet from any to 192.168.0.0/16
block out quick on tl0 inet from any to 127.0.0.0/8
```

Comme vous pouvez le voir, la simplification des règles figurant dans le fichier `pf.conf` est une facilité offerte à l'auteur et à la personne chargée de la maintenance de ce fichier. Ce n'est pas une simplification réelle des règles traitées par pf(4).

Les macros peuvent être utilisées n'importe où dans le fichier de règles PF et pas seulement pour définir des adresses et des ports :

```
pre = "pass in quick on ep0 inet proto tcp from "
post = "to any port { 80, 6667 } keep state"

# classe de David
$pre 21.14.24.80 $post

# logement de Nick
$pre 24.2.74.79 $post
$pre 24.2.74.178 $post
```

Les macros précédentes sont interprétées comme suit :

```
pass in quick on ep0 inet proto tcp from 21.14.24.80 to any \
   port = 80 keep state
pass in quick on ep0 inet proto tcp from 21.14.24.80 to any \
   port = 6667 keep state
pass in quick on ep0 inet proto tcp from 24.2.74.79 to any \
   port = 80 keep state
pass in quick on ep0 inet proto tcp from 24.2.74.79 to any \
   port = 6667 keep state
pass in quick on ep0 inet proto tcp from 24.2.74.178 to any \
   port = 80 keep state
pass in quick on ep0 inet proto tcp from 24.2.74.178 to any \
   port = 6667 keep state
```

Grammaire de PF

La grammaire de PF est assez flexible ce qui permet une grande flexibilité dans les jeux de règles. PF est capable de supposer certains mots-clés ce qui veut dire qu'ils n'ont pas besoin d'être explicitement inclus dans une règle, et l'ordonnancement des mots-clés est relâché de telle façon à ce qu'il ne soit pas nécessaire de mémoriser une syntaxe stricte.

Elimination of Keywords

Pour définir une politique de blocage par défaut, deux règles sont utilisés :

```
block in  all
block out all
```

Ceci peut être réduit à :

```
block all
```

Quand aucune direction n'est spécifiées, PF suppose que la règle s'applique aux paquets passant dans les deux sens.

De manière similaire, les clauses "`from any to any`" et "`all`" peuvent ne pas figurer dans une règle. Par exemple :

```
block in on rl0 all
pass  in quick log on rl0 proto tcp from any to any port 22 keep state
```

peut être simplifiée de la manière suivante :

```
block in on rl0
pass  in quick log on rl0 proto tcp to port 22 keep state
```

La première règle bloque tous les paquets en entrée de n'importe où vers n'importe où sur l'interface rl0, et la deuxième règle laisse passer le trafic TCP sur rl0 à destination du port 22.

Simplification des Règles avec `return`

Un jeu de règles utilisé pour bloquer des paquets et répondre avec un TCP RST ou un ICMP Unreachable peut être écrit comme suit :

```
block in all
block return-rst in proto tcp all
block return-icmp in proto udp all
block out all
block return-rst out proto tcp all
block return-icmp out proto udp all
```

Il peut être largement simplifié comme suit :

```
block return
```

Quand PF voit le mot-clé `return`, il "sait" ce qu'il faut envoyer comme réponse (ou pas en envoyer du tout), selon le protocole du paquet bloqué.

Ordonnancement de Mots-clés

L'ordre selon lequel les mots-clés sont spécifiés est flexible dans la plupart des cas. Par exemple, une règle écrite comme suit :

```
pass in log quick on rl0 proto tcp to port 22 \
   flags S/SA keep state queue ssh label ssh
```

Peut aussi être écrite de cette façon :

```
pass in quick log on rl0 proto tcp to port 22 \
   queue ssh keep state label ssh flags S/SA
```

Des variations similaires fonctionneront aussi.

[Précédent : Redirection de Trafic ("Forwarding" de Ports)] [Index] [Suivant : Options de Fonctionnement]


www@openbsd.org

Originally [OpenBSD: shortcuts.html,v 1.10]
$Translation: shortcuts.html,v 1.2 2004/01/02 22:10:05 saad Exp $
$OpenBSD: shortcuts.html,v 1.2 2004/01/03 19:45:23 jufi Exp $

[Précédent : Raccourcis pour La Création des Jeux de Règles] [Index] [Suivant : Scrub (Normalisation de Paquets)]

PF : Options de Fonctionnement

Des options sont utilisées pour contrôler le fonctionnement de PF. Celles-ci sont spécifiées dans le fichier `pf.conf` en utilisant la directive `set`.

`set block-policy`

Paramètre le comportement pour par défaut des règles de filtrage qui ont pour effet de bloquer les paquets (mot-clé "block").

- `drop` - Le paquet est détruit sans aucune notification.
- `return` - un paquet TCP RST est renvoyé pour les paquets TCP et un paquet ICMP Unreachable est renvoyé pour tous les autres.

Il est à noter que les règles de filtrage individuelles peuvent avoir leur propre réponse.

`set limit`

`frags` - nombre maximal d'entrées dans la zone mémoire utilisée pour le réassemblage de paquets (règles scrub). La valeur par défaut est 5000.
`states` - nombre maximal d'entrées dans la zone mémoire utilisée pour les entrées de la table d'état (règles de filtrage qui contiennent le mot-clé). La valeur par défaut est 10000.

`set loginterface` *`int`*

Paramètre l'interface pour laquelle PF devra récupérer des statistiques telles que le nombre d'octets entrants/sortant les paquets acceptés/bloqués. Les statistiques ne peuvent être récupérées que pour `une` interface à la fois. Il est à noter que les indicateurs `match`, `bad-offset`, etc., et les indicateurs de la table d'état sont enregistrés que `loginterface` soit positionnée ou pas.

`set optimization`

Optimise PF pour l'un de ces environnements réseau :

- `normal` - convient à pratiquement tous les réseaux. Valeur par défaut.
- `high-latency` - réseaux à haute latence tels que les réseaux à connexion satellite.
- `aggressive` - les connexions expirent plus rapidement de la table d'état. Les pré-requis mémoire sont ainsi fortement réduits sur un pare-feu particulièrement chargé au risque de terminer des connexions inactives trop rapidement.
- `conservative` - paramétrage extrêmement conservateur. Contrairement à `aggressive`, ce paramétrage évite de terminer les connexions inactives ce qui se traduit par une plus grande utilisation mémoire et une utilisation du processeur un peu plus soutenue.

`set timeout`

`interval` - nombre de secondes entre les purges d'états expirés et de fragments de paquets.
`frag` - nombre de secondes avant qu'un fragment non assemblé n'expire.

Exemple :

```
set timeout interval 10
set timeout frag 30
set limit { frags 5000, states 2500 }
set optimization high-latency
set block-policy return
set loginterface dc0
```

[Précédent : Raccourcis pour La Création des Jeux de Règles] [Index] [Suivant : Scrub (Normalisation de Paquets)]


www@openbsd.org

Originally [OpenBSD: options.html,v 1.6]
$Translation: options.html,v 1.3 2004/01/02 22:10:05 saad Exp $
$OpenBSD: options.html,v 1.3 2004/01/03 19:45:23 jufi Exp $

[Précédent : Options de Fonctionnement] [Index] [Suivant : Ancres et Bases de Règles Nommées (Sub)]

PF : Scrub (Normalisation de Paquets)

Table des Matières

Introduction

Le "scrubbing" est la normalisation de paquets utilisée pour supprimer toute ambiguïté dans l'interprétation d'un paquet qui sera effectuée par la destination finale de ce dernier. La directive `scrub` réassemble aussi des paquets fragmentés, afin de protéger certains systèmes d'exploitation de quelques types d'attaques. Cette directive rejette aussi les paquets TCP contenants des combinaisons invalides de drapeaux. Voici un exemple simple d'utilisation de la directive scrub :

```
scrub in all
```

Ceci aura pour effet d'appliquer le "scrub" sur tous les paquets en entrée de chaque interface.

Il existe des cas où il ne faut pas appliquer le "scrub". Par exemple, une interface qui véhicule du trafic NFS à travers PF. Certaines plates-formes non basées sur OpenBSD envoient (et s'attendent à recevoir) des paquets étranges -- des paquets fragmentés avec le bit "do not fragment" (ne pas fragmenter) positionné, qui sont rejetés (comportement normal) par `scrub`. On peut résoudre ce problème en utilisant l'option `no-df`. Un autre exemple est l'utilisation de jeux multi-joueurs qui ont des problèmes de connexion à travers PF lorsque `scrub` est activé. Mis à part ces cas quelque peu inhabituels, la normalisation de paquets "scrub" est une pratique *hautement recommandée*.

La syntaxe de la directive `scrub` est très similaire à la syntaxe de filtrage ce qui rend aisé la normalisation de certains paquets et pas les autres.

Vous pouvez trouver plus d'informations concernant le principe et les concepts de la normalisation de paquets dans l'article Network Intrusion Detection: Evasion, Traffic Normalization, and End-to-End Protocol Semantics.

Options

Les options de `scrub` sont les suivantes :

`no-df`
: Supprime le bit *don't fragment* de l'en-tête du paquet IP. Quelques systèmes d'exploitation sont connus pour générer des

paquets fragmentés avec le bit *don't fragment* positionné. Ceci est particulièrement vrai dans le cas de NFS. `scrub` va bloquer tous les paquets qui sont dans ce cas sauf si l'option `no-df` est spécifiée. Vu que certains systèmes d'exploitation génèrent des paquets *don't fragment* avec un champ d'identification à zéeo au niveau de l'en-tête IP, il est recommandé d'utiliser `no-df` conjointement avec l'option `random-id`.

`random-id`
: Remplace le champ d'identification IP des paquets sortants avec des valeurs aléatoires pour contourner les valeurs prévisibles utilisées par certains systèmes d'exploitation. Cette option s'applique uniquement pour les paquets sortants qui ne sont pas fragmentés après le réassemblage optionnel des paquets.

`min-ttl` *num*
: S'assure que le Time To Live (TTL) est au moins égal à la valeur donnée en argument dans les en-têtes des paquets IP.

`max-mss` *num*
: S'assure que le Maximum Segment Size (MSS) est au plus égal à la valeur donnée en argument dans les en-têtes des paquets TCP.

`fragment reassemble`
: Met dans une mémoire tampon les fragments de paquets et les réassemble en paquet complet avant de les transmettre au moteur de filtrage. Les règles de filtrage peuvent ainsi se charger de filtrer le paquet complet sans se soucier des fragments. L'inconvénient est la mémoire additionnelle nécessaire pour le tampon contenant les fragments de paquets. C'est le comportement par défaut lorsqu'aucune option `fragment` n'est spécifiée. C'est aussi la seule option `fragment` qui fonctionne avec la NAT.

`fragment crop`
: Supprime les fragments dupliqués et tout chevauchement entre fragment. Contrairement à `fragment reassemble`, les fragments ne sont pas gardés en mémoire tampon mais sont transmis dès leur arrivée.

`fragment drop-ovl`
: Assez similaire à `fragment crop`. Tous les paquets fragments de paquets dupliqués et se chevauchant sont supprimés ainsi que tous les fragments suivants qui correspondent à ces fragments.

`reassemble tcp`
: Normalise de manière "stateful" les connexions TCP. Lorsque `scrub reassemble tcp` est utilisée, une direction (in/out) peut ne pas être spécifiée. Les normalisations suivantes sont effectuées :
  - Aucune extrêmité de la connexion n'est autorisée à réduire le TTL IP. Cette normalisation est appliquée pour assurer une protection contre un attaquant qui envoie un paquet de telle façon à ce que ce dernier atteigne le pare-feu, affecte les informations d'état mémorisées pour cette connexion, et expire avant d'atteindre sa destination finale. Le TTL de tous les paquets est positionné à la valeur la plus haute observée pour la connexion.
  - Module les "timestamps" RFC 1323 dans l'en-tête des paquets TCP avec un nombre aléatoire. Ceci peut empêcher un observateur de déduire le "uptime" d'un hôte ou de deviner combien de hôtes sont derrière une passerelle NAT.

Exemples :

```
scrub in on fxp0 all fragment reassemble min-ttl 15 max-mss 1400
scrub in on fxp0 all no-df
scrub    on fxp0 all reassemble tcp
```

[Précédent : Options de Fonctionnement] [Index] [Suivant : Ancres et Bases de Règles Nommées (Sub)]


www@openbsd.org

Originally [OpenBSD: scrub.html,v 1.8]
$Translation: scrub.html,v 1.3 2004/02/19 20:31:29 saad Exp $
$OpenBSD: scrub.html,v 1.3 2004/02/20 06:33:11 jufi Exp $

[Previous: Scrub (Packet Normalization)] [Contents] [Next: Packet Queueing and Prioritization]

PF: Anchors and Named Rulesets

Table of Contents

Introduction

In addition to the main ruleset, PF can also evaluate sub rulesets. Since sub rulesets can be manipulated on the fly by using pfctl(8), they provide a convenient way of dynamically altering an active ruleset. Whereas a table is used to hold a dynamic list of addresses, a sub ruleset is used to hold a dynamic set of filter, `nat`, `rdr`, and `binat` rules.

Sub rulesets are attached to the main ruleset by using anchors. There are four types of `anchor` rules:

- `anchor` *`name`* - evaluates all filter rules in the anchor *`name`*
- `binat-anchor` *`name`* - evaluates all `binat` rules in the anchor *`name`*
- `nat-anchor` *`name`* - evaluates all `nat` rules in the anchor *`name`*
- `rdr-anchor` *`name`* - evaluates all `rdr` rules in the anchor *`name`*

Only the main ruleset can contain `anchor` rules.

Named Rulesets

A named ruleset is a group of filter and/or translation rules that has been assigned a name. An `anchor` point may contain more than one such ruleset. When PF comes across an `anchor` rule in the main ruleset, it will evaluate all the rulesets attached to that `anchor` point in alphabetical order according to their names. Processing will then continue in the main ruleset unless the packet matches a filter rule that uses the `quick` option or a translation rule within the anchor in which case the match will be considered final and will abort the evaluation of rules in both the anchor and the main rulesets.

For example:

```
ext_if = "fxp0"

block on $ext_if all
```

```
pass  out on $ext_if all keep state
anchor goodguys
```

This ruleset sets a default deny policy on `fxp0` for both incoming and outgoing traffic. Traffic is then statefully passed out and an anchor rule is created named `goodguys`. Anchors can be populated with rules by two methods:

- using a `load` rule
- using pfctl(8)

The `load` rule causes `pfctl` to populate the specified anchor and named ruleset by reading rules from a text file. Example:

```
load anchor goodguys:ssh from "/etc/anchor-goodguys-ssh"
```

When the main ruleset is loaded, the rules listed in the file `/etc/anchor-goodguys-ssh` will be loaded into the named ruleset `ssh` attached to the `goodguys` anchor.

To add rules to an anchor using `pfctl`, the following type of command can be used:

```
# echo "pass in proto tcp from 192.0.2.3 to any port 22" \
   | pfctl -a goodguys:ssh -f -
```

This adds a `pass` rule to the ruleset named `ssh` attached to the `goodguys` anchor. PF will then evaluate this rule (and any other filter rules that get added) when it reaches the `anchor goodguys` line in the main ruleset.

Rules can also be saved and loaded from a text file:

```
# cat >> /etc/anchor-goodguys-www
pass in proto tcp from 192.0.2.3 to any port 80
pass in proto tcp from 192.0.2.4 to any port { 80 443 }

# pfctl -a goodguys:www -f /etc/anchor-goodguys-www
```

This loads the rules from the `/etc/anchor-goodguys-www` file into the named ruleset `www` in the `goodguys` anchor.

Filter and translation rules can be loaded into a named ruleset using the same syntax and options as rules loaded into the main ruleset. One caveat, however, is that any macros that are used must also be defined within the named ruleset; macros that are defined in the main ruleset are *not* visible from named rulesets.

Each named ruleset, as well as the main ruleset, exist separately from the other rulesets. Operations done on one ruleset, such as flushing the rules, do not affect any of the others. In addition, removing an anchor point from the main ruleset does not destroy the anchor or any named rulesets that are attached to that anchor. A named ruleset is not destroyed until it's flushed of all rules using pfctl(8). Once an anchor point has no named rulesets attached to it, it's also destroyed.

Anchor Options

Optionally, `anchor` rules can specify interface, protocol, source and destination address, etc., using the same syntax as filter rules. When such information is given, `anchor` rules are only processed if the packet matches the `anchor` rule's definition. For example:

```
ext_if = "fxp0"
```

```
block on $ext_if all
pass  out on $ext_if all keep state
anchor ssh in on $ext_if proto tcp from any to any port 22
```

The rules in the anchor `ssh` are only evaluated for TCP packets destined for port 22 that come in on `fxp0`. Rules are then added to the `anchor` like so:

```
# echo "pass in from 192.0.2.10 to any" | pfctl -a ssh:allowed -f -
```

So, even though the filter rule doesn't specify an interface, protocol, or port, the host 192.0.2.10 will only be permitted to connect using SSH because of the `anchor` rule's definition.

Manipulating Named Rulesets

Manipulation of named rulesets is performed via `pfctl`. It can be used to add and remove rules from a ruleset without reloading the main ruleset.

To list all the rules in the ruleset `allowed` attached to the `ssh` anchor:

```
# pfctl -a ssh:allowed -s rules
```

To flush all filter rules from the same ruleset:

```
# pfctl -a ssh:allowed -F rules
```

If the ruleset name is omitted, the action applies to all rules in the anchor.

For a full list of commands, please see pfctl(8).

[Previous: Scrub (Packet Normalization)] [Contents] [Next: Packet Queueing and Prioritization]


www@openbsd.org

$OpenBSD: anchors.html,v 1.10 2004/01/01 04:16:17 nick Exp $

[Previous: Anchors and Named (Sub) Rulesets] [Contents] [Next: Address Pools and Load Balancing]

PF: Packet Queueing and Prioritization

Table of Contents

Queueing

To queue something is to store it, in order, while it awaits processing. In a computer network, when data packets are sent out from a host, they enter a queue where they await processing by the operating system. The operating system then decides which queue and which packet(s) from that queue should be processed. The order in which the operating system selects the packets to process can affect network performance. For example, imagine a user running two network applications: SSH and FTP. Ideally, the SSH packets should be processed before the FTP packets because of the time-sensitive nature of SSH; when a key is typed in the SSH client, an immediate response is expected, but an FTP transfer being delayed by a few extra seconds hardly bears any notice. But what happens if the router handling these connections processes a large chunk of packets from the FTP connection before processing the SSH connection? Packets from the SSH connection will remain in the queue (or possibly be dropped by the router if the queue isn't big enough to hold all of the packets) and the SSH session may appear to lag or slow down. By modifying the queueing strategy being used, network bandwidth can be shared fairly between different applications, users, and computers.

Note that queueing is only useful for packets in the *outbound* direction. Once a packet arrives on an interface in the inbound direction it's already too late to queue it -- it's already consumed network bandwidth to get to the interface that just received it. The only solution is to enable queueing on the adjacent router or, if the host that received the packet is acting as a router, to enable queueing on the internal interface where packets exit the router.

Schedulers

The scheduler is what decides which queues to process and in what order. By default, OpenBSD uses a First In First Out (FIFO) scheduler. A FIFO queue works like the line-up at a supermarket or a bank -- the first item into the queue is the first processed. As new packets arrive they are added to the end of the queue. If the queue becomes full, newly arriving packets are dropped. This is known as tail-drop.

OpenBSD supports two additional schedulers:

- Class Based Queueing
- Priority Queueing

Class Based Queueing

Class Based Queueing (CBQ) is a queueing algorithm that divides a network connection's bandwidth among multiple queues or classes. Each queue then has traffic assigned to it based on source or destination address, port number, protocol, etc. A queue may optionally be configured to borrow bandwidth from its parent queue if the parent is being under-utilized. Queues are also given a priority such that those containing interactive traffic, such as SSH, can have their packets processed ahead of queues containing bulk traffic, such as FTP.

CBQ queues are arranged in an hierarchical manner. At the top of the hierarchy is the root queue which defines the total amount of bandwidth available. Child queues are created under the root queue, each of which can be assigned some portion of the root queue's bandwidth. For example, queues might be defined as follows:

```
Root Queue (2Mbps)
        Queue A (1Mbps)
        Queue B (500Kbps)
        Queue C (500Kbps)
```

In this case, the total available bandwidth is set to 2 megabits per second (Mbps). This bandwidth is then split among three child queues.

The hierarchy can further be expanded by defining queues within queues. To split bandwidth equally among different users and also classify their traffic so that certain protocols don't starve others for bandwidth, a queueing structure like this might be defined:

```
Root Queue (2Mbps)
        UserA (1Mbps)
                ssh (50Kbps)
                bulk (950Kbps)
        UserB (1Mbps)
                audio (250Kbps)
                bulk (750Kbps)
                        http (100Kbps)
                        other (650Kbps)
```

Note that at each level the sum of the bandwidth assigned to each of the queues is not more than the bandwidth assigned to the parent queue.

A queue can be configured to borrow bandwidth from its parent if the parent has excess bandwidth available due to it not being used by the other child queues. Consider a queueing setup like this:

```
Root Queue (2Mbps)
        UserA (1Mbps)
                ssh (100Kbps)
                ftp (900Kbps, borrow)
        UserB (1Mbps)
```

If traffic in the `ftp` queue exceeds 900Kbps and traffic in the `UserA` queue is less than 1Mbps (because the ssh queue is using less than its assigned 100Kbps), the `ftp` queue will borrow the excess bandwidth from `UserA`. In this way the `ftp` queue is able to use more than its assigned bandwidth when it faces overload. When the `ssh` queue increases its load, the borrowed bandwidth will be returned.

CBQ assigns each queue a priority level. Queues with a higher priority are preferred during congestion over queues with a lower priority as long as both queues share the same parent (in other words, as long as both queues are on the same branch in the hierarchy). Queues with the same priority are processed in a round-robin fashion. For example:

```
Root Queue (2Mbps)
        UserA (1Mbps, priority 1)
                ssh (100Kbps, priority 5)
                ftp (900Kbps, priority 3)
        UserB (1Mbps, priority 1)
```

CBQ will process the `UserA` and `UserB` queues in a round-robin fashion -- neither queue will be preferred over the other. During the time when the `UserA` queue is being processed, CBQ will also process its child queues. In this case, the `ssh` queue has a higher priority and will be given preferential treatment over the `ftp` queue if the network is congested. Note how the `ssh` and `ftp` queues do not have their priorities compared to the `UserA` and `UserB` queues because they are not all on the same branch in the hierarchy.

For a more detailed look at the theory behind CBQ, please see References on CBQ.

Priority Queueing

Priority Queueing (PRIQ) assigns multiple queues to a network interface with each queue being given a unique priority level. A queue with a higher priority is *always* processed ahead of a queue with a lower priority.

The queueing structure in PRIQ is flat -- you cannot define queues within queues. The root queue is defined, which sets the total amount of bandwidth that is available, and then sub queues are defined under the root. Consider the following example:

Root Queue (2Mbps)
    Queue A (priority 1)
    Queue B (priority 2)
    Queue C (priority 3)

The root queue is defined as having 2Mbps of bandwidth available to it and three subqueues are defined. The queue with the highest priority (the highest priority number) is served first. Once all the packets in that queue are processed, or if the queue is found to be empty, PRIQ moves onto the queue with the next highest priority. Within a given queue, packets are processed in a First In First Out (FIFO) manner.

It is important to note that when using PRIQ you must plan your queues very carefully. Because PRIQ *always* processes a higher priority queue before a lower priority one, it's possible for a high priority queue to cause packets in a lower priority queue to be delayed or dropped if the high priority queue is receiving a constant stream of packets.

Random Early Detection

Random Early Detection (RED) is a congestion avoidance algorithm. Its job is to avoid network congestion by making sure that the queue doesn't become full. It does this by continually calculating the average length (size) of the queue and comparing it to two thresholds, a minimum threshold and a maximum threshold. If the average queue size is below the minimum threshold then no packets will be dropped. If the average is above the maximum threshold then *all* newly arriving packets will be dropped. If the average is between the threshold values then packets are dropped based on a probability calculated from the average queue size. In other words, as the average queue size approaches the maximum threshold, more and more packets are dropped. When dropping packets, RED randomly chooses which connections to drop packets from. Connections using larger amounts of bandwidth have a higher probability of having their packets dropped.

RED is useful because it avoids a situation known as global synchronization and it is able to accommodate bursts of traffic. Global synchronization refers to a loss of total throughput due to packets being dropped from several connections at the same time. For example, if congestion occurs at a router carrying traffic for 10 FTP connections and packets from all (or most) of these connections are dropped (as is the case with FIFO queueing), overall throughput will drop sharply. This isn't an ideal situation because it causes all of the FTP connections to reduce their throughput and also means that the network is no longer being used to its maximum potential. RED avoids this by randomly choosing which connections to drop packets from instead of choosing all of them. Connections using large amounts of bandwidth have a higher chance of their packets being dropped. In this way, high bandwidth connections will be throttled back, congestion will be avoided, and sharp losses of overall throughput will not occur. In addition, RED is able to handle bursts of traffic because it starts to drop packets *before* the queue becomes full. When a burst of traffic comes through there will be enough space in the queue to hold the new packets.

RED should only be used when the transport protocol is capable of responding to congestion indicators from the network. In most cases this means RED should be used to queue TCP traffic and not UDP or ICMP traffic.

For a more detailed look at the theory behind RED, please see References on RED.

Explicit Congestion Notification

Explicit Congestion Notification (ECN) works in conjunction with RED to notify two hosts communicating over the network of any congestion along the communication path. It does this by enabling RED to set a flag in the packet header instead of dropping the packet. Assuming the sending host has support for ECN, it can then read this flag and throttle back its network traffic accordingly.

For more information on ECN, please refer to RFC 3168.

Configuring Queueing

Since OpenBSD 3.0 the Alternate Queueing (ALTQ) queueing implementation has been a part of the base system. Starting with OpenBSD 3.3 ALTQ has been integrated into PF. OpenBSD's ALTQ implementation supports the Class Based Queueing (CBQ) and Priority Queueing (PRIQ) schedulers. It also supports Random Early Detection (RED) and Explicit Congestion Notification (ECN).

Because ALTQ has been merged with PF, PF must be enabled for queueing to work. Instructions on how to enable PF can be found in Getting Started.

Queueing is configured in `pf.conf`. There are two types of directives that are used to configure queueing:

- `altq on` - enables queueing on an interface, defines which scheduler to use, and creates the root queue
- `queue` - defines the properties of a child queue

The syntax for the `altq on` directive is:

```
altq on interface scheduler bandwidth bw qlimit qlim \
```

```
tbrsize size queue { queue_list }
```

- *interface* - the network interface to activate queueing on.
- *scheduler* - the queueing scheduler to use. Possible values are `cbq` and `priq`. Only one scheduler may be active on an interface at a time.
- *bw* - the total amount of bandwidth available to the scheduler. This may be specified as an absolute value using the suffixes `b`, `Kb`, `Mb`, and `Gb` to represent bits, kilobits, megabits, and gigabits per second, respectively or as a percentage of the *interface* bandwidth.
- *qlim* - the maximum number of packets to hold in the queue. This parameter is optional. The default is 50.
- *size* - the size of the token bucket regulator in bytes. If not specified, the size is set based on the *interface* bandwidth.
- *queue_list* - a list of child queues to create under the root queue.

For example:

```
altq on fxp0 cbq bandwidth 2Mb queue { std, ssh, ftp }
```

This enables CBQ on the `fxp0` interface. The total bandwidth available is set to 2Mbps. Three child queues are defined: `std`, `ssh`, and `ftp`.

The syntax for the `queue` directive is:

```
queue name [on interface] bandwidth bw [priority pri] [qlimit qlim] \
   scheduler ( sched_options ) { queue_list }
```

- *name* - the name of the queue. This must match the name of one of the queues defined in the `altq on` directive's *queue_list*. For `cbq` it can also match the name of a queue in a previous `queue` directive's *queue_list*. Queue names must be no longer than 15 characters.
- *interface* - the network interface that the queue is valid on. This value is optional, and when not specified, will make the queue valid on all interfaces.
- *bw* - the total amount of bandwidth available to the queue. This may be specified as an absolute value using the suffixes `b`, `Kb`, `Mb`, and `Gb` to represent bits, kilobits, megabits, and gigabits per second, respectively or as a percentage of the parent queue's bandwidth. This parameter is only applicable when using the `cbq` scheduler.
- *pri* - the priority of the queue. For `cbq` the priority range is 0 to 7 and for `priq` the range is 0 to 15. Priority 0 is the lowest priority. When not specified, a default of 1 is used.
- *qlim* - the maximum number of packets to hold in the queue. When not specified, a default of 50 is used.
- *scheduler* - the scheduler being used, either `cbq` or `priq`. Must be the same as the root queue.
- *sched_options* - further options may be passed to the scheduler to control its behavior:
  - `default` - defines a default queue where all packets not matching any other queue will be queued. Exactly one default queue is required.
  - `red` - enables Random Early Detection (RED) on this queue.
  - `rio` - enables RED with IN/OUT. In this mode, RED will maintain multiple average queue lengths and multiple threshold values, one for each IP Quality of Service level.
  - `ecn` - enables Explicit Congestion Notification (ECN) on this queue. `Ecn` implies `red`.
  - `borrow` - the queue can borrow bandwidth from its parent. This can only be specified when using the `cbq` scheduler.
- *queue_list* - a list of child queues to create under this queue. A *queue_list* may only be defined when using the `cbq` scheduler.

Continuing with the example above:

```
queue std bandwidth 50% cbq(default)
queue ssh { ssh_login, ssh_bulk }
  queue ssh_login priority 4 cbq(ecn)
  queue ssh_bulk cbq(ecn)
queue ftp bandwidth 500Kb priority 3 cbq(borrow red)
```

Here the parameters of the previously defined child queues are set. The `std` queue is assigned a bandwidth of 50% of the root queue's bandwidth (or 1Mbps) and is set as the default queue. The `ssh` queue defines two child queues, `ssh_login` and `ssh_bulk`. The `ssh_login` queue is given a higher priority than `ssh_bulk` and both have ECN enabled. The `ftp` queue is assigned a bandwidth of 500Kbps and given a priority of 3. It can also borrow bandwidth when extra is available and has RED enabled.

Assigning Traffic to a Queue

To assign traffic to a queue, the `queue` keyword is used in conjunction with PF's filter rules. For example, consider a set of filtering rules containing a line such as:

```
pass out on fxp0 from any to any port 22
```

Packets matching that rule can be assigned to a specific queue by using the `queue` keyword:

```
pass out on fxp0 from any to any port 22 queue ssh
```

When using the `queue` keyword with `block` directives, any resulting TCP RST or ICMP Unreachable packets are assigned to the specified queue.

Note that queue designation can happen on an interface other than the one defined in the `altq on` directive:

```
altq on fxp0 cbq bandwidth 2Mb queue { std, ftp }
queue std cbq(default)
queue ftp bandwidth 1.5Mb

pass in on dc0 from any to any port 21 queue ftp
```

Queueing is enabled on `fxp0` but the designation takes place on `dc0`. If packets matching the `pass` rule exit from interface `fxp0`, they will be queued in the `ftp` queue. This type of queueing can be very useful on routers.

Normally only one queue name is given with the `queue` keyword, but if a second name is specified that queue will be used for packets with a Type of Service (ToS) of low-delay and for TCP ACK packets with no data payload. A good example of this is found when using SSH. SSH login sessions will set the ToS to low-delay while SCP and SFTP sessions will not. PF can use this information to queue packets belonging to a login connection in a different queue than non-login connections. This can be useful to prioritize login connection packets over file transfer packets.

```
pass out on fxp0 from any to any port 22 queue(ssh_bulk, ssh_login)
```

This assigns packets belonging to SSH login connections to the `ssh_login` queue and packets belonging to SCP and SFTP connections to the `ssh_bulk` queue. SSH login connections will then have their packets processed ahead of SCP and SFTP connections because the `ssh_login` queue has a higher priority.

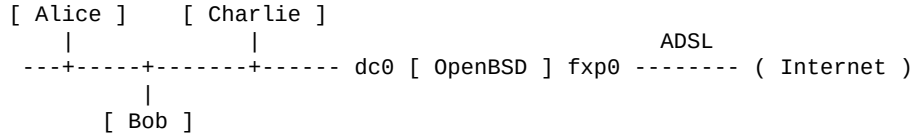
Assigning TCP ACK packets to a higher priority queue is useful on asymmetric connections, that is, connections that have different upload and download bandwidths such as ADSL lines. With an ADSL line, if the upload channel is being maxed out and a download is started, the download will suffer because the TCP ACK packets it needs to send will run into congestion when they try to pass through the upload channel. Testing has shown that to achieve the best results, the bandwidth on the upload queue should be set to a value less than what the connection is capable of. For instance, if an ADSL line has a max upload of 640Kbps, setting the root queue's `bandwidth` to a value such as 600Kb should result in better performance. Trial and error will yield the best `bandwidth` setting.

When using the `queue` keyword with rules that `keep state` such as:

```
pass in on fxp0 proto tcp from any to any port 22 flags S/SA \
   keep state queue ssh
```

PF will record the queue in the state table entry so that packets traveling back out `fxp0` that match the stateful connection will end up in the `ssh` queue. Note that even though the `queue` keyword is being used on a rule filtering incoming traffic, the goal is to specify a queue for the corresponding outgoing traffic; the above rule does not queue incoming packets.

Example #1: Small, Home Network



In this example, OpenBSD is being used on an Internet gateway for a small home network with three workstations. The gateway is performing packet filtering and NAT duties. The Internet connection is via an ADSL line running at 2Mbps down and 640Kbps up.

The queueing policy for this network:

- Reserve 80Kbps of download bandwidth for Bob so he can play his online games without being lagged by Alice or Charlie's downloads. Allow Bob to use more than 80Kbps when it's available.
- Interactive SSH and instant message traffic will have a higher priority than regular traffic.
- DNS queries and replies will have the second highest priority.
- Outgoing TCP ACK packets will have a higher priority than all other outgoing traffic.

Below is the ruleset that meets this network policy. Note that only the `pf.conf` directives that apply directly to the above policy are present; `nat`, `rdr`, options,

etc., are not shown.

```
# enable queueing on the external interface to control traffic going to
# the Internet. use the priq scheduler to control only priorities. set
# the bandwidth to 610Kbps to get the best performance out of the TCP
# ACK queue.

altq on fxp0 priq bandwidth 610Kb queue { std_out, ssh_im_out, dns_out, \
        tcp_ack_out }

# define the parameters for the child queues.
# std_out      - the standard queue. any filter rule below that does not
#                explicitly specify a queue will have its traffic added
#                to this queue.
# ssh_im_out   - interactive SSH and various instant message traffic.
# dns_out      - DNS queries.
# tcp_ack_out  - TCP ACK packets with no data payload.

queue std_out     priq(default)
queue ssh_im_out  priority 4 priq(red)
queue dns_out     priority 5
queue tcp_ack_out priority 6

# enable queueing on the internal interface to control traffic coming in
# from the Internet. use the cbq scheduler to control bandwidth. max
# bandwidth is 2Mbps.

altq on dc0 cbq bandwidth 2Mb queue { std_in, ssh_im_in, dns_in, bob_in }

# define the parameters for the child queues.
# std_in      - the standard queue. any filter rule below that does not
#               explicitly specify a queue will have its traffic added
#               to this queue.
# ssh_im_in   - interactive SSH and various instant message traffic.
# dns_in      - DNS replies.
# bob_in      - bandwidth reserved for Bob's workstation. allow him to
#               borrow.

queue std_in    cbq(default)
queue ssh_im_in priority 4
queue dns_in    priority 5
queue bob_in    bandwidth 80Kb cbq(borrow)


# ... in the filtering section of pf.conf ...

alice         = "192.168.0.2"
bob           = "192.168.0.3"
charlie       = "192.168.0.4"
local_net     = "192.168.0.0/24"
ssh_ports     = "{ 22 2022 }"
im_ports      = "{ 1863 5190 5222 }"

# filter rules for fxp0 inbound
block in on fxp0 all

# filter rules for fxp0 outbound
block out on fxp0 all
pass  out on fxp0 inet proto tcp from (fxp0) to any flags S/SA \
        keep state queue(std_out, tcp_ack_out)
pass  out on fxp0 inet proto { udp icmp } from (fxp0) to any keep state
pass  out on fxp0 inet proto { tcp udp } from (fxp0) to any port domain \
        keep state queue dns_out
pass  out on fxp0 inet proto tcp from (fxp0) to any port $ssh_ports \
        flags S/SA keep state queue(std_out, ssh_im_out)
pass  out on fxp0 inet proto tcp from (fxp0) to any port $im_ports \
        flags S/SA keep state queue(ssh_im_out, tcp_ack_out)

# filter rules for dc0 inbound
block in on dc0 all
pass  in on dc0 from $local_net

# filter rules for dc0 outbound
block out on dc0 all
pass  out on dc0 from any to $local_net
```

```
pass  out on dc0 proto { tcp udp } from any port domain to $local_net \
        queue dns_in
pass  out on dc0 proto tcp from any port $ssh_ports to $local_net \
        queue(std_in, ssh_im_in)
pass  out on dc0 proto tcp from any port $im_ports to $local_net \
        queue ssh_im_in
pass  out on dc0 from any to $bob queue bob_in
```

Example #2: Company Network

In this example, the OpenBSD host is acting as a firewall for a company network. The company runs a WWW server in the DMZ portion of their network where customers upload their websites via FTP. The IT department has their own subnet connected to the main network, and the boss has a PC on his desk that's used for email and surfing the web. The connection to the Internet is via a T1 line running at 1.5Mbps in both directions. All other network segments are using Fast Ethernet (100Mbps).

The network administrator has decided on the following policy:

- Limit traffic between the WWW server and the Internet to 500Kbps in each direction.
- No bandwidth limit on traffic between the WWW server and the internal network.
- Give HTTP traffic between the WWW server and the Internet a higher priority than other traffic between the WWW server and the Internet (such as FTP uploads).
- Reserve 500Kbps for the IT Dept network so they can download the latest software updates in a timely manner. They should be able to use more than 500Kbps when extra bandwidth is available.
- Give traffic between the boss's PC and the Internet a higher priority than other traffic to/from the Internet.

Below is the ruleset that meets this network policy. Note that only the `pf.conf` directives that apply directly to the above policy are present; `nat`, `rdr`, options, etc., are not shown.

```
# enable queueing on the external interface to queue packets going out
# to the Internet. use the cbq scheduler so that the bandwidth use of
# each queue can be controlled. the max outgoing bandwidth is 1.5Mbps.

altq on fxp0 cbq bandwidth 1.5Mb queue { std_ext, www_ext, boss_ext }

# define the parameters for the child queues.
# std_ext        - the standard queue. also the default queue for
#                  outgoing traffic on fxp0.
# www_ext        - container queue for WWW server queues. limit to
#                  500Kbps.
#   www_ext_http - http traffic from the WWW server
#   www_ext_misc - all non-http traffic from the WWW server
# boss_ext       - traffic coming from the boss's computer

queue std_ext        cbq(default)
queue www_ext        bandwidth 500Kb { www_ext_http, www_ext_misc }
  queue www_ext_http priority 3 cbq(red)
  queue www_ext_misc priority 1
queue boss_ext       priority 3

# enable queueing on the internal interface to control traffic coming
# from the Internet or the DMZ. use the cbq scheduler to control the
# bandwidth of each queue. bandwidth on this interface is set to the
# maximum. traffic coming from the DMZ will be able to use all of this
# bandwidth while traffic coming from the Internet will be limited to
# 1.0Mbps (because 0.5Mbps (500Kbps) is being allocated to fxp1).

altq on dc0 cbq bandwidth 100% queue { net_int, www_int }
```

```
# define the parameters for the child queues.
# net_int    - container queue for traffic from the Internet. bandwidth
#              is 1.0Mbps.
#   std_int  - the standard queue. also the default queue for outgoing
#              traffic on dc0.
#   it_int   - traffic to the IT Dept network.
#   boss_int - traffic to the boss's PC.
# www_int    - traffic from the WWW server in the DMZ.

queue net_int    bandwidth 1.0Mb { std_int, it_int, boss_int }
  queue std_int  cbq(default)
  queue it_int   bandwidth 500Kb cbq(borrow)
  queue boss_int priority 3
queue www_int    cbq(red)

# enable queueing on the DMZ interface to control traffic destined for
# the WWW server. cbq will be used on this interface since detailed
# control of bandwidth is necessary. bandwidth on this interface is set
# to the maximum. traffic from the internal network will be able to use
# all of this bandwidth while traffic from the Internet will be limited
# to 500Kbps.

altq on fxp1 cbq bandwidth 100% queue { internal_dmz, net_dmz }

# define the parameters for the child queues.
# internal_dmz   - traffic from the internal network.
# net_dmz        - container queue for traffic from the Internet.
#   net_dmz_http - http traffic.
#   net_dmz_misc - all non-http traffic. this is also the default queue.

queue internal_dmz      # no special settings needed
queue net_dmz        bandwidth 500Kb { net_dmz_http, net_dmz_misc }
  queue net_dmz_http priority 3 cbq(red)
  queue net_dmz_misc priority 1 cbq(default)


# ... in the filtering section of pf.conf ...

main_net  = "192.168.0.0/24"
it_net    = "192.168.1.0/24"
int_nets  = "{ 192.168.0.0/24, 192.168.1.0/24 }"
dmz_net   = "10.0.0.0/24"

boss      = "192.168.0.200"
wwwserv   = "10.0.0.100"

# default deny
block on { fxp0, fxp1, dc0 } all

# filter rules for fxp0 inbound
pass in on fxp0 proto tcp from any to $wwwserv port { 21, \
        > 49151 } flags S/SA keep state queue www_ext_misc
pass in on fxp0 proto tcp from any to $wwwserv port 80 \
        flags S/SA keep state queue www_ext_http

# filter rules for fxp0 outbound
pass out on fxp0 from $int_nets to any keep state
pass out on fxp0 from $boss to any keep state queue boss_ext

# filter rules for dc0 inbound
pass in on dc0 from $int_nets to any keep state
pass in on dc0 from $it_net to any queue it_int
pass in on dc0 from $boss to any queue boss_int
pass in on dc0 proto tcp from $int_nets to $wwwserv port { 21, 80, \
        > 49151 } flags S/SA keep state queue www_int

# filter rules for dc0 outbound
pass out on dc0 from dc0 to $int_nets

# filter rules for fxp1 inbound
pass in on fxp1 proto { tcp, udp } from $wwwserv to any port 53 \
        keep state

# filter rules for fxp1 outbound
pass out on fxp1 proto tcp from any to $wwwserv port { 21, \
```

```
        > 49151 } flags S/SA keep state queue net_dmz_misc
pass out on fxp1 proto tcp from any to $wwwserv port 80 \
        flags S/SA keep state queue net_dmz_http
pass out on fxp1 proto tcp from $int_nets to $wwwserv port { 80, \
        21, > 49151 } flags S/SA keep state queue internal_dmz
```

[Previous: Anchors and Named (Sub) Rulesets] [Contents] [Next: Address Pools and Load Balancing]


www@openbsd.org

$OpenBSD: queueing.html,v 1.18 2004/01/19 00:21:29 nick Exp $

OpenBSD

[Previous: Packet Queueing and Prioritization] [Contents] [Next: Packet Tagging]

PF: Address Pools and Load Balancing

Table of Contents

Introduction

An address pool is a supply of two or more addresses whose use is shared among a group of users. An address pool can be specified as the redirection address in `rdr` rules, as the translation address in `nat` rules, and as the target address in `route-to`, `reply-to`, and `dup-to` filter options.

There are four methods for using an address pool:

- `bitmask` - grafts the network portion of the pool address over top of the address that is being modified (source address for `nat` rules, destination address for `rdr` rules). Example: if the address pool is 192.0.2.1/24 and the address being modified is 10.0.0.50, then the resulting address will be 192.0.2.50. If the address pool is 192.0.2.1/25 and the address being modified is 10.0.0.130, then the resulting address will be 192.0.2.2.
- `random` - randomly selects an address from the pool.
- `source-hash` - uses a hash of the source address to determine which address to use from the pool. This method ensures that a given source address is always mapped to the same pool address. The key that is fed to the hashing algorithm can optionally be specified after the `source-hash` keyword in hex format or as a string. By default, pfctl(8) will generate a random key every time the ruleset is loaded.
- `round-robin` - loops through the address pool in sequence. This is the default method and also the only method allowed when the address pool is specified using a table.

Except for the `round-robin` method, the address pool must be expressed as a CIDR (Classless Inter-Domain Routing) network block. The `round-robin` method will accept multiple individual addresses using a list or table.

NAT Address Pool

An address pool can be used as the translation address in `nat` rules. Connections will have their source address translated to an address from the pool based on the method chosen. This can be useful in situations where PF is performing NAT for a very large network. Since the number of NATed connections per translation address is limited, adding additional translation addresses will allow the NAT gateway to scale to serve a larger number of users.

In this example a pool of two addresses is being used to translate outgoing packets. For each outgoing connection PF will rotate through the addresses in a round-robin manner.

```
nat on $ext_if inet from any to any -> { 192.0.2.5, 192.0.2.10 }
```

One drawback with this method is that successive connections from the same internal address will not always be translated to the same translation address. This can cause interference, for example, when browsing websites that track user logins based on IP address. An alternate approach is to use the `source-hash` method so that each internal address is always translated to the same translation address. To do this, the address pool must be a CIDR network block.

```
nat on $ext_if inet from any to any -> 192.0.2.4/31 source-hash
```

This `nat` rule uses the address pool 192.0.2.4/31 (192.0.2.4 - 192.0.2.5) as the translation address for outgoing packets. Each internal address will always be translated to the same translation address because of the `source-hash` keyword.

Load Balance Incoming Connections

Address pools can also be used to load balance incoming connections. For example, incoming web server connections can be distributed across a web server farm:

```
web_servers = "{ 10.0.0.10, 10.0.0.11, 10.0.0.13 }"

rdr on $ext_if proto tcp from any to any port 80 -> $web_servers
```

Successive connections will be redirected to the web servers in a round-robin manner.

As with the NAT example, if the web servers are all placed within a CIDR network block, the `source-hash` keyword can be used so that connections from a given IP address are always redirected to the same physical web server. Again, this is sometimes necessary to maintain session information while browsing a website.

Load Balance Outgoing Traffic

Address pools can be used in combination with the `route-to` filter option to load balance two or more Internet connections when a proper multi-path routing protocol (like BGP4) is unavailable. By using `route-to` with a `round-robin` address pool, outbound connections can be evenly distributed among multiple outbound paths.

One additional piece of information that's needed to do this is the IP address of the adjacent router on each Internet connection. This is fed to the `route-to` option to control the destination of outgoing packets.

The following example balances outgoing traffic across two Internet connections:

```
lan_net = "192.168.0.0/24"
int_if  = "dc0"
ext_if1 = "fxp0"
ext_if2 = "fxp1"
ext_gw1 = "68.146.224.1"
ext_gw2 = "142.59.76.1"

pass in on $int_if route-to \
   { ($ext_if1 $ext_gw1), ($ext_if2 $ext_gw2) } round-robin \
   from $lan_net to any keep state
```

The `route-to` option is used on traffic coming *in* on the *internal* interface to specify the outgoing network interfaces that traffic will be balanced across along with their respective gateways. Note that the `route-to` option must be present on *each* filter rule that traffic is to be balanced for. Return packets will be routed back to the same external interface that they exited (this is done by the ISPs) and will be routed back to the internal network normally.

To ensure that packets with a source address belonging to `$ext_if1` are always routed to `$ext_gw1` (and similarly for `$ext_if2` and `$ext_gw2`), the following two lines should be included in the ruleset:

```
pass out on $ext_if1 route-to ($ext_if2 $ext_gw2) from $ext_if2 \
   to any
pass out on $ext_if2 route-to ($ext_if1 $ext_gw1) from $ext_if1 \
   to any
```

Finally, NAT can also be used on each outgoing interface:

```
nat on $ext_if1 from $lan_net to any -> ($ext_if1)
nat on $ext_if2 from $lan_net to any -> ($ext_if2)
```

A complete example that load balances outgoing traffic might look something like this:

```
lan_net = "192.168.0.0/24"
int_if  = "dc0"
ext_if1 = "fxp0"
ext_if2 = "fxp1"
ext_gw1 = "68.146.224.1"
ext_gw2 = "142.59.76.1"

#  nat outgoing connections on each internet interface
nat on $ext_if1 from $lan_net to any -> ($ext_if1)
nat on $ext_if2 from $lan_net to any -> ($ext_if2)

#  default deny
block in  from any to any
block out from any to any

#  pass all outgoing packets on internal interface
pass out on $int_if from any to $lan_net
#  pass in quick any packets destined for the gateway itself
pass in quick on $int_if from $lan_net to $int_if
#  load balance outgoing tcp traffic from internal network.
pass in on $int_if route-to \
    { ($ext_if1 $ext_gw1), ($ext_if2 $ext_gw2) } round-robin \
    proto tcp from $lan_net to any flags S/SA modulate state
#  load balance outgoing udp and icmp traffic from internal network
pass in on $int_if route-to \
    { ($ext_if1 $ext_gw1), ($ext_if2 $ext_gw2) } round-robin \
    proto { udp, icmp } from $lan_net to any keep state

#  general "pass out" rules for external interfaces
pass out on $ext_if1 proto tcp from any to any flags S/SA modulate state
pass out on $ext_if1 proto { udp, icmp } from any to any keep state
pass out on $ext_if2 proto tcp from any to any flags S/SA modulate state
pass out on $ext_if2 proto { udp, icmp } from any to any keep state

#  route packets from any IPs on $ext_if1 to $ext_gw1 and the same for
#  $ext_if2 and $ext_gw2
pass out on $ext_if1 route-to ($ext_if2 $ext_gw2) from $ext_if2 to any
pass out on $ext_if2 route-to ($ext_if1 $ext_gw1) from $ext_if1 to any
```

[Previous: Packet Queueing and Prioritization] [Contents] [Next: Packet Tagging]


www@openbsd.org

$OpenBSD: pools.html,v 1.10 2004/01/01 04:16:17 nick Exp $

[Précédent : Ensembles d'Adresses ("Pools") et Partage de Charge] [Index] [Suivant : Journal des Evénements]

PF: Balisage de Paquets

Table des Matières

Introduction

Le balisage de paquets est une méthode pour marquer les paquets avec un identifiant interne qui peut être utilisé comme critère dans les règles de filtrage et de traduction d'adresses. Grâce au balisage, il est possible de créer des paquets dits "de confiance" entre des interfaces et de déterminer si des paquets ont été traités par les règles de traduction d'adresses. Il est aussi possible de faire du filtrage suivant une politique au lieu de faire du filtrage par règle.

Affectation de Balises aux Paquets

Pour ajouter une balise à un paquet, utilisez le mot-clé `tag` :

```
pass in on $int_if all tag INTERNAL_NET keep state
```

La balise `INTERNAL_NET` sera ajoutée à tout paquet qui correspondra à la règle précitée. Il est à noter que le mot-clé `keep state` : `keep state` (ou `modulate state`/`synproxy state`) doit être utilisé dans les règles `pass`qui affectent des balises aux paquets.

L'affectation de balises observe les règles suivantes :

- Les balises sont adhérentes ("sticky"). Une fois une balise est appliquée à un paquet par une règle correspondante, elle n'est jamais supprimée. Cependant, elle peut être remplacée par une balise différente.
- A cause de cette forte adhérence d'une balise, un paquet peut avoir une balise même si la dernière règle correspondante n'utilise pas le mot-clé `tag`.
- Un paquet ne peut avoir qu'une seule balise à un moment donné.
- Les balises sont des identifiants *internal*. Elles ne sont pas envoyées sur le réseau.

Prenons le jeu de règles suivant comme exemple :

```
(1) pass in on $int_if tag INT_NET keep state
(2) pass in quick on $int_if proto tcp to port 80 tag \
        INT_NET_HTTP keep state
(3) pass in quick on $int_if from 192.168.1.5 keep state
```

- Les paquets arrivant sur l'interface `$int_if` se verront attribuer une balise `INT_NET` par la règle #1.
- Les paquets TCP arrivant sur l'interface `$int_if` et destinés au port 80 se verront d'abord attribuer une balise `INT_NET` par la règle #1. Cette balise sera ensuite remplacée par la balise `INT_NET_HTTP` à cause de la règle #2.
- Les paquets arrivant sur l'interface `$int_if` et en provenance de 192.168.1.5 seront autorisés par la règle #3 car c'est la dernière règle qui correspond au filtrage de ces paquets. Cependant, ces paquets se verront attribuer la balise `INT_NET_HTTP` s'ils sont destinés au port TCP 80, sinon ils se verront attribuer la balise `INT_NET`.

De même que pour les règles de filtrage, les balises peuvent être appliquées par des règles `nat`, `rdr`, et `binat` en utilisant le mot-clé `tag`.

Vérification des Balises Appliquées

Pour vérifier les balises précédemment appliquées, utilisez le mot-clé `tagged` comme dans l'exemple suivant :

```
pass out on $ext_if tagged INT_NET keep state
```

Les paquets sortant à partir de `$ext_if`doivent être balisés avec la balise `INT_NET` pour que la règle ci-dessus corresponde à ces paquets. La correspondance inverse peut aussi être réalisée avec l'opérateur `!` :

```
pass out on $ext_if tagged ! WIFI_NET keep state
```

Filtrage par Politique

Le filtrage par politique utilise une approche différente pour l'écriture d'un jeu de règles. Une politique est définie par rapport aux types de trafic : règles pour les types de trafic à passer, règles pour les types de trafic à bloquer. Les paquets sont ensuite classifiés au sein de la politique selon les critères traditionnels : adresse IP source/destination, protocole, etc. Examinez la politique de filtrage qui suit :

- Le trafic provenant du LAN interne et à destination de la DMZ est autorisé (LAN_DMZ)
- Le trafic provenant d'Internet et à destination de serveurs dans la DMZ est autorisé (INET_DMZ)
- Le trafic provenant d'Internet et redirigé vers spamd(8) est autorisé (SPAMD)
- Tout autre trafic est bloqué

Notez la manière dont la politique couvre *all* le trafic qui transite par le pare-feu. Le mot entre parenthèses indique le nom de la balise qui sera utilisée pour chaque élément de la politique.

Des règles de filtrage et de traduction doivent être écrites maintenant pour classifier les paquets au sein de la politique.

```
rdr on $ext_if proto tcp from <spamd> to port smtp \
   tag SPAMD -> 127.0.0.1 port 8025

block all
pass in on $int_if from $int_net tag LAN_INET keep state
pass in on $int_if from $int_net to $dmz_net tag LAN_DMZ keep state
pass in on $ext_if proto tcp to $www_server port 80 tag INET_DMZ keep state
```

Maintenant les règles qui constituent la politique sont définies.

```
pass in  quick on $ext_if tagged SPAMD keep state
pass out quick on $ext_if tagged LAN_INET keep state
pass out quick on $dmz_if tagged LAN_DMZ keep state
pass out quick on $dmz_if tagged INET_DMZ keep state
```

Maintenant que le jeu de règles a été paramétré, les modifications futures sont à apporter uniquement dans les règles de classifications. Par exemple, si un serveur POP3/SMTP est ajouté à la DMZ, il sera nécessaire d'ajouter des règles de classification pour le trafic POP3 et SMTP comme le montre l'exemple suivant :

```
mail_server = "192.168.0.10"
...
pass in on $ext_if proto tcp to $mail_server port { smtp, pop3 } \
   tag INET_DMZ keep state
```

Le trafic mail sera autorisé car il fait partie de la classification INET_DMZ.

Voici le jeu de règles complet :

```
# macros
int_if  = "dc0"
dmz_if  = "dc1"
ext_if  = "ep0"
int_net = "10.0.0.0/24"
dmz_net = "192.168.0.0/24"
www_server = "192.168.0.5"
mail_server = "192.168.0.10"

table <spamd> persist file "/etc/spammers"

# classification -- classifier les paquets selon la politique
# définie
rdr on $ext_if proto tcp from <spamd> to port smtp \
    tag SPAMD -> 127.0.0.1 port 8025

block all
pass in on $int_if from $int_net tag LAN_INET keep state
pass in on $int_if from $int_net to $dmz_net tag LAN_DMZ keep state
pass in on $ext_if proto tcp to $www_server port 80 tag INET_DMZ keep state
pass in on $ext_if proto tcp to $mail_server port { smtp, pop3 } \
    tag INET_DMZ keep state

# filtrage -- autoriser/bloquer suivant la politique.
pass in  quick on $ext_if tagged SPAMD keep state
pass out quick on $ext_if tagged LAN_INET keep state
pass out quick on $dmz_if tagged LAN_DMZ keep state
pass out quick on $dmz_if tagged INET_DMZ keep state
```

Balisage des Trames Ethernet

Le balisage peut être effectué au niveau Ethernet si la machine de balisage/filtrage est aussi un pont (bridge(4)). En créant des règles de filtrage pour bridge(4) qui utilisent le mot-clé `tag`, PF peut filtrer les paquets d'après leur adresse MAC source/destination. Les règles pour bridge(4) sont créés avec la commande brconfig(8). Exemple :

```
# brconfig bridge0 rule pass in on fxp0 src 0:de:ad:be:ef:0 \
   tag USER1
```

Puis dans `pf.conf`:

```
pass in on fxp0 tagged USER1
```

[Précédent : Ensembles d'Adresses ("Pools") et Partage de Charge] [Index] [Suivant : Journal des Evénements]


www@openbsd.org

Originally [OpenBSD: tagging.html,v 1.2]
$Translation: tagging.html,v 1.2 2004/01/02 22:10:05 saad Exp $
$OpenBSD: tagging.html,v 1.2 2004/01/03 19:45:23 jufi Exp $

OpenBSD

[Précédent : Balisage de Paquets] [Index] [Suivant : Performances]

PF: Journal des Evénements

Table des Matières

Introduction

L'enregistrement des Evénements dans PF est effectué par pflogd(8) qui écoute sur l'interface pflog0 et écrit les paquets dans un fichier d'événements (normalement /var/log/pflog) au format binaire tcpdump(8). Les flux correspondant aux règles de filtrage qui contiennent les mots-clés `log` ou `log-all` sont enregistrés de cette manière.

Lire un Fichier d'Evénements

Le fichier d'événements écrit par pflogd est dans un format binaire que seul tcpdump(8) ou tout outil connaissant le format binaire tcpdump peut lire.

Pour lire le fichier d'événements :

```
# tcpdump -n -e -ttt -r /var/log/pflog
```

Il est à noter que l'utilisation de tcpdump(8) pour lire le fichier pflog ne donne *pas* un affichage en temps réel. Cependant, il est possible d'obtenir un tel affichage en utilisant l'interface pflog0 :

```
# tcpdump -n -e -ttt -i pflog0
```

REMARQUE : Lorsque vous examinez les événements, une attention particulière doit être prise par rapport au décodage verbeux des protocoles effectué par tcpdump (décodage activé par l'option `-v`). Les décodeurs protocolaires de tcpdump ne possèdent pas un historique sécurité parfait. En théorie du moins, une attaque retardée peut être possible à travers les charges utiles partielles enregistrées par le périphérique d'enregistrement d'événements. Il est recommandé de déplacer les fichiers d'événements du pare-feu avant de procéder à leur examen de cette manière.

Des précautions supplémentaires doivent êtres prises pour sécuriser l'accès aux événements. Par défaut, pflogd enregistrera 96 octets du paquet dans le fichier d'événements. L'accès aux fichiers d'événements pourrait fournir un accès partiel à des charges

utiles de paquets sensibles (tels que les noms d'utilisateurs et les mots de passe telnet(1) ou ftp(1)).

Filtrer les Evénements en Sortie

Etant donné que pflogd enregistre les événements au format binaire tcpdump, la totalité des fonctionnalités tcpdump peut être utilisée pour analyser les événements. Par exemple, pour voir uniquement les paquets qui correspondent à un certain port :

```
# tcpdump -n -e -ttt -r /var/log/pflog port 80
```

Ceci peut être encore affiné en limitant l'affichage des paquets à une certaine combinaison de hôte et de port :

```
# tcpdump -n -e -ttt -r /var/log/pflog port 80 and host 192.168.1.3
```

La même idée peut être appliquée quand on lit les événéments directement à partir de l'interface pflog0 :

```
# tcpdump -n -e -ttt -i pflog0 host 192.168.4.2
```

Il est à noter que ces manipulations n'ont aucun impact sur les paquets enregistrés dans le fichier d'événements pflogd; les commandes ci-dessus ne feront qu'afficher les paquets au fur et à mesure qu'ils sont enregistrés.

En plus de l'utilisation des règles de filtrage standard tcpdump(8), le langage de filtrage du tcpdump fourni avec OpenBSD a été étendu pour lire la sortie de pflogd :

- `ip` - la famille d'adresses est IPv4.
- `ip6` - la famille d'adresses est IPv6.
- `on` *`int`* - le paquet est passé à travers l'interface *int*.
- `ifname` *`int`* - idem.
- `rulenum` *`num`* - la règle de filtrage à laquelle correspondait le paquet était la règle numéro *num*.
- `action` *`act`* - l'action prise pour le paquet. Les actions possibles sont `pass` et `block`.
- `reason` *`res`* - la raison pour laquelle l'`action` a été prise. Les raisons possibles sont `match`, `bad-offset`, `fragment`, `short`, `normalize`, et `memory`.
- `inbound` - le paquet était entrant.
- `outbound` - le paquet était sortant.

Exemple :

```
# tcpdump -n -e -ttt -i pflog0 inbound and action block and on wi0
```

Cette commande affichera l'événement en temps réel crée par des paquets entrants bloqués sur l'interface wi0.

Enregistrement des Evénements à Travers Syslog

Dans plusieurs situations, il est souhaitable de disposer des événements enregistrés par le pare-feu sous format ASCII ou de les envoyer à un serveur d'événements central distant. Tout ceci peut être effectué par deux petits scripts shell, quelques modifications mineures aux fichiers de configuration OpenBSD, et syslogd(8), le service qui permet d'enregistrer les événements. Syslogd enregistre les événements sous format ASCII et il est aussi capable de les envoyer à un serveur d'événements distant.

Il faut d'abord créer un utilisateur, `pflogger`, avec un shell `/sbin/nologin`. La façon la plus simple de créer cet utilisateur consiste à utiliser la commande adduser(8).

Après avoir crée l'utilisateur `pflogger`, créez les deux scripts suivants :

`/etc/pflogrotate`

```
FILE=/home/pflogger/pflog5min.$(date "+%Y%m%d%H%M")
kill -ALRM $(cat /var/run/pflogd.pid)
if [ $(ls -l /var/log/pflog | cut -d " " -f 8) -gt 24 ]; then
   mv /var/log/pflog $FILE
   chown pflogger $FILE
   kill -HUP $(cat /var/run/pflogd.pid)
fi
```

`/home/pflogger/pfl2sysl`

```
for logfile in /home/pflogger/pflog5min* ; do
   tcpdump -n -e -ttt -r $logfile | logger -t pf -p local0.info
   rm $logfile
done
```

Editez la liste des tâches cron du super-utilisateur root :

```
# crontab -u root -e
```

Ajoutez-y les deux lignes suivantes :

```
# effectuer une rotation du fichier d'événements
# toutes les 5 minutes
0-59/5 * * * * /bin/sh /etc/pflogrotate
```

Créer une liste de tâches cron pour l'utilisateur `pflogger` :

```
# crontab -u pflogger -e
```

Ajoutez-y les deux lignes suivantes :

```
# alimenter le(s) fichier(s) pflog après rotation à
# syslog
0-59/5 * * * * /bin/sh /home/pflogger/pfl2sysl
```

Ajoutez la ligne suivante au fichier `/etc/syslog.conf` :

```
local0.info     /var/log/pflog.txt
```

Si vous voulez les envoyer aussi à un serveur d'événements distant, ajoutez la ligne suivante

```
local0.info     @syslogger
```

Assurez vous que le hôte *syslogger* a bien été défini dans le fichier hosts(5).

Créez le fichier `/var/log/pflog.txt` afin de permettre à syslog d'enregistrer les événements dans ce fichier.

```
# touch /var/log/pflog.txt
```

Relancez syslogd pour que les modifications soient prises en compte :

```
# kill -HUP $(cat /var/run/syslog.pid)
```

Tous les événements enregistrés sont maintenant envoyés vers `/var/log/pflog.txt`. Si la seconde ligne a été ajoutée, les événements sont aussi envoyés au serveur d'événements distant *sysloger*.

Le script `/etc/pflogrotate` traite désormais `/var/log/pflog` et le supprime à la fin du traitement. La rotation de ce fichier par newsyslog(8) n'est plus nécessaire désormais et devrait être désactivée. Cependant `/var/log/pflog.txt` remplace `/var/log/pflog` et sa rotation devrait être activée. Modifiez `/etc/newsyslog.conf` comme suit :

```
#/var/log/pflog        600    3    250    *    ZB /var/run/pflogd.pid
/var/log/pflog.txt     600    7    *      24
```

PF, à l'aide du mécanisme précité, générera des événements qui seront enregistrés dans le fichier `/var/log/pflog.txt`. Ces événements pourront aussi être envoyés à un serveur d'événements distant. La conversion en ASCII n'est pas immédiate. Elle prend jusqu'à 5-6 minutes (l'intervalle d'exécution de la tâche cron) avant que les événements enregistrés apparaissent dans le fichier.

[Précédent : Balisage de Paquets] [Index] [Suivant : Performances]


www@openbsd.org

Originally [OpenBSD: logging.html,v 1.13]
$Translation: logging.html,v 1.3 2004/02/19 20:31:29 saad Exp $
$OpenBSD: logging.html,v 1.3 2004/02/20 06:33:11 jufi Exp $

OpenBSD

[Précédent : Journal des Evénements] [Index] [Suivant :]

PF : Performances

"Combien de bande passante PF peut-il gérer ?" "Quelle puissance machine est suffisante pour gérer mon accès Internet ?" "How much computer do I need to handle my Internet connection?"

Il n'y a pas de réponse facile à ces questions. Pour certaines applications, un 486/66 avec une paire de bonnes cartes réseau ISA pourrait filtrer et faire de la NAT à quelques 5Mbps, mais d'autres applications une machine beaucoup plus puissante avec des cartes réseau PCI plus efficaces pourrait être insuffisante. La vraie question n'est pas le nombre de bits par seconde mais plutôt le nombre de paquets par seconde et la complexité des jeux de règles.

Les performance de PF sont déterminées par plusieurs variables :

- Nombre de paquets par seconde. Pratiquement le même traitement doit être effectué sur une paquet avec une charge utile de 1400 octets que sur un paquet avec une charge utile d'un octet. Le nombre de paquets par seconde détermine le nombre de fois que la table d'état et, au cas où il n'y a pas de correspondance, les règles de filtrage doivent être évaluées chaque seconde, déterminant ainsi à quel point on sollicite le système.
- Performance du bus système. Le bus ISA a une bande passante maximale de 8MB/sec, et lorsque le processeur y accède, il doit diminuer sa vitesse à la vitesse effective d'un 80286 cadencé à 8MHz, peu importe la vitesse réelle du processeur. Le bus PCI a une bande passante effective beaucoup plus grande et a moins d'impact sur le processeur.
- Efficacité de la carte réseau. Certaines cartes réseau sont plus efficaces que d'autres. Les cartes basées sur le Realtek 8139 (rl(4)) ont tendance a avoir de faibles performances alors que les cartes basées sur l'Intel 21143 (dc(4)) ont tendance à avoir d'excellentes performances. Pour des performances maximales, utilisez des cartes Ethernet gigabit même si vous ne vous connectez pas à des réseaux gigabit. Ces cartes ont une gestion de la mémoire tampon beaucoup plus avancée que les autres cartes.
- Complexité et conception de votre jeu de règles. Plus le jeu de règles est complexe, plus lent le pare-feu sera. Plus vous utiliserez des règles `keep state` et `quick` pour filtrer les paquets, meilleures les performances seront. Plus le nombre de lignes devant être évaluées pour chaque paquet, plus faibles seront les performances.
- Mentionnons quand même deux éléments : le CPU et la RAM. Vu que PF est un processus noyau, il n'utilisera pas d'espace de pagination (swap). Donc si vous avez suffisamment de RAM, il fonctionne sinon il se met en mode panique à cause de l'épuisement de pool(9) exhaustion. Des quantités énormes de RAM ne sont guère nécessaires. 32MB devraient permettre de contenir 30 000 états ce qui un nombre énorme pour des applications SoHo. La plupart des utilisateurs trouveront qu'une machine "recyclée" est plus que suffisante pour un système PF. Un système cadendé à 300MHz pourra gérer un très grand nombre de paquets rapidement, dans la mesure où il a des bonnes cartes réseau et un bon jeu de règles.

Les utilisateurs demandent souvent des benchmarks PF. Le seul benchmark qui compte vraiment est la performance de *votre* système dans *votre* environnement. Un benchmark qui ne réplique pas votre environnement ne vous aidera pas à planifier correctement votre système pare-feu. La meilleure action possible est de tester PF vous-même dans les mêmes conditions réseau ou du moins dans les conditions les plus proches possibles que votre pare-feu actuel sur le même matériel.

PF est utilisé dans certaines applications très conséquentes et à haut trafic, et les développeurs sont des "power users" de PF. Il y a donc beaucoup de chance pour que PF fonctionne dans votre environnement avec des performances correctes.

[Précédent : Journal des Evénements] [Index] [Suivant : Gestion du Protocole FTP]


www@openbsd.org

Originally [OpenBSD: perf.html,v 1.12]
$Translation: perf.html,v 1.6 2004/04/07 09:07:17 xsa Exp $
$OpenBSD: perf.html,v 1.6 2004/04/10 11:40:51 jufi Exp $

[Précédent : Performances] [Index] [Suivant : Authpf : Shell Utilisateur pour les Passerelles d'Authentification]

PF: Gestion du Protocole FTP

Table des Matières

Modes FTP

FTP est un protocole qui date d'une époque où Internet était un petit ensemble de machines avec des relations de confiance mutuelle et où tout le monde connaissait tout le monde. A cette époque le besoin de filtrer ou d'assurer une sécurité stricte était inexistant. FTP n'était pas conçu pour filtrer, pour faire transiter du trafic à travers les pare-feux, ou pour fonctionner avec la NAT.

FTP peut fonctionner dans deux modes : passif et actif. De manière générale, le choix du mode actif ou passif consiste à déterminer l'extrêmité de communication FTP qui a un problème avec le filtrage pare-feu. Dans le meilleur des mondes, il faudrait supporter les deux modes pour rendre vos utilisateurs heureux.

Dans le mode FTP actif, quand un utilisateur se connecte à un serveur FTP distant et demande une information ou un fichier, le serveur FTP effectue une connexion vers le client pour transférer les données demandées. Cette connexion est appelée la *connexion de données*. Pour commencer, le client FTP choisit un port aléatoire qui lui servira pour recevoir les données. Le client envoie le numéro de port qu'il a choisi au serveur FTP puis se met à l'écoute des connexions entrantes à destination de ce port. Le serveur FTP initie ensuite une connexion vers l'adresse du client sur le port choisi et transfère les données. Ce mode de fonctionnement est problématique pour les utilisateurs essayant d'accéder à des serveurs FTP lorsqu'ils sont derrière une passerelle NAT. Vu la manière dont fonctionne la NAT, le serveur FTP initie la connexion de données en se connectant à l'adresse externe de la passerelle NAT sur le port choisi. La passerelle NAT n'a pas de correspondance pour le paquet reçu dans sa table d'état. Le paquet sera ignoré et ne sera pas délivré au client.

Dans le mode FTP passif (le mode par défaut utilisé par le client ftp(1) d'OpenBSD), le client demande au serveur de choisir un port aléatoire sur lequel ce dernier se mettra à l'écoute en attente de la connexion de données. Le serveur communique au client le port qu'il a choisi pour le transfert des données. Malheureusement, ce n'est pas toujours possible ou souhaitable à cause de la possibilité d'existence d'un pare-feu devant le serveur FTP qui bloquerait les connexions de données en entrée. Le client ftp(1) d'OpenBSD utilise le mode passif par défaut; pour forcer le mode FTP actif, utilisez le drapeau -A du client ftp(1). Il est aussi possible de désactiver le mode passif (et donc d'activer par cette action le mode actif) en utilisant la commande "`passive off`" à l'invite de commandes "`ftp>`".

Client FTP derrière un pare-feu

Comme précédemment indiqué, FTP ne fonctionne pas très bien à travers des mécanismes de NAT et des pare-feux.

Packet Filter fournit une solution à ce cas en redirigeant le trafic FTP à travers un serveur mandataire FTP (proxy). Ce processus agit de telle façon à "guider" votre trafic FTP à travers la passerelle NAT/pare-feu. Le mandataire FTP utilisé par OpenBSD et PF est ftp-proxy(8). Pour l'activer, ajustez la ligne suivante selon vos besoins et ajoutez la à la section NAT de `pf.conf` :

```
rdr on $int_if proto tcp from any to any port 21 -> 127.0.0.1 \
   port 8021
```

Voici l'explication de cette ligne : "Le trafic entrant sur l'interface interne est redirigé vers le serveur mandataire sur cette machine et écoutant sur le port 8021".

Il est évident que le serveur mandataire doit être démarré et exécuté sur la machine OpenBSD. Ceci peut être effectué en insérant la ligne suivante dans `/etc/inetd.conf` :

```
127.0.0.1:8021 stream tcp nowait root /usr/libexec/ftp-proxy \
   ftp-proxy
```

Il faut ensuite redémarrer le système ou envoyer le signal 'HUP' à inetd(8). Une manière d'envoyer le signal 'HUP' consiste à utiliser la commande suivante :

```
kill -HUP `cat /var/run/inetd.pid`
```

Notez que ftp-proxy écoute sur le serveur 8031, le même port qui est utilisé dans la ligne `rdr`. Le choix du port 8021 est arbitraire, bien que 8021 est un bon choix vu qu'il n'est utilisé par aucune autre application.

Veuillez noter que ftp-proxy(8) est utilisé pour aider les **clients FTP** derrière un pare-feu PF; il n'est pas utilisé pour prendre en charge un **serveur FTP** derrière un tel pare-feu.

Serveur FTP protégé par un pare-feu PF local

Dans ce cas, PF est activé sur le serveur FTP lui-même plutôt que sur un pare-feu dédié. Pour les besoins d'une connexion FTP passive, FTP utilisera un haut port TCP choisi de manière aléatoire pour les données entrantes. Par défaut, le serveur FTP natif d'OpenBSD ftpd(8) utilise la plage 49152 à 65535. Ces ports doivent être autorisés en entrée ainsi que le port 21 (le port de contrôle FTP) :

```
pass in on $ext_if proto tcp from any to any port 21 keep state
pass in on $ext_if proto tcp from any to any port > 49151 \
   keep state
```

Si vous le souhaitez, vous pouvez restreindre cette plage considérablement. Dans le cas du programme ftpd(8) d'OpenBSD, ceci peut être effectué en utilisant les variables sysctl(8) `net.inet.ip.porthifirst` et `net.inet.ip.porthilast`.

Serveur FTP protégé par un pare-feu PF externe avec NAT

Dans ce cas, le pare-feu doit rediriger tout le trafic vers le serveur FTP. De plus, il ne doit bloquer aucun des ports requis. Pour fournir un exemple concret, on supposera encore une fois que le serveur FTP est le serveur standard sous OpenBSD : ftpd(8), utilisant la plage de ports par défaut.

Voici un exemple de règles qui pourraient être utilisées dans ce cas de figure :

```
ftp_server = "10.0.3.21"

rdr on $ext_if proto tcp from any to any port 21 -> $ftp_server \
   port 21
rdr on $ext_if proto tcp from any to any port 49152:65535 -> \
   $ftp_server port 49152:65535

# in on $ext_if
pass in quick on $ext_if proto tcp from any to $ftp_server \
   port 21 keep state
pass in quick on $ext_if proto tcp from any to $ftp_server \
   port > 49151 keep state

# out on $int_if
pass out quick on $int_if proto tcp from any to $ftp_server \
   port 21 keep state
pass out quick on $int_if proto tcp from any to $ftp_server \
   port > 49151 keep state
```

Plus d'Informations sur FTP

Vous pouvez obtenir plus d'informations concernant le filtrage et le fonctionnement de FTP de manière générale dans le livre blanc suivant :

- FTP Reviewed

[Précédent : Performances] [Index] [Suivant : Authpf : Shell Utilisateur pour les Passerelles d'Authentification]


www@openbsd.org
Originally [OpenBSD: ftp.html,v 1.12]
$Translation: ftp.html,v 1.4 2004/02/19 20:31:29 saad Exp $
$OpenBSD: ftp.html,v 1.4 2004/02/20 06:33:11 jufi Exp $

[Précédent : Problèmes avec FTP] [Index] [Suivant : Exemple #1 : Pare-feu SoHo]

PF: Authpf : Shell Utilisateur pour les Passerelles d'Authentification

---

Table des Matières

Introduction

Authpf(8) est un shell utilisateur pour les passerelles d'authentification. Une passerelle d'authentification est comme n'importe quelle passerelle réseau normale (un routeur) excepté que les utilisateurs doivent d'abord s'authentifier sur la passerelle avant que celle-ci ne permette au trafic généré par les utilisateurs de la traverser. Quand le shell d'un utilisateur est `/usr/sbin/authpf` (au lieu que son shell soit ksh(1), csh(1), etc) et que l'utilisateur se connecte en utilisant SSH, authpf fera les modifications nécessaires au jeu de règles pf(4) actif de telle manière à permettre au trafic généré par l'utilisateur de passer à travers les filtres et/ou d'être traduit en utilisant la Traduction d'Adresses IP ("NAT") ou d'être redirigé. Une fois que l'utilisateur se déconnecte ou que sa session est terminée, authpf supprimera toutes les règles chargées pour l'utilisateur et enlevera toutes les connexions avec maintien d'état que l'utilisateur aura ouvertes. Ainsi, le trafic de l'utilisateur ne passera à travers la passerelle que si l'utilisateur garde sa session SSH ouverte.

Authpf altère le jeu de règles pf(4) en ajoutant des règles à un jeu de règles nommé attaché à un point d'ancrage. Chaque fois qu'un utilisateur s'authentifie, authpf crée un nouveau jeu de règles nommé et y charge les règles de filtrage, `nat`, `binat`, et `rdr`. Les règles qu'authpf charge peuvent être configurées par utilisateur ou globalement.

Voici quelques exemples d'utilisation d'authpf :

- Exiger que les utilisateurs s'authentifient avant de permettre un accès à Internet.
- Permettre à certains utilisateurs -- tels que les administrateurs -- l'accès à certaines parties restreintes du réseau.
- Permettre uniquement aux utilisateurs connus l'accès au reste du réseau ou à Internet depuis un segment de réseau sans fil.
- Permettre à des travailleurs d'accèder à des ressources du système d'information de l'entreprise depuis chez eux, lorsqu'ils sont sur la route... Les utilisateurs en dehors des bureaux peuvent avoir accès au réseau d'entreprise et aussi être redirigés vers des ressources spécifiques (leur propre station de travail par exemple) selon le nom d'utilisateur qu'ils utilisent pour s'authentifier.
- Dans une configuration telle que celle d'une bibliothèque ou un autre lieu équipé de terminaux avec un accès Internet public, PF peut être configuré pour permettre un accès limité à Internet aux utilisateurs de passage. Authpf peut alors être utilisé pour permettre aux utilisateurs connus un accès complet.

Authpf enregistre le nom d'utilisateur et l'adresse IP de chaque utilisateur qui réussit à s'authentifier ainsi que le début et la fin de leur session. L'enregistrement se fait via syslogd(8). En utilisant cette information, un administrateur peut déterminer qui s'est connecté et responsabiliser les utilisateurs par rapport à leur trafic réseau.

Configuration

La section suivante fournit les étapes de base nécessaires pour configurer authpf. Pour une description complète de la configuration d'authpf, veuillez consulter la

page du manuel authpf.

Attacher Authpf au Jeu de Règles Principal

Authpf peut être rattaché au jeu de règles principal en utilisant des règles d'ancrage. Ces règles utilisent le mot-clé `anchor` :

```
nat-anchor authpf
rdr-anchor authpf
binat-anchor authpf
anchor authpf
```

PF "sortira" du jeu de règles principal pour évaluer les règles authpf à l'endroit où les règles `anchor` sont placées dans le jeu de règles. Il n'est pas nécessaire que les quatre règles `anchor` soient présentes. Par exemple, si authpf ne doit charger aucune règle de traduction `nat`, la règle `nat-anchor` peut être omise.

Configurer les Règles Chargées

Authpf charge les règles à partir d'un des deux fichiers suivants :

- `/etc/authpf/users/$USER/authpf.rules`
- `/etc/authpf/authpf.rules`

Le premier fichier contient des règles qui seront uniquement chargées lorsque l'utilisateur `$USER` (cette variable étant à remplacer par le nom d'utilisateur) se connecte. La configuration spécifique par utilisateur est utilisée lorsqu'un utilisateur donné -- un administrateur par exemple -- nécessite un jeu de règles différent du jeu de règles authpf par défaut. Le deuxième fichier contient les règles par défaut qui sont chargées pour tout utilisateur qui ne possède pas son propre fichier `authpf.rules`. Si le fichier spécifique à l'utilisateur existe, il est chargé au lieu du fichier par défaut. Au moins un des fichiers doit exister. Autrement authpf ne fonctionnera pas.

Les règles de filtrage et de traduction ont la même syntaxe que pour n'importe quel autre jeu de règles PF avec une seule exception : Authpf permet l'utilisateur de deux macros prédéfinies :

- `$user_ip` - l'adresse IP de l'utilisateur connecté
- `$user_id` - le nom d'utilisateur de l'utilisateur connecté

Il est recommandé d'utiliser la macro `$user_ip` pour ne permettre que le trafic provenant de la machine de l'utilisateur authentifié.

Les Listes de Contrôle d'Accès

On peut empêcher les utilisateurs d'utiliser authpf en créant un fichier sous le répertoire `/etc/authpf/banned/`. Le fichier doit avoir comme nom le nom de l'utilisateur banni. Le contenu du fichier sera affiché à l'utilisateur avant qu'authpf ne le déconnecte. Cette fonctionnalité fournit une méthode pratique pour notifier un utilisateur ou lui dire pourquoi son accès n'est pas autorisé et la personne qu'il doit contacter pour la restauration de son accès.

Autrement, il est aussi possible d'autoriser uniquement l'accès à des utilisateurs spécifiques en mettant leur nom d'utilisateur dans le fichier `/etc/authpf/authpf.allow`. Si le fichier `/etc/authpf/authpf.allow` n'existe pas ou contient "`*`", authpf permettra l'accès à n'importe quel utilisateur qui a réussi à se connecter via SSH et qui n'est pas explicitement banni.

Si authpf n'est pas capable de déterminer s'il doit autoriser ou interdire l'accès à un utilisateur, il affichera un message bref et déconnectera l'utilisateur. Une entrée dans le fichier `/etc/authpf/banned/` est toujours prise en compte avant une entrée dans le fichier `/etc/authpf/authpf.allow`.

Attribuer Authpf comme Shell Utilisateur

Afin qu'authpf fonctionne, il doit être attribué à l'utilisateur en tant que shell de connexion ("login shell"). Lorsque l'utilisateur s'authentifie selon les mécanismes offerts par sshd(8), authpf sera executé comme shell de l'utilisateur. Authpf vérifiera ensuite si l'utilisateur est autorisé à utiliser authpf, chargera les règles à partir du fichier approprié, etc.

Il existe plusieurs méthodes pour attribuer authpf à un utilisateur comme shell de connexion :

1. Manuellement pour chaque utilisateur en utilisant chsh(1), vipw(8), useradd(8), usermod(8), etc.
2. En affectant les utilisateurs à une classe de connexion et en changeant l'option `shell` dans `/etc/login.conf`.

Voir Les Personnes Connectées

Une fois l'utilisateur connecté et les règles chargées par authpf, ce dernier changera son nom de processus pour indiquer le nom d'utilisateur et l'adresse IP de l'utilisateur connecté :

```
# ps -ax | grep authpf
23664 p0  Is+     0:00.11 -authpf: charlie@192.168.1.3 (authpf)
```

Dans l'exemple ci-dessus, `charlie` est connecté depuis la machine 192.168.1.3. En envoyant un signal SIGTERM au processus authpf, on peut déconnecter l'utilisateur. Authpf supprimera aussi toute règle chargée pour l'utilisateur et supprimer toutes les connexions à état que l'utilisateur aura ouvert.

```
# kill -TERM 23664
```

Plus d'Information

Pour une description complète du fonctionnement d'authpf, veuillez vous référer à la page du manuel authpf.

Exemple

Dans cet exemple, authpf est utilisé sur une passerelle OpenBSD pour authentifier des utilisateurs sur un réseau sans fil faisant partie d'un réseau de campus plus grand. Une fois les utilisateurs authentifiés, et en supposant qu'ils ne font pas partie de la liste des utilisateurs bannis, ils seront autorisés à établir des sessions SSH vers l'extérieur et de naviguer sur le web (y compris sur les sites web sécurisés). De plus, ils pourront accéder à un des serveurs DNS du campus.

Le fichier `/etc/authpf/authpf.rules` contient les règles suivantes :

```
wifi_if = "wi0"
dns_servers = "{ 10.0.1.56, 10.0.2.56 }"

pass in quick on $wifi_if proto udp from $user_ip to $dns_servers \
   port domain keep state
pass in quick on $wifi_if proto tcp from $user_ip to port { ssh, http, \
   https } flags S/SA keep state
```

L'administrateur `charlie` devra être capable d'accéder aux serveurs SMTP et POP3 du campus en plus des droits d'accès précités. Les règles suivantes font partie du fichier `/etc/authpf/users/charlie/authpf.rules` :

```
wifi_if = "wi0"
smtp_server = "10.0.1.50"
pop3_server = "10.0.1.51"
dns_servers = "{ 10.0.1.56, 10.0.2.56 }"

pass in quick on $wifi_if proto udp from $user_ip to $dns_servers \
   port domain keep state
pass in quick on $wifi_if proto tcp from $user_ip to $smtp_server \
   port smtp flags S/SA keep state
pass in quick on $wifi_if proto tcp from $user_ip to $pop3_server \
   port pop3 flags S/SA keep state
pass in quick on $wifi_if proto tcp from $user_ip to port { ssh, http, \
   https } flags S/SA keep state
```

Voici le contenu du jeu de règles principal figurant dans le fichier `/etc/pf.conf` :

```
# macros
wifi_if = "wi0"
ext_if  = "fxp0"

scrub in all

# filtrage
block drop all

pass out quick on $ext_if proto tcp from $wifi_if:network flags S/SA \
   modulate state
pass out quick on $ext_if proto { udp, icmp } from $wifi_if:network \
   keep state

pass in quick on $wifi_if proto tcp from $wifi_if:network to $wifi_if \
   port ssh flags S/SA keep state

anchor authpf in on $wifi_if
```

Le jeu de règles est très simple :

- Bloquer tout par défaut.
- Autoriser les trafics TCP, UDP et ICMP sortants sur l'interface externe en provenance du réseau sans fil.
- Autoriser le trafic SSH entrant en provenance du réseau sans fil et destiné à la passerelle. Cette règle est nécessaire pour permettre aux utilisateurs de s'authentifier.
- Créer le point d'ancrage "authpf" pour le trafic entrant sur l'interface sans fil.

Le jeu de règles par défaut est utilisé pour bloquer tout trafic non strictement nécessaire pour le fonctionnement de l'architecture. Le trafic sortant de l'interface externe est autorisé. Cependant le trafic en entrée de l'interface sans fil est interdit. Une fois l'utilisateur authentifié, leur trafic est autorisé à traverser l'interface sans fil et à accéder au reste du réseau. Le mot-clé `quick` est utilisé un peu partout afin que PF n'évalue pas tous les jeux de règles nommés quand une nouvelle connexion traverse la passerelle.

[Précédent : Problèmes avec FTP] [Index] [Suivant : Exemple #1 : Pare-feu SoHo]


www@openbsd.org

Originally [OpenBSD: authpf.html,v 1.5]
$Translation: authpf.html,v 1.3 2004/02/19 20:30:12 saad Exp $
$OpenBSD: authpf.html,v 1.3 2004/02/20 06:33:11 jufi Exp $

[Précédent : Authpf: Shell Utilisateur pour les Passerelles d'Authentification] [Index]

PF : Exemple #1 : Pare-feu SoHo

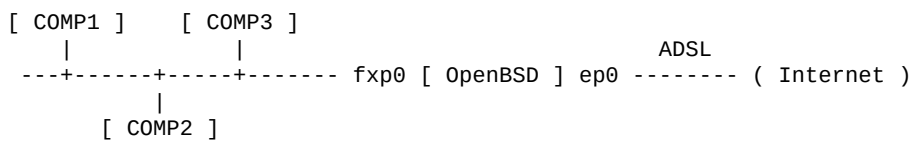
Table des Matières

Le Scenario

Dans cet exemple, PF fonctionne sur une machine OpenBSD jouant le rôle de pare-feu et de passerelle NAT pour un réseau SoHo. L'objectif global est de fournir un accès Internet au réseau local et de fournir un accès limité au pare-feu depuis Internet. Ce document a pour but de vous montrer comment on construit un jeu de règles correspondant à l'objectif précité.

Le Réseau

Le réseau est conçu de la manière suivante :



Il y a un certain nombre de machines sur le réseau interne. Le diagramme en montre trois mais le vrai nombre n'est pas une donnée utile. Ces machines sont des stations de travail normales servant à surfer sur le web, écrire des messages électroniques, participer à des forums de discussion en ligne, etc. Le réseau interne utilise le bloc de réseau 192.168.0.0 / 255.255.255.0.

Le routeur OpenBSD est une machine dotée d'un Pentium 100 et de deux cartes réseau : une 3Com 3c509B (`ep0`) et d'une Intel EtherExpress Pro/100 (`fxp0`). Le routeur a une connexion ADSL vers Internet et utilise la NAT pour partager cette connexion avec le réseau interne. L'adresse IP de l'interface externe est attribuée dynamiquement par le Fournisseur d'Accès Internet.

Les Objectifs

Les objectifs sont les suivants :

- Fournir un accès Internet sans restriction pour chaque machine du réseau interne.
- Utiliser un jeu de règles bloquant tout flux par défaut.

- Autoriser les flux entrants suivants sur le pare-feu à partir d'Internet :
  - SSH (port TCP 22) : utilisé pour la maintenance externe du pare-feu.
  - Auth/Ident (port TCP 113): utilisé par quelques services tels que SMTP et IRC.
  - Messages ICMP "Echo Request" : Le type de paquet ICMP utilisé par ping(8).
- Enregistrer des statistiques de filtrage pour l'interface externe.
- Par défaut, renvoyer un RST TCP ou un message ICMP "Unreachable" pour les paquets bloqués.
- S'assurer que le jeu de règles est aussi simple et facile à maintenir que possible.

Préparation

Ce document suppose que le hôte OpenBSD a été correctement configuré pour fonctionner comme routeur : configuration réseau, connexion Internet, `net.inet.ip.forwarding` avec une valeur de "`1`" etc.

Le Jeu de Règles

Les sections ci-après détaillent la manière dont le jeu de règles répondra aux objectifs précités.

Macros

Les macros suivantes sont définies pour rendre la maintenance et la lecture du jeu de règles plus faciles :

```
int_if = "fxp0"
ext_if = "ep0"

tcp_services = "{ 22, 113 }"
icmp_types = "echoreq"

priv_nets = "{ 127.0.0.0/8, 192.168.0.0/16, 172.16.0.0/12, 10.0.0.0/8 }"
```

Les deux premières lignes définissent les interfaces réseau sur lesquelles le filtrage sera effectué. Les deux lignes suivantes listent les numéros de port TCP des services ouverts depuis Internet (SSH et ident/auth) et les types de paquets ICMP qui sont autorisés à parvenir jusqu'aù pare-feu. La dernière ligne définit le réseau de loopback et les blocs d'adresses RFC 1918.

Remarque : Si la connexion Internet ADSL nécessite l'utilisation de PPPoE, le filtrage et la NAT s'effectueront sur l'interface `tun0` au lieu de `ep0`.

Options

Les deux options suivantes spécifient la réponse par défaut fournie par les règles de filtrage `block` et activent la collecte de statistiques sur l'interface externe :

```
set block-policy return
set loginterface $ext_if
```

Scrub

Il n'y a aucune raison pour ne pas utiliser la normalisation de paquets recommandée pour tous les paquets entrants. Il suffit d'utiliser la ligne suivante :

```
scrub in all
```

Traduction d'Adresses Réseau

Pour effectuer la NAT du réseau interne, la règle de `nat` suivante est utilisée :

```
nat on $ext_if from $int_if:network to any -> ($ext_if)
```

Vu que l'adresse IP de l'interface externe est attribuée dynamiquement, des parenthèses sont utilisés autour de l'interface de traduction afin que PF tienne compte automatiquement des changements d'adresse IP sur cette interface.

Redirection

La seule redirection nécessaire est pour ftp-proxy(8) afin de permettre aux clients FTP sur le réseau interne de se connecter à des serveurs FTP sur Internet.

```
rdr on $int_if proto tcp from any to any port 21 -> 127.0.0.1 port 8021
```

Il est à noter que cette règle ne fonctionnera que pour les connexions FTP au port 21. Si les utilisateurs se connectent de manière régulière à des serveurs FTP sur d'autres ports, une liste devra être utilisée pour spécifier le port de destination, par exemple : `from any to any port { 21, 2121 }`.

Règles de Filtrage

Détaillons maintenant les règles de filtrage. Commencez par l'interdiction par défaut de tout trafic :

```
block all
```

Avec cette règle, aucun trafic ne sera autorisé y compris le trafic provenant du réseau interne. Les règles ci-après vont ouvrir un certain nombre de flux sur le pare-feu afin de répondre aux objectifs précités et d'ouvrir toutes les interfaces virtuelles nécessaires.

Tout système Unix a une interface de "loopback". C'est une interface virtuelle représentant un réseau utilisé par les applications pour établir des canaux de communication locaux à la machine. De manière générale, tout le trafic au niveau de l'interface de "loopback" doit être autorisé. Sous OpenBSD, l'interface de "loopback" est lo(4).

```
pass quick on lo0 all
```

Ensuite, les adresses RFC 1918 doivent être bloquées en entrée et en sortie de l'interface externe. Ces adresses ne doivent jamais apparaître sur le réseau Internet public. Leur filtrage permet de s'assurer que le routeur ne divulgue pas les adresses utilisées par le réseau interne et de bloquer tous les paquets entrants avec une adresse source appartenant à l'un de ces réseaux.

```
block drop in  quick on $ext_if from $priv_nets to any
block drop out quick on $ext_if from any to $priv_nets
```

Il est à noter que `block drop` est utilisé pour dire à PF de ne pas répondre par un paquet TCP RST ou ICMP "Unreachable". Vu que les adresses correspondant à la RFC 1918 n'existent pas sur Internet, tout paquet envoyé vers une de ces adresses ne sera jamais acheminé vers sa destination de toute façon. L'option `quick` est utilisée pour dire à PF de ne pas évaluer le reste des règles de filtrage si un paquet correspond à l'une des règles ci-dessus; les paquets de et vers les réseaux `$priv_nets` seront immédiatement détruits.

Maintenant, il faut ouvrir les ports utilisés par les services réseau disponibles depuis Internet :

```
pass in on $ext_if inet proto tcp from any to ($ext_if) \
   port $tcp_services flags S/SA keep state
```

La spécification des ports réseau par le biais de la macro `$tcp_services` rend l'ouverture de nouveaux services pour des connexions provenant d'Internet plus facile dans la mesure où il suffira de modifier la macro et recharger le jeu de règles. Des services UDP peuvent aussi être mis à disposition en créant la macro `$udp_services` et en ajoutant une règle de filtrage adéquate similaire à la règle de filtrage ci-dessus en spécifiant `proto udp`.

Le trafic ICMP doit aussi être permis :

```
pass in inet proto icmp all icmp-type $icmp_types keep state
```

Comme pour la macro `$tcp_services`, la macro `$icmp_types` peut facilement être modifiée pour changer les types des paquets ICMP qui doivent être autorisés à atteindre le pare-feu. Notez que cette règle s'applique à toutes les interfaces réseau.

Maintenant, le trafic en provenance du réseau interne doit être autorisé. Nous supposerons que les utilisateurs du réseau interne savent ce qu'ils font et ne provoqueront pas de problème sur Internet. Ce n'est pas nécessairement une bonne supposition; pour certains environnements, il serait plus judicieux d'utiliser un jeu de règles plus restrictif.

```
pass in on $int_if from $int_if:network to any keep state
```

La règle ci-dessus permettra à n'importe quelle machine interne d'envoyer des paquets à travers le pare-feu; cependant, le pare-feu ne sera *pas* autorisé à à initier une connexion vers une machine interne. Est-ce une bonne idée ? Ceci dépendra de certains détails fins de la configuration réseau. Si le pare-feu est aussi un serveur DHCP, il aurait éventuellement besoin de vérifier la présence d'une adresse ("ping") pour voir si elle est disponible avant de l'attribuer à une machine. Permettre au pare-feu de se connecter au réseau interne veut dire aussi que quelqu'un qui accéderait en SSH au pare-feu depuis Internet sera autorisé à accèder

aux machines sur le réseau. Gardez à l'esprit qu'interdire au pare-feu de communiquer directement avec le réseau n'est pas d'un grand bénéfice du point de vue de la sécurité; si quelqu'un accède au pare-feu, il pourra très probablement altérer les règles de filtrage de toute façon. En ajoutant la règle suivante, le pare-feu sera capable d'initier des connexions vers le réseau interne :

```
pass out on $int_if from any to $int_if:network keep state
```

Notez que si les deux lignes ci-dessus sont utilisées, l'option `keep state` n'est pas nécessaire car il y a une règle pour laisser passer les paquets dans les deux directions. Cependant, si la ligne `pass out` n'est *pas* utilisée, la règle `pass in` *doit* comporter l'option `keep state`. Garder l'état d'une connexion permet aussi d'améliorer les performances : Les tables d'état sont vérifiées avant l'évaluation des règles, et si un état est trouvé, le passage du paquet à travers le pare-feu est autorisé sans que le jeu de règles ne soit évalué. Cette méthode de fonctionnement peut offrir de meilleures performances pour un pare-feu très chargé bien que pour un système aussi simple, la charge ne sera très certainement pas assez significative pour que cela fasse une différence.

Finalement, il faut laisser le trafic sortir de l'interface externe :

```
pass out on $ext_if proto tcp all modulate state flags S/SA
pass out on $ext_if proto { udp, icmp } all keep state
```

Le trafic TCP, UDP, et ICMP à destination d'Internet est autorisé sortir du pare-feu. L'information sur l'état des connexions est sauvegardée pour permettre aux paquets de retour de passer à leur tour la barrière que constitue le pare-feu.

Le Jeu de Règles Complet

```
# macros
int_if = "fxp0"
ext_if = "ep0"

tcp_services = "{ 22, 113 }"
icmp_types = "echoreq"

priv_nets = "{ 127.0.0.0/8, 192.168.0.0/16, 172.16.0.0/12, 10.0.0.0/8 }"

# options
set block-policy return
set loginterface $ext_if

# scrub
scrub in all

# nat/rdr
nat on $ext_if from $int_if:network to any -> ($ext_if)
rdr on $int_if proto tcp from any to any port 21 -> 127.0.0.1 \
   port 8021

# règles de filtrage
block all

pass quick on lo0 all

block drop in  quick on $ext_if from $priv_nets to any
block drop out quick on $ext_if from any to $priv_nets

pass in on $ext_if inet proto tcp from any to ($ext_if) \
   port $tcp_services flags S/SA keep state

pass in inet proto icmp all icmp-type $icmp_types keep state

pass in  on $int_if from $int_if:network to any keep state
pass out on $int_if from any to $int_if:network keep state

pass out on $ext_if proto tcp all modulate state flags S/SA
pass out on $ext_if proto { udp, icmp } all keep state
```

[Précédent : Authpf: Shell Utilisateur pour les Passerelles d'Authentification] [Index]


www@openbsd.org

Originally [OpenBSD: example1.html,v 1.12]
$Translation: example1.html,v 1.2 2004/01/02 21:49:07 saad Exp $
$OpenBSD: example1.html,v 1.2 2004/01/03 19:45:23 jufi Exp $