TP3 - Architecture Internet

Authors : Gabin Chognot & Julien Da Costa - P2025

Table of contents

Mise en oeuvre de la solution pour *client1*

Ajout des IPv4

On up l'interface `eth1` de *R2_home* et l'interface `eth0` de *client1* -> Aucune des deux n'a d'IPv4. On les rajoute donc :

- Sur *R2_home* : `ip addr add 192.168.100.254/24 dev eht1`
- Sur *client1* : `ip addr add 192.168.100.1/24 dev eht0`

On peut maintenant ping *client1* depuis *R2_home* et inversement.

Routes par défaut

On ajoute les routes par défaut :

- Sur *client1*, vers *R2_home* : `route add default gw 192.168.100.254`
- Sur *R2_home*, vers *R3_FAI* : `route add default gw 195.25.25.254`

Masquerading

On veut que *client1* puisse accéder à internet avec l'IP publique de *R2_home*. On va donc configurer du masquerading, en réutilisant les commandes du TP2.

On crée une table `nat` pour la translation d'adresse :

```
root@firewall:~# nft add table nat
```

Puis on ajoute une chaîne `postrouting` à cette table :

```
root@firewall:~# nft 'add chain nat postrouting { type nat hook postrouting
priority 100 ; }'
```

On a maintenant notre structure de base à laquelle on peut ajouter nos règles de translation d'adresse. On peut activer le masquerading *R2_home* :

```
root@firewall:~# nft add rule nat postrouting ip saddr 192.168.100.1 oif eth0
masquerade
```

On peut maintenant ping *R1* depuis *client1* avec `ping -c1 193.23.23.1`.

Filtrage

On reprend les commandes du TP2 pour créer la table de filtrage et la chaine de filtrage sur *R2_home* :

```
root@firewall: nft add table ip filter

root@firewall: nft 'add chain ip filter forward { type filter hook forward
priority 0; policy drop;}';

root@firewall: nft insert rule ip filter forward ct state established counter
accept
```

On ajoute les règles avec la commande type `nft add rule ip filter forward` :

- Autoriser le DNS : `nft add rule ip filter forward ip saddr 192.168.100.1 udp dport domain accept` et de même avec le port TCP.
- Autoriser HTTP : `nft add rule ip filter forward ip saddr 192.168.100.1 tcp dport { http, https } accept`

On ne peut maintenant plus contacter *R1* via `ping 193.23.23.2` (ICMP), comme attendu, mais on peut `curl 193.23.23.2` (HTTP), même si *R1* refuse la connexion sur son port 80.

Configuration du serveur DNS

> On veut :
>
> - Créer une zone *titi.fr* sur *srvDNS* avec un enregistrement A vers *R1*
> - Configurer les clients pour qu'ils utilisent *srvDNS* comme resolver

Création de la zone *titi.fr*

On reprend les éléments du TP1.

On modifie `/etc/bind/named.conf.local`, qui contient la configuration locale du serveur DNS, pour y déclarer les zones associées au domaine :

```
zone ".fr" {
        type master;
        file "/etc/bind/db.titi.fr";
};
```

Puis on crée le fichier `/etc/bind/db.titi.fr` et on y ajoute les RR nécessaires (SOA, NS, A) :

```
$TTL    3600
@       IN      SOA     .titi.fr. root.fr. (
                   2023050301;
                         3600;
                          600;
                        86400;
                          600
);

@        IN      NS      r1.titi.fr.
r1       IN      A       193.23.23.2
@        IN      A       193.23.23.2
```

On réalise un *test load* sur *titi.fr* avec `named-checkconf -z`, et un test de validité avec `named-checkzone fr /etc/bind/db.titi.fr`.

Tests :

- On peut utiliser *srvDNS* comme resolver depuis lui-même avec `dig r1.titi.fr @localhost`, la configuration est donc bonne.
- On ne peut pas utiliser *srvDNS* depuis les clients car on ne peut pas accéder au réseau privé de *R1* depuis l'extérieur.
    - Une première solution serait d'attribuer des IPs publiques à *srvDNS* et *srvHTTP*.
    - Une autre serait de réaliser du NAT entrant : c'est ce qu'on va faire.

Mise en œuvre de la solution pour l'accès aux serveurs

> On veut rendre *srvDNS* et *srvHTTP* accessibles depuis les clients en utilisant l'IP de *R1* (193.23.23.2).

On démarre le serveur http avec `systemctl enable --now lighttpd`.

Redirection de port

> Voir wiki.nftables.org

On veut rediriger :

- Sur *srvDNS* : 53 (en UDP et TCP)
- Sur *srvHTTP* : 80 et 443 (en TCP)

On crée la table et la chaine de redirection :

```
nft add table nat

nft 'add chain nat prerouting { type nat hook prerouting priority -100; }'
```

Puis on redirige le port 53 vers *srvDNS* :

```
nft 'add rule nat prerouting iif eth0 udp dport 53 dnat to 192.168.1.120'

nft 'add rule nat prerouting iif eth0 tcp dport 53 dnat to 192.168.1.120'
```

Explications :

- `iif eth0` correspond à l'interface d'entrée des paquets
- `dnat to 192.168.100.1` correspond à la destination de la redirection

Et on fait de même pour le port 80 et 443 vers *srvHTTP* :

```
nft 'add rule nat prerouting iif eth0 tcp dport { http, https } dnat to
192.168.100.2'
```

Routes

On ajoute :

- Une route sur *R1* vers son réseau local : `route add -net 192.168.100.0/24 dev eth1`
- Une route sur *srvDNS* et sur *srvHTP* vers *R1* avec la même commande : `route add default gw 192.168.100.0/24`

Configuration du resolver

On modifie `/etc/resolv.conf` sur *client1* pour y ajouter la ligne `nameserver 193.23.23.2`.

Tests avec `curl` et `dig`

Depuis *client1*, on peut maintenant :

- Accéder au serveur DNS sans préciser le resolver avec `dig titi.fr`
- Accéder au serveur HTTP via son domaine avec `curl titi.fr`

Tests avec `tcpdump` et `conntrack`

On flush la table de `conntrack` avec `conntrack -F`

On lance un `tcpdump` sur *R1* avec `tcpdump -i eth1` pour observer les paquets entrants. On peut observer :

- La requête DNS de *client1* vers *srvDNS* :

```
13:16:32.963769 IP 195.25.25.2.57741 > 192.168.100.1.domain: 9791+ A?
titi.fr. (25)

13:16:32.963779 IP 195.25.25.2.57741 > 192.168.100.1.domain: 23620+ AAAA?
titi.fr. (25)
```

- La réponse de *srvDNS* vers *client1* :

```
13:16:32.964322 IP 192.168.100.1.domain > 195.25.25.2.57741: 9791*- 1/1/1 A
193.23.23.2 (74)

13:16:32.964332 IP 192.168.100.1.domain > 195.25.25.2.57741: 23620*- 0/1/0
(74)
```

- La requête HTTP de *client1* vers *srvHTTP* :

```
13:16:32.965797 IP 195.25.25.2.49986 > 192.168.100.2.www: Flags [S], seq
807905498, win 64240, options [mss 1460,sackOK,TS val 777826176 ecr
0,nop,wscale 5], length 0
```

- La réponse de *srvHTTP* vers *client1* :

```
13:16:32.965987 IP 192.168.100.2.www > 195.25.25.2.49986: Flags [S.], seq
4235496285, ack 807905499, win 65160, options [mss 1460,sackOK,TS val
828930862 ecr 777826176,nop,wscale 5], length 0
```

- Un échange entre *srvHTTP* et *client* pour charger le reste de la page (trop long pour être affiché).

Avec `conntrack -L`, on peut observer les connexions établies :

```
root@R1:~# conntrack -L
udp      17 26 src=195.25.25.2 dst=193.23.23.2 sport=56508 dport=53
src=192.168.100.1 dst=195.25.25.2 sport=53 dport=56508 [ASSURED] use=1

tcp      6 117 TIME_WAIT src=195.25.25.2 dst=193.23.23.2 sport=50036 dport=80
src=192.168.100.2 dst=195.25.25.2 sport=80 dport=50036 [ASSURED] use=1

conntrack v1.4.5 (conntrack-tools): 2 flow entries have been shown.
```

- Une connexion UDP entre *R2_home* et *R1* (entre *client1* et *srvDNS*) sur le port 53 (DNS).
- Une connexion TCP entre *R2_home* et *R1* (entre *client1* et *srvHTTP*) sur le port 80 (HTTP).

`conntrack` permet de tracker les connexions établies et de les afficher. Contrairement à `tcpdump`, il ne traque pas les requêtes, et fonctionnement statiquement (affiche l'historique), pas dynamiquement (n'affiche pas les paquets en temps réel).

Mise en oeuvre de la solution pour *client2*

> On veut que *client2* accède à internet en utilisant le serveur *proxy*.

Il n'y a pas besoin de translation d'adresse pour *client2*, puisque *proxy* enverra ses requêtes.

Routes

On configure les routes par défaut de :

- *client2* vers *proxy* avec `route add default gw 192.168.0.2`
- *proxy* vers *R4_office* avec `route add default gw 192.168.0.1`
- *R4_office* vers *R1* avec `route add default gw 195.25.25.1`

On en profite pour configurer le resolver de *proxy* en ajoutant `nameserver 193.23.23.2` dans `/etc/resolv.conf`.

Masquerading pour *proxy*

On configure le masquerading sur *R4_office* pour que *proxy* puisse accéder à internet :

```
root@firewall:~# nft add table nat

root@firewall:~# nft 'add chain nat postrouting { type nat hook postrouting
priority 100 ; }'

root@firewall:~# nft add rule nat postrouting ip saddr 192.168.0.2 oif eth0
masquerade
```

On peut bien `ping 193.23.23.2` depuis *proxy*.

Firewalling sur *R4_office*

On configure le firewalling sur *R2_home* pour autoriser seulement les flux DNS, HTTP, et HTTPS de *proxy* :

```
root@firewall: nft add table ip filter

root@firewall: nft 'add chain ip filter forward { type filter hook forward
priority 0; policy drop;}';

root@firewall: nft insert rule ip filter forward ct state established counter
accept

root@firewall: nft add rule ip filter forward ip saddr 192.168.0.2 udp dport
domain accept

root@firewall: nft add rule ip filter forward ip saddr 192.168.0.2 udp dport
domain accept
```

```
root@firewall: nft add rule ip filter forward ip saddr 192.168.0.2 tcp dport {
http, https } accept
```

On ne peut plus `ping 193.23.23.2` depuis *proxy*, mais on peut toujours `curl titi.fr`. Le firewalling fonctionne donc.

Configuration de *proxy*

On modifie le fichier de configuration `/etc/tinyproxy/tinyproxy.conf` autoriser les requêtes depuis *client2* :

```
...
Allow 192.168.0.10
...
```

Puis on redémarre *tinyproxy* avec `systemctl restart tinyproxy`.

Test de configuration

On effectue un `wget -e use_proxy=yes -e http_proxy=192.168.0.2:8888 titi.fr` depuis *client2*, on reçoit bien la page web en question :

```
--2023-05-03 14:19:38--  http://titi.fr/
Connecting to 192.168.0.2:8888... connected.
Proxy request sent, awaiting response... 200 OK
Length: 3388 (3.3K) [text/html]
Saving to: 'index.html'

index.html          100%[===================>]   3.31K  --.-KB/s    in 0s

2023-05-03 14:19:38 (81.4 MB/s) - 'index.html' saved [3388/3388]
```

On peut vérifier que la requête passe bien par *proxy* en regardant les logs dans `/var/log/tinyproxy/tinyproxy.log` :

```
CONNECT   May 03 14:19:38 [1885]: Connect (file descriptor 7): 192.168.0.10
[192.168.0.10]
CONNECT   May 03 14:19:38 [1885]: Request (file descriptor 7): GET http://titi.fr/
HTTP/1.1
INFO      May 03 14:19:38 [1885]: No upstream proxy for titi.fr
INFO      May 03 14:19:38 [1885]: opensock: opening connection to titi.fr:80
INFO      May 03 14:19:38 [1885]: opensock: getaddrinfo returned for titi.fr:80
CONNECT   May 03 14:19:38 [1885]: Established connection to host "titi.fr" using
file descriptor 8.
```

```
INFO      May 03 14:19:38 [1885]: Closed connection between local client (fd:7)
and remote client (fd:8)
```

On peut voir que la requête est bien passée par *proxy*. Notre configuration fonctionne.

Analyse

La solution actuelle (utilisation d'un proxy HTTP) présente ses avantages et ses inconvénients :

Avantages :

- Possibilité de filtrer les requêtes HTTP (pour éviter d'accéder à des URLs malveillantes).
- Respect de la sécurité et de la vie privée.
- Possibilité d'effectuer du *caching* (stockage des ressources web fréquemment utilisées) pour fluidifier l'accès à internet.

Inconvénients :

- Vitesse réduite à cause de l'ajout d'un intermédiaire.
- Potentielle baisse de fiabilité à cause de ce même intermédiaire
- Dépendance du client au proxy (ni route vers le routeur ni DNS de configurés), et de fait plus faible adaptabilité.