Debuter avec Scapy

March 2, 2022

1 Remerciements

Avant toute chose, je me dois de remercier Willy Guillemin, Maître de Conférences à l'IUT R&T de Vélizy, pour son document "Introduction à Scapy" dont je me suis largement inspiré pour rédiger cette SAE24 ! Merci Willy d'avoir mis ton travail à disposition de la communauté R&T en cette période d'urgence de transition DUT/BUT.

2 Introduction

Scapy est un puissant programme interactif de manipulation de paquets. Il est capable de forger et d'envoyer des paquets avec un grand nombre de protocoles réseau, de recevoir, de capturer et d'analyser des paquets (récupérer des informations dans le paquet), de faire correspondre des requêtes et des réponses, et bien plus encore. On vous propose ici une introduction à Scapy en vous présentant les fonctionnalité nécessaires à la réalisation de votre SAE.

Vous pouvez trouver plus d'informations dans la documentation en ligne à l'adresse https://scapy.readthedocs.io.

3 Configuration par défaut et protocoles supportés

Scapy peut être utiliser selon 2 modes : en mode interactif depuis un terminal en tapant scapy ou dans un script ou un notebook Jupyter en Python. On importe la librairie scapy avec : **from scapy.all import** *.

Les paramètres de configuration par défaut peuvent être visualisés et modifiés avec la commande conf.

Remarque : on rappelle que les chaines de caractères formatées (aussi appelées f-strings) permettent d'inclure la valeur d'expressions Python dans des chaines de caractères en les préfixant avec f" chaine {expression}".

[1]:
```
from scapy.all import *
print(f"La version de Scapy est {conf.version}.")
print(f"\nL'interface par défaut utilisée pour l'émission et la réception des␣
 ↪paquets est {conf.iface}.")
print(f"\nLa table de routage utilisée est : \n  {conf.route}.")
print('\nLa passerelle par défaut est :', conf.route.route("0.0.0.0")[2])
```

```
La version de Scapy est 2.4.5.

L'interface par défaut utilisée pour l'émission et la réception des paquets est
eno1.

La table de routage utilisée est :
  Network      Netmask        Gateway   Iface  Output IP  Metric
0.0.0.0      0.0.0.0        10.2.4.1  eno1   10.2.4.3   100
10.2.4.0     255.255.255.0  0.0.0.0   eno1   10.2.4.3   100
127.0.0.0    255.0.0.0      0.0.0.0   lo     127.0.0.1  1
169.254.0.0  255.255.0.0    0.0.0.0   eno1   10.2.4.3   1000  .

La passerelle par défaut est : 10.2.4.1
```

Pour voir les protocoles pris en charge et la structure des données de protocole, utilisez la commande ls(protocole). Certains champs ont une valeur par défaut (par exemple 64 pour le ttl dans un paquet IP) :

[ ]:
```
ls()
```

[3]:
```
ls(IP)
```

```
version    : BitField  (4 bits)                = ('4')
ihl        : BitField  (4 bits)                = ('None')
tos        : XByteField                        = ('0')
len        : ShortField                        = ('None')
id         : ShortField                        = ('1')
flags      : FlagsField                        = ('<Flag 0 ()>')
frag       : BitField  (13 bits)               = ('0')
ttl        : ByteField                         = ('64')
proto      : ByteEnumField                     = ('0')
chksum     : XShortField                       = ('None')
src        : SourceIPField                     = ('None')
dst        : DestIPField                       = ('None')
options    : PacketListField                   = ('[]')
```

4 Forger, visualiser et modifier un packet

4.1 Forger et visualiser des paquets

Pour créer/forger un paquet, indiquez la pile protocolaire du protocole le plus bas au plus haut en séparant les protocoles par un slash '/'. Scapy configurera automatiquement le champs que vous n'indiquez pas avec la configuration par défaut.

Le paquet est un objet et on peut peut visualiser le contenu du paquet avec la méthode show() ou la méthode show2() qui calcule en plus les champs comme la longueur, le checksum du paquet ou la conversion d'un nom en une adresse IP.

[3]:
```
paquet1=IP()/UDP()
paquet1.summary()
```

[3]:
```
'IP / UDP 127.0.0.1:domain > 127.0.0.1:domain'
```

[4]:
```
paquet1.show()
```

```
###[ IP ]###
  version   = 4
  ihl       = None
  tos       = 0x0
  len       = None
  id        = 1
  flags     =
  frag      = 0
  ttl       = 64
  proto     = udp
  chksum    = None
  src       = 127.0.0.1
  dst       = 127.0.0.1
  \options   \
###[ UDP ]###
     sport     = domain
     dport     = domain
     len       = None
     chksum    = None
```

[5]:
```
IPs='192.168.1.1'
IPd='192.168.1.254'
paquet2=IP(src=IPs, dst=IPd)/UDP()
paquet2.show()
paquet2.show2()
```

```
###[ IP ]###
  version   = 4
  ihl       = None
  tos       = 0x0
  len       = None
  id        = 1
  flags     =
  frag      = 0
  ttl       = 64
  proto     = udp
  chksum    = None
  src       = 192.168.1.1
  dst       = 192.168.1.254
  \options   \
```

```
###[ UDP ]###
     sport     = domain
     dport     = domain
     len       = None
     chksum    = None

###[ IP ]###
  version   = 4
  ihl       = 5
  tos       = 0x0
  len       = 28
  id        = 1
  flags     =
  frag      = 0
  ttl       = 64
  proto     = udp
  chksum    = 0xf680
  src       = 192.168.1.1
  dst       = 192.168.1.254
  \options   \
###[ UDP ]###
     sport     = domain
     dport     = domain
     len       = 8
     chksum    = 0x7b24
```

Si on spécifie un nom de domaine à la place d'une adresse IP, Scapy fait une requête DNS pour obtenir l'adresse IP correspondante.

[6]:
```
MACs='11:22:33:44:55:66'
MACd='00:0A:1F:3B:4E:64'
IPs='192.168.1.1'
IPd='www.univ-pau.fr'
frm=Ether(src=MACs, dst=MACd)/IP(src=IPs, dst=IPd)/TCP(flags='SA')/"C est␣
 ↪vraiment bien Scapy"
frm.show()
frm.show2()
```

```
###[ Ethernet ]###
  dst       = 00:0A:1F:3B:4E:64
  src       = 11:22:33:44:55:66
  type      = IPv4
###[ IP ]###
     version   = 4
     ihl       = None
     tos       = 0x0
     len       = None
```

```
     id        = 1
     flags     =
     frag      = 0
     ttl       = 64
     proto     = tcp
     chksum    = None
     src       = 192.168.1.1
     dst       = Net("www.univ-pau.fr/32")
     \options   \
###[ TCP ]###
        sport     = ftp_data
        dport     = http
        seq       = 0
        ack       = 0
        dataofs   = None
        reserved  = 0
        flags     = SA
        window    = 8192
        chksum    = None
        urgptr    = 0
        options   = ''
###[ Raw ]###
           load      = 'C est vraiment bien Scapy'

###[ Ethernet ]###
  dst       = 00:0a:1f:3b:4e:64
  src       = 11:22:33:44:55:66
  type      = IPv4
###[ IP ]###
     version   = 4
     ihl       = 5
     tos       = 0x0
     len       = 65
     id        = 1
     flags     =
     frag      = 0
     ttl       = 64
     proto     = tcp
     chksum    = 0x59f4
     src       = 192.168.1.1
     dst       = 194.167.156.113
     \options   \
###[ TCP ]###
        sport     = ftp_data
        dport     = http
        seq       = 0
        ack       = 0
        dataofs   = 5
```

```
        reserved  = 0
        flags     = SA
        window    = 8192
        chksum    = 0x786d
        urgptr    = 0
        options   = []
###[ Raw ]###
           load      = 'C est vraiment bien Scapy'
```

4.2 Charger un fichier pcap

La fonction rdpcap("nom de fichier") lit un fichier pcap ou pcapng (format Wireshark) et renvoie une liste de paquets (vous devez être dans le répertoire de fichiers). On peut ensuite inspecter les différents paquet de cette capture par exemple avant de travailler en temps réel sur le réseau.

[7]:
```
trames=rdpcap("Wireshark/Ping_Google.pcapng")
print("La capture comprend les paquets suivants :\n")
trames.summary()
```

```
La capture comprend les paquets suivants :

Ether / IP / ICMP 192.168.1.48 > 8.8.8.8 echo-request 0 / Raw
Ether / IP / ICMP 8.8.8.8 > 192.168.1.48 echo-reply 0 / Raw
Ether / IP / ICMP 192.168.1.48 > 8.8.8.8 echo-request 0 / Raw
Ether / IP / ICMP 8.8.8.8 > 192.168.1.48 echo-reply 0 / Raw
```

5 Inspecter et obtenir la valeur d'un champ d'un paquet

Scapy utilise sa propre structure de données pour représenter les paquets. Cette structure est basée sur des **dictionnaires** imbriqués. Un dictionnaire est une collection qui associe une clé à une valeur. Pour créer un dictionnaire, on associe une clé à une valeur en les séparant par :, le tout entre accolades {}. Pour accéder à la valeur d'élément d'un dictionnaire, il faut utiliser les crochets et préciser la valeur de la clé. Il est possible de changer la valeur pour une clé donnée ou ajouter une nouvelle valeur pour une nouvelle clé.

[8]:
```
RT={'Siami' : 'chef de departement', 'Bascou' :'enseignant', 'Rivenq' :␣
  ↪'secretaire' }
RT['Siami']
```

[8]:
```
'chef de departement'
```

[9]:
```
RT['Gallon']='enseignant'
[print(f"Mr {nom} est {RT[nom]} au département R&T de l'IUT de Mont de Marsan")␣
  ↪for nom in RT]
```

```
Mr Siami est chef de departement au département R&T de l'IUT de Mont de Marsan
Mr Bascou est enseignant au département R&T de l'IUT de Mont de Marsan
```

```
Mr Rivenq est secretaire au département R&T de l'IUT de Mont de Marsan
Mr Gallon est enseignant au département R&T de l'IUT de Mont de Marsan
```

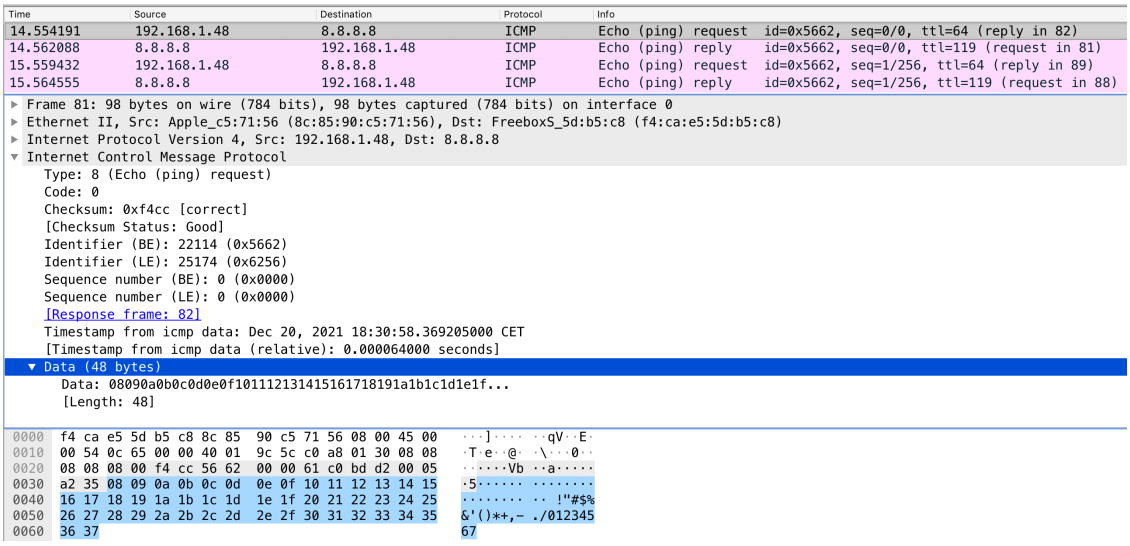
[9]:
```
[None, None, None, None]
```

Des dictionnaires imbriqués sont des dictionnaires dans des dictionnaires comme illustré ci-dessous :

[10]:
```
RT={'Siami' : {'fonction' : 'chef de departement', 'année de recutement' :␣
 ↪2002, \
                  'Enseignant en' : 'Electronique, Programmation'},
   'Bascou' : {'fonction' : 'DE 1ere annee de BUT', 'année de recutement' :␣
 ↪2000, \
                  'Enseignant en' : 'Réseaux'}}
RT['Bascou']['fonction']
```

[10]:
```
'DE 1ere annee de BUT'
```

Dans un fichier pcap ou pcapng, chaque paquet est un élément d'un dictionnaire, la clé correspondant au numéro de paquet en partant de 0. Dans la capture chargée précédemement on a 4 paquets :



On peut donc récupérer le premier paquet avec :

[11]:
```
print('La premiere trame est la suivante :')
trames[0]
```

```
La premiere trame est la suivante :
```

[11]:
```
<Ether  dst=f4:ca:e5:5d:b5:c8 src=8c:85:90:c5:71:56 type=IPv4 |<IP  version=4
ihl=5 tos=0x0 len=84 id=3173 flags= frag=0 ttl=64 proto=icmp chksum=0x9c5c
src=192.168.1.48 dst=8.8.8.8 |<ICMP  type=echo-request code=0 chksum=0xf4cc
```

```
id=0x5662 seq=0x0 unused='' |<Raw  load='a\\xc0\\xbd\\xd2\x00\x05\\xa25\x08\t\n\
x0b\x0c\r\x0e\x0f\x10\x11\x12\x13\x14\x15\x16\x17\x18\x19\x1a\x1b\x1c\x1d\x1e\x1
f !"#$%&\'()*+,-./01234567' |>>>>
```

Chaque paquet est une collection de dictionnaires imbriqués, chaque couche étant un dictionnaire enfant de la couche précédente, construit à partir de la couche la plus basse. On peut le voir avec l'imbrication des signes < et > dans le paquet précédent. On peut donc accéder à une couche et les couches supérieur puisqu'elles sont imbriquées avec :

[12]:
```
trames[2]['IP']
```

[12]:
```
<IP  version=4 ihl=5 tos=0x0 len=84 id=7714 flags= frag=0 ttl=64 proto=icmp
chksum=0x8a9f src=192.168.1.48 dst=8.8.8.8 |<ICMP  type=echo-request code=0
chksum=0xe086 id=0x5662 seq=0x1 unused='' |<Raw  load='a\\xc0\\xbd\\xd3\x00\x05\
\xb6y\x08\t\n\x0b\x0c\r\x0e\x0f\x10\x11\x12\x13\x14\x15\x16\x17\x18\x19\x1a\x1b\
x1c\x1d\x1e\x1f !"#$%&\'()*+,-./01234567' |>>>
```

Ou plus simplement sans mettre les guillemets , Scapy interprétenant le contenu des crochets comme une str :

[13]:
```
trames[2][IP]
```

[13]:
```
<IP  version=4 ihl=5 tos=0x0 len=84 id=7714 flags= frag=0 ttl=64 proto=icmp
chksum=0x8a9f src=192.168.1.48 dst=8.8.8.8 |<ICMP  type=echo-request code=0
chksum=0xe086 id=0x5662 seq=0x1 unused='' |<Raw  load='a\\xc0\\xbd\\xd3\x00\x05\
\xb6y\x08\t\n\x0b\x0c\r\x0e\x0f\x10\x11\x12\x13\x14\x15\x16\x17\x18\x19\x1a\x1b\
x1c\x1d\x1e\x1f !"#$%&\'()*+,-./01234567' |>>>
```

On peut ensuite accéder à un champ d'une couche avec un point et le nom du champ :

[14]:
```
trames[2][IP].dst
```

[14]:
```
'8.8.8.8'
```

Certains protocoles (généralement applicatifs) ne sont pas décodés par Scapy. Les données sont alors des données brutes indiqué par le mot clé **"Raw"** et ce sont des bytes notés b". C'est par exemple le cas du contenu du paquet ICMP :

[15]:
```
trames[2][Raw].load
```

[15]:
```
b'a\xc0\xbd\xd3\x00\x05\xb6y\x08\t\n\x0b\x0c\r\x0e\x0f\x10\x11\x12\x13\x14\x15\x
16\x17\x18\x19\x1a\x1b\x1c\x1d\x1e\x1f !"#$%&\'()*+,-./01234567'
```

Si le protocole est en mode texte comme ici, on peut alors récupérer (parser) des champs du message avec les méthodes sur les chaines de caractères notamment la methode split qui permet de séparer les chaines en fonction d'un séparateur (ici l'espace, la tabulation ()ˆ ou le retour à la ligne(\n): sep = None) dans une liste :

[16]:
```
trames[2][Raw].load.split(sep=None)
```

[16]:
```
[b'a\xc0\xbd\xd3\x00\x05\xb6y\x08',
 b'\x0e\x0f\x10\x11\x12\x13\x14\x15\x16\x17\x18\x19\x1a\x1b\x1c\x1d\x1e\x1f',
 b'!"#$%&\'()*+,-./01234567']
```

On peut alors récupérer le champ désiré en récupérant le bon élément de la liste :

[17]:
```
trames[2][Raw].load.split(sep=None)[2]
```

[17]:
```
b'!"#$%&\'()*+,-./01234567'
```

Il s'agit d'un contenu hexadécimal noté b" et on peut donc le décoder en indiquant le format d'encodage (par défaut on prendra UTF8) pour obtenir la chaîne de caratères correspondante :

[18]:
```
trames[2][Raw].load.split(sep=None)[2].decode('UTF8')
```

[18]:
```
'!"#$%&\'()*+,-./01234567'
```

6 Emettre des paquets ou des trames

Il est souvent utile de forger ses propres paquets ou ses propres trames, afin de spécifier les différents champs des différents en-têtes (ARP, IP (v4 ou v6), ICMP (v4 ou v6), UDP, TCP, ...) et de les envoyer. Pour cela il existe deux fonctions sous Scapy: send (envoi de paquet au niveau 3) ou sendp (envoi de trame au niveau 2).

Voici un exemple très simple pour envoyer un ping (à savoir que le type par défaut d'un paquet ICMP si il n'est pas spécifié, est une requête "Echo", soit le type 8)

[19]:
```
ping=IP(dst='10.2.12.4')/ICMP()/"On peut ajouter ici des données à émettre avec␣
 ↪le ping"
send(ping)
```

```
Sent 1 packets.
.
```

IMPORTANT: Notez bien que vous devrez lancer ce code en tant que "root" pour avoir les permissions d'émettre au niveau 2 ou 3. Seules les émissions au niveau 4 sont autorisées pour les utilisateurs non privilégiés.

REMARQUE: Il est généralement intéressant de lancer une capture Wireshark en parallèle, dans une autre fenêtre, afin de vusualiser les informations émises par ce biais.

La commande send accepte plusieurs arguments, comme par exemple:

[ ]:
```
send(ping, loop=1) // NE PAS EXECUTER !!!
```

qui enverra des pings en boucle infinie sur le pauvre récepteur ..., ou encore:

[ ]:
```
send(ping, loop=1, inter=1) // NE PAS EXECUTER NON PLUS !!!
```

qui enverra des pings en boucle infinie, mais respectera un intervalle de 1s entre chaque envoi. C'est beaucoup plus respectueux ! Toutes les options de cette fonction sont documentées ici: https://scapy.readthedocs.io/en/latest/api/scapy.sendrecv.html

Voici le même exemple, mais depuis la couche 2 (notez qu'il faut dans ce cas définir l'en-tête de trame …). Dans l'exemple ci-dessous, l'interface de sortie peut être spécifié explicitement … cela peut être parfois utile (surtout avec des adresses APIPA en IPv4 ou "Local Link" en IPv6).

[20]:
```
trameping=Ether()/IP(dst='10.2.12.4')/ICMP()/"On peut ajouter ici des données à␣
 ↪émettre avec le ping"
sendp(trameping, iface="eno1")
```

```
Sent 1 packets.
.
```

7 Recevoir les réponses à ses émissions

Nous venons de voir comment émettre les PDU (Protocol Data Unit), qu'il s'agisse de paquets ou de trames. Nous allons maintenant voir comment récupérer les réponses. Il existe pour cela plusieurs fonctions à chaque niveau: * Pour le niveau 3: * sr * sr1 * srloop * … * Pour le niveau 2: * srp * srp1 * srploop * … Comme précédemment, nous devons créer le PDU correspondant (paquet ou trame), seule la façon de l'envoyer va varier: nous allons utiliser les nouvelles fonctions.

[21]:
```
ping=IP(dst='10.2.12.4')/ICMP()/"On peut ajouter ici des données à émettre avec␣
 ↪le ping"
sr(ping)
```

```
Begin emission:
Finished sending 1 packets.
*
Received 1 packets, got 1 answers, remaining 0 packets
```

[21]:
```
(<Results: TCP:0 UDP:0 ICMP:1 Other:0>,
 <Unanswered: TCP:0 UDP:0 ICMP:0 Other:0>)
```

Comme on le voit ci-dessus, la réponse reçue consécutivement à l'envoi du ping est affichée à l'écran. "Results" correspond aux réponses reçues, "Unanswered" correspond aux requêtes restées sans réponses. Ceci peut être très pratique pour faire des statistiques sur les "ping" perdus …
Il est important de connaître le principe pour récupérer dans deux tableaux (par exemple ok et nonok) les PDU reçus et les PDU restés sans réponses afin de les traiter ultérieurement:

[22]:
```
ping=IP(dst='10.2.12.4')/ICMP()/"On peut ajouter ici des données à émettre avec␣
 ↪le ping"
ok, nonok = srloop(ping, count=10, inter=1)
```

```
RECV 1: IP / ICMP 10.2.12.4 > 10.2.4.3 echo-reply 0 / Raw
RECV 1: IP / ICMP 10.2.12.4 > 10.2.4.3 echo-reply 0 / Raw
RECV 1: IP / ICMP 10.2.12.4 > 10.2.4.3 echo-reply 0 / Raw
```

```
RECV 1: IP / ICMP 10.2.12.4 > 10.2.4.3 echo-reply 0 / Raw
RECV 1: IP / ICMP 10.2.12.4 > 10.2.4.3 echo-reply 0 / Raw
RECV 1: IP / ICMP 10.2.12.4 > 10.2.4.3 echo-reply 0 / Raw
RECV 1: IP / ICMP 10.2.12.4 > 10.2.4.3 echo-reply 0 / Raw
RECV 1: IP / ICMP 10.2.12.4 > 10.2.4.3 echo-reply 0 / Raw
RECV 1: IP / ICMP 10.2.12.4 > 10.2.4.3 echo-reply 0 / Raw
RECV 1: IP / ICMP 10.2.12.4 > 10.2.4.3 echo-reply 0 / Raw

Sent 10 packets, received 10 packets. 100.0% hits.
```

Il est ensuite facile d'utiliser ce tableau pour récupérer les informations qu'il contient:

[23]:
```
ok
```

[23]:
```
<Results: TCP:0 UDP:0 ICMP:10 Other:0>
```

[24]:
```
ok[0]
```

[24]:
```
QueryAnswer(query=<IP  frag=0 proto=icmp dst=10.2.12.4 |<ICMP  |<Raw  load='On
peut ajouter ici des données à émettre avec le ping' |>>>, answer=<IP  version=4
ihl=5 tos=0x0 len=85 id=15535 flags= frag=0 ttl=63 proto=icmp chksum=0x1aef
src=10.2.12.4 dst=10.2.4.3 |<ICMP  type=echo-reply code=0 chksum=0x2929 id=0x0
seq=0x0 unused='' |<Raw  load='On peut ajouter ici des données à émettre avec le
ping' |>>>)
```

Et vous remarquerez que dans cet affichage détaillé, sont présents les deux PDU: le ping envoyé ainsi que la réponse reçue, que vous pourrez bien entendu afficher séparément en utilisant la syntaxe: ok[0][0]ou ok[0][1]

[25]:
```
ok[0][0]
```

[25]:
```
<IP  frag=0 proto=icmp dst=10.2.12.4 |<ICMP  |<Raw  load='On peut ajouter ici
des données à émettre avec le ping' |>>>
```

[26]:
```
ok[0][1]
```

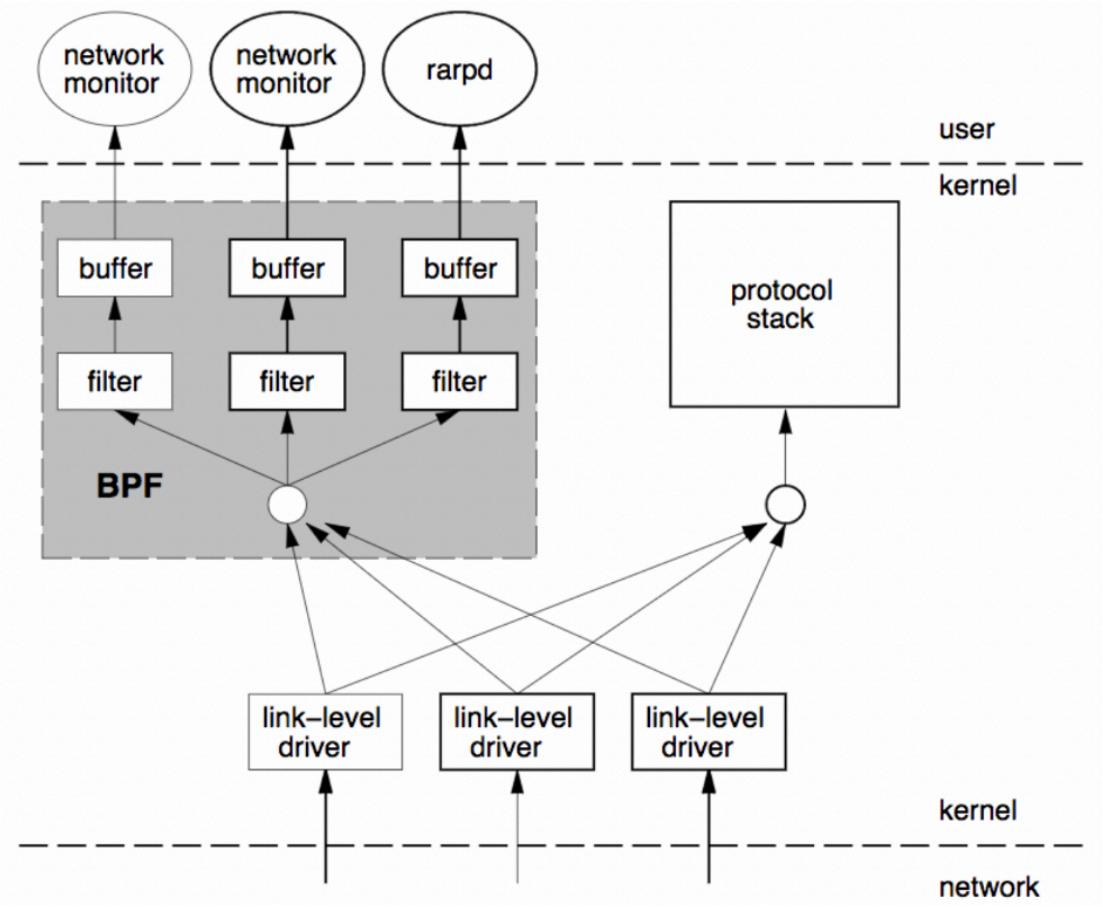
[26]:
```
<IP  version=4 ihl=5 tos=0x0 len=85 id=15535 flags= frag=0 ttl=63 proto=icmp
chksum=0x1aef src=10.2.12.4 dst=10.2.4.3 |<ICMP  type=echo-reply code=0
chksum=0x2929 id=0x0 seq=0x0 unused='' |<Raw  load='On peut ajouter ici des
données à émettre avec le ping' |>>>
```

8 Sniffer des paquets

La fonction **sniff(filter="", count=0, prn=None, lfilter=None, timeout=None, ..)** permet de capturer le trafic réseau à partir d'une ou plusieurs interfaces. sniff() a 6 options :

- filter : filtre les paquets à l'intérieur du noyau Linux ce qui rend le filtrage très rapide. L'écriture du filtre utilise la syntaxe BPF "Berkeley Packet Filter" dont on trouvera une

documentation ici : https://biot.com/capstats/bpf.html. On utilisera donc ce filtre dans la SAE pour ne garder que les paquets associés au protocole applicatif utilisé. Ex: filter="tcp and port 80"



- count : nombre de paquet à capturer. La valeur par défaut est 0 ce qui veut dire qu'il n'y a pas de limite au nombre de paquets capturés.
- prn : nom de la fonction à appliquer à chaque paquet reçu. C'est dans cette fonction qu'on effectuera les traitements sur les paquets (détection du login, mot de passe, …)
- lfilter : filtre les paquets à l'aide d'une fonction Python, on pourra utiliser une fonction lambda. Comme filter ce filtre est utilisé pour filtrer les paquets en utilisant la syntaxe Python/Scapy. On peut dont cibler n'importe quel champ d'un protocole par contre ce filtre est beaucoup plus lent car pas implémenté dans le noyau. Ex : lfilter = lambda pkt: TCP in pkt and(pkt[TCP].dport == 80 or pkt[TCP].sport == 80)
- timeout : durée de la capture, par défaut None=∞ (^C pour arrêter)
- iface : interface sur laquelle on souhaite capturer les paquets (défaut :all)
- store : s'il faut stocker les paquets capturés ou les supprimer (store=0). Si la capture dure dans le temps et qu'on stocke les paquets, la RAM allouée au processus augmentera progressivement avec l'enregistrement des paquets.
- stopfilter : fonction à évaluer pour arrêter la capture (la fonction doit retourner true pour arrêter, false pour continuer)

Tous les paramètres disponibles pour cette fonction sont documentés ici:

https://scapy.readthedocs.io/en/latest/api/scapy.sendrecv.html

Si on veut utiliser la fonction sniff() dans un notebook, on mettra une timeout ou un nombre de paquets à capturer. Un exemple d'utilisation de la fonction sniff pour afficher les 4 premiers paquets ICMP émis et reçus sur une interface est donné ci-dessous :

[ ]:
```
from scapy.all import *

ICMP_types={ 0 : 'Echo-Reply', 3 : 'Destination Unreachable', 8 : 'Echo'}

def print_icmp (packet) :
    type=packet[ICMP].type
    ips=packet[IP].src
    ipd=packet[IP].dst
    if ips==iface_ip :
        print(f"Emission d'un paquet ICMP {ICMP_types[type]} vers {ipd}")
    else :
        print(f"Réception d'un paquet ICMP {ICMP_types[type]} en provenance de␣
 ↪{ips}")

iface_ip=get_if_addr(conf.iface)
sniff(filter="icmp", prn=print_icmp, store=0, iface='en0', count=4)
```

Il existe bien entendu de multiples autres fonctionnalités offertes par ce formidable outil, vous aurez tout loisir de les découvrir à travers de nombreuses documentations disponibles sur Internet. L'important est est de PRATIQUER … et n'oubliez pas d'utiliser Wireshark en parallèle, il est très facile d'importer des fichiers .pcap ou .pcapng sous Scapy.