Introduction à Scapy SAE11

January 2, 2022

1 Introduction

Il existe de nombreux outils de sécurité que vous pouvez retrouvez par exemple dans la distribution Kali Linux. Cependant, ces outils de sécurité ont des paramètres modifiables limités. Vous vous retrouverez donc toujours dans des situations où vous souhaitez générer une séquence de paquets qui n'est pas possible avec ces logiciels et vous devrez construire vos propres outils.

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. Un cours spécifique sur Scapy vous sera donnée dans le module R307 Penstesting si vous choisissez le parcours Cybersécurité. Vous pouvez trouver plus d'informations dans la documentation en ligne à l'adresse https://scapy.readthedocs.io.

2 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}".

[12]:
```
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 2.4.3.
```

```
L'interface par défaut utilisée pour l'émission et la réception des paquets est
en0.

La table de routage utilisée est :
  Network           Netmask           Gateway        Iface  Output IP     Metric
0.0.0.0           0.0.0.0           192.168.1.254  en0    192.168.1.48  1
127.0.0.0         255.0.0.0         0.0.0.0        lo0    127.0.0.1     1
127.0.0.1         255.255.255.255   0.0.0.0        lo0    127.0.0.1     1
169.254.0.0       255.255.0.0       0.0.0.0        en0    192.168.1.48  1
192.168.1.0       255.255.255.0     0.0.0.0        en0    192.168.1.48  1
192.168.1.13      255.255.255.255   0.0.0.0        en0    192.168.1.48  1
192.168.1.254     255.255.255.255   0.0.0.0        en0    192.168.1.48  1
192.168.1.254     255.255.255.255   0.0.0.0        en0    192.168.1.48  1
192.168.1.255     255.255.255.255   0.0.0.0        en0    192.168.1.48  1
192.168.1.29      255.255.255.255   0.0.0.0        en0    192.168.1.48  1
192.168.1.48      255.255.255.255   0.0.0.0        en0    192.168.1.48  1
192.168.1.48      255.255.255.255   0.0.0.0        lo0    127.0.0.1     1
192.168.1.49      255.255.255.255   0.0.0.0        en0    192.168.1.48  1
192.168.1.5       255.255.255.255   0.0.0.0        en0    192.168.1.48  1
224.0.0.0         240.0.0.0         0.0.0.0        en0    192.168.1.48  1
224.0.0.251       255.255.255.255   0.0.0.0        en0    192.168.1.48  1
239.255.255.250   255.255.255.255   0.0.0.0        en0    192.168.1.48  1
255.255.255.255   255.255.255.255   0.0.0.0        en0    192.168.1.48  1.

La passerelle par défaut est : 192.168.1.254
```

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()
```

```
AH         : AH
AKMSuite   : AKM suite
ARP        : ARP
ASN1P_INTEGER : None
ASN1P_OID  : None
ASN1P_PRIVSEQ : None
ASN1_Packet : None
ATT_Error_Response : Error Response
ATT_Exchange_MTU_Request : Exchange MTU Request
ATT_Exchange_MTU_Response : Exchange MTU Response
ATT_Execute_Write_Request : Execute Write Request
ATT_Execute_Write_Response : Execute Write Response
ATT_Find_By_Type_Value_Request : Find By Type Value Request
ATT_Find_By_Type_Value_Response : Find By Type Value Response
ATT_Find_Information_Request : Find Information Request
ATT_Find_Information_Response : Find Information Response
ATT_Handle : ATT Short Handle
ATT_Handle_UUID128 : ATT Handle (UUID 128)
ATT_Handle_Value_Indication : Handle Value Indication
ATT_Handle_Value_Notification : Handle Value Notification
```

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

```
version    : BitField (4 bits)                = (4)
ihl        : BitField (4 bits)                = (None)
tos        : XByteField                       = (0)
```

```
len        : ShortField                          = (None)
id         : ShortField                          = (1)
flags      : FlagsField (3 bits)                 = (<Flag 0 ()>)
frag       : BitField (13 bits)                  = (0)
ttl        : ByteField                           = (64)
proto      : ByteEnumField                       = (0)
chksum     : XShortField                         = (None)
src        : SourceIPField                       = (None)
dst        : DestIPField                         = (None)
options    : PacketListField                     = ([])
```

3 Forger, visualiser et modifier un packet

3.1 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.

[5]:
```
paquet1=IP()/UDP()
paquet1.summary()
```

[5]:
```
'IP / UDP 127.0.0.1:domain > 127.0.0.1:domain'
```

[6]:
```
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
```

[7]:
```
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.

[8]:
```
MACs='11:22:33:44:55:66'
MACd='00:0A:1F:3B:4E:64'
IPs='192.168.1.1'
IPd='www.iut-velizy.uvsq.fr'
pkt=Ether(src=MACs, dst=MACd)/IP(src=IPs, dst=IPd)/TCP(flags='SA')/"C est␣
 ↪vraiment bien Scapy"
pkt.show()
pkt.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.iut-velizy.uvsq.fr')
     \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    = 0xdcd6
     src       = 192.168.1.1
     dst       = 193.51.27.3
     \options   \
###[ TCP ]###
        sport     = ftp_data
        dport     = http
        seq       = 0
        ack       = 0
        dataofs   = 5
        reserved  = 0
        flags     = SA
        window    = 8192
        chksum    = 0xfb4f
        urgptr    = 0
        options   = []
###[ Raw ]###
           load      = 'C est vraiment bien Scapy'
```

3.2 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.

[4]:
```
paquets=rdpcap("Wireshark/Ping_Google.pcapng")
print("La capture comprend les paquets suivants :\n")
paquets.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
```

4 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.

[10]:
```
RT={'Guillemin' : 'chef de departement', 'Soulayrol' :'enseignant',␣
 ↪'Maingreaud' : 'secretaire' }
RT['Guillemin']
```

[10]:
```
'chef de departement'
```

[11]:
```
RT['Marty']='enseignant'
[print(f"Mr {nom} est {RT[nom]} au département R&T de l'IUT de Vélizy") for nom␣
 ↪in RT]
```

```
Mr Guillemin est chef de departement au département R&T de l'IUT de Vélizy
Mr Soulayrol est enseignant au département R&T de l'IUT de Vélizy
Mr Maingreaud est secretaire au département R&T de l'IUT de Vélizy
Mr Marty est enseignant au département R&T de l'IUT de Vélizy
```

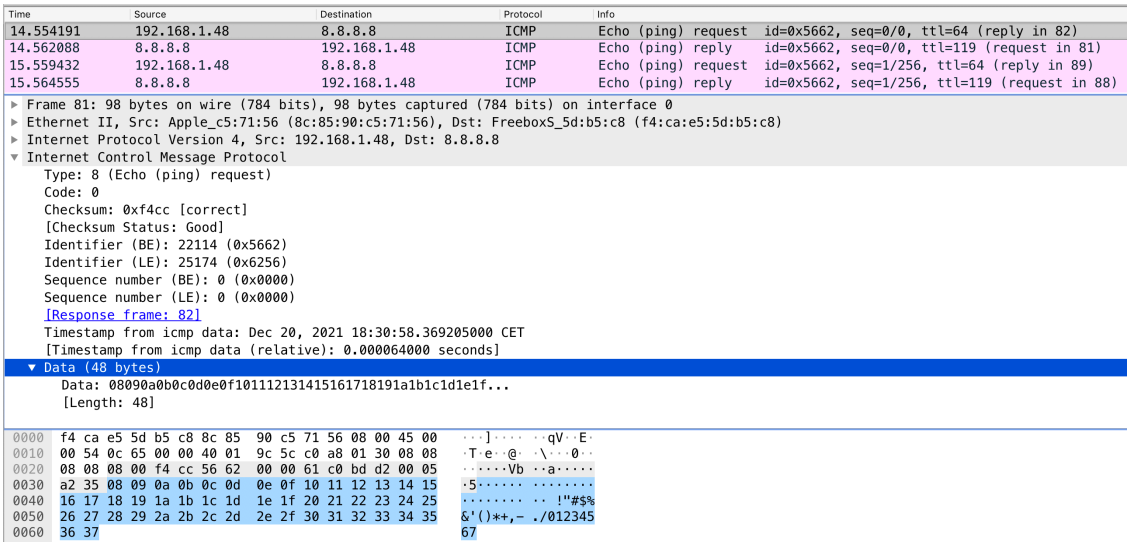
[11]:
```
[None, None, None, None]
```

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

[12]:
```
RT={'Guillemin' : {'fonction' : 'chef de departement', 'année de recutement' :␣
 ↪2002, \
                    'Enseignant en' : 'Réseaux, télécom'},
    'Marty' : {'fonction' : 'Responsable de l\'alternance', 'année de␣
 ↪recutement' : 2010, \
                    'Enseignant en' : 'Réseaux'}}
RT['Guillemin']['fonction']
```

[12]:
```
'chef de departement'
```

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 :

[13]:
```
print('Le premier paquet est le suivant :')
paquets[0]
```

```
Le premier paquet est le suivant :
```

[13]:
```
<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 |<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\x1f
!"#$%&\'()*+,-./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 :

[5]:
```
paquets[2]['IP']
```

[5]:
```
<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 |<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\x1
f !"#$%&\'()*+,-./01234567' |>>>
```

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

[15]:
```
paquets[2][IP]
```

[15]:
```
<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 |<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\x1
f !"#$%&\'()*+,-./01234567' |>>>
```

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

[16]:
```
paquets[2][IP].dst
```

[16]:
```
'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 :

[17]:
```
paquets[0][Raw].load
```

[17]:
```
b'a\xc0\xbd\xd2\x00\x05\xa25\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'
```

C'est aussi le cas du protocole SIP. Comme le contenu du protocole SIP est en mode texte, on pourra cependant récupérer les différents champs mais on devra parser les champs manuellement en utilisant les différentes fonctions (méthodes) fournies par les chaîne de caractère notamment la fonction split qui permet de séparer les différents élements en fonction d'un séparateur, ici l'espace (sep=None). Comme le protocole SIP est en mode texte avec un encodage UTF8, on peut ensuite décoder un éléments de la liste avec la fonction decode.

[7]:
```
paquets2=rdpcap("Wireshark/SIP.pcapng")
paquets2[0][Raw]
```

[7]:
```
<Raw  load='REGISTER sip:192.168.51.234:5060 SIP/2.0\r\nVia: SIP/2.0/UDP
192.168.35.42:5060;branch=z9hG4bK32fe687e88f536aca\r\nMax-Forwards: 70\r\nFrom:
"e1_tel1" <sip:e1_tel1@192.168.51.234:5060>;tag=cbd9deb1a0\r\nTo: "e1_tel1"
<sip:e1_tel1@192.168.51.234:5060>\r\nCall-ID: 09873fddf9ec647f\r\nCSeq:
2090097983 REGISTER\r\nAccept-Language: en\r\nAllow: INVITE, ACK, CANCEL, BYE,
NOTIFY, REFER, OPTIONS, UPDATE, PRACK, SUBSCRIBE, INFO, PUBLISH\r\nAllow-Events:
talk, hold, conference, LocalModeStatus\r\nAuthorization: Digest username="e1_te
l1",realm="asterisk",nonce="274fcc0b",uri="sip:192.168.51.234:5060",response="4f
b0cc1fbb5111ca76e9c147bc6bcabe",algorithm=MD5\r\nContact: "e1_tel1" <sip:e1_tel1
@192.168.35.42:5060;transport=udp>;+sip.instance="<urn:uuid:00000000-0000-1000-8
000-00085D6BA371>"\r\nSupported: path, gruu\r\nUser-Agent: Aastra
6863i/4.1.0.156\r\nContent-Length: 0\r\n\r\n' |>
```

Si le protocole est en mode texte comme SIP, 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 : sep = None) dans une liste :

[97]:
```
paquets2[0][Raw].load.split(sep=None)
```

```
[97]: [b'REGISTER',
       b'sip:192.168.51.234:5060',
       b'SIP/2.0',
       b'Via:',
       b'SIP/2.0/UDP',
       b'192.168.35.42:5060;branch=z9hG4bK32fe687e88f536aca',
       b'Max-Forwards:',
       b'70',
       b'From:',
       b'"e1_tel1"',
       b'<sip:e1_tel1@192.168.51.234:5060>;tag=cbd9deb1a0',
       b'To:',
       b'"e1_tel1"',
       b'<sip:e1_tel1@192.168.51.234:5060>',
       b'Call-ID:',
       b'09873fddf9ec647f',
       b'CSeq:',
       b'2090097983',
       b'REGISTER',
       b'Accept-Language:',
       b'en',
       b'Allow:',
       b'INVITE,',
       b'ACK,',
       b'CANCEL,',
       b'BYE,',
       b'NOTIFY,',
       b'REFER,',
       b'OPTIONS,',
       b'UPDATE,',
       b'PRACK,',
       b'SUBSCRIBE,',
       b'INFO,',
       b'PUBLISH',
       b'Allow-Events:',
       b'talk,',
       b'hold,',
       b'conference,',
       b'LocalModeStatus',
       b'Authorization:',
       b'Digest',
       b'username="e1_tel1",realm="asterisk",nonce="274fcc0b",uri="sip:192.168.51.234:
      5060",response="4fb0cc1fbb5111ca76e9c147bc6bcabe",algorithm=MD5',
       b'Contact:',
       b'"e1_tel1"',
       b'<sip:e1_tel1@192.168.35.42:5060;transport=udp>;+sip.instance="<urn:uuid:00000
      000-0000-1000-8000-00085D6BA371>"',
```

```
 b'Supported:',
 b'path,',
 b'gruu',
 b'User-Agent:',
 b'Aastra',
 b'6863i/4.1.0.156',
 b'Content-Length:',
 b'0']
```

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

[19]:
```
paquets2[0][Raw].load.split(sep=None)[0]
```

[19]:
```
b'REGISTER'
```

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 :

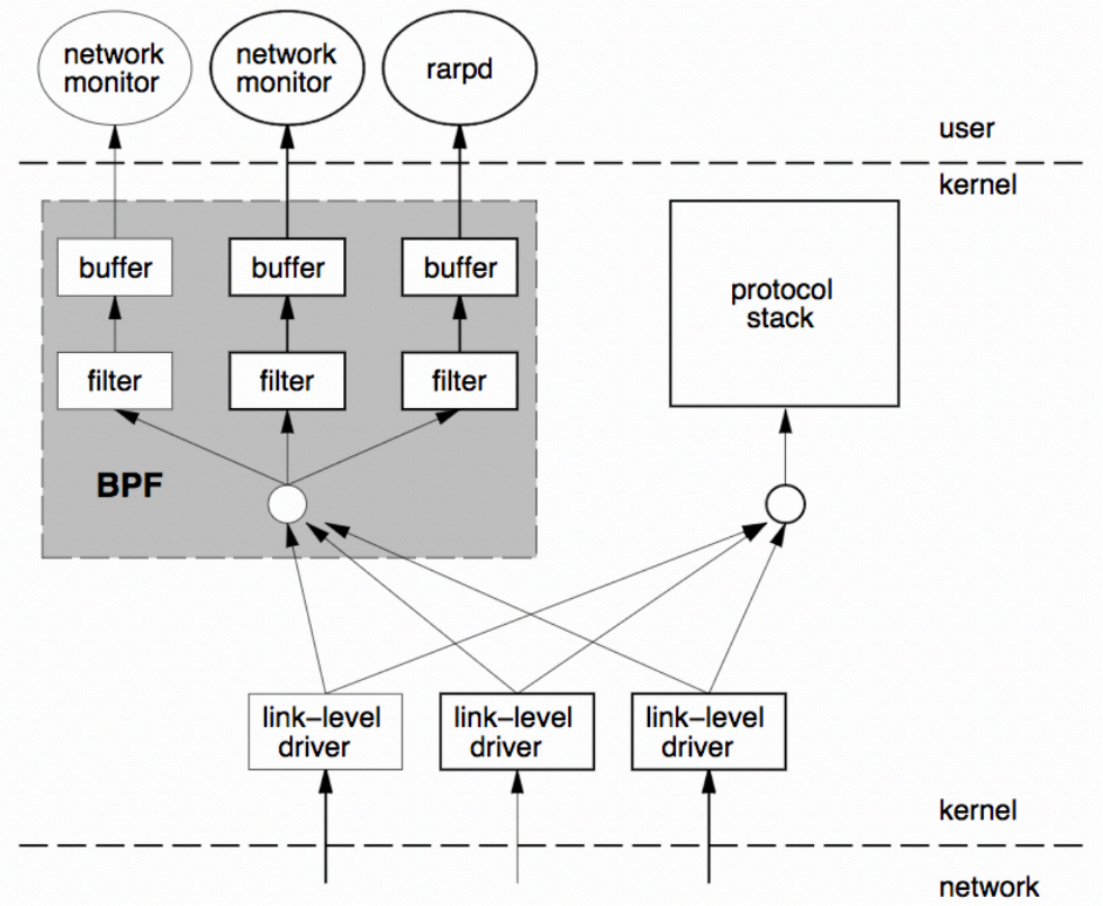
[20]:
```
paquets2[0][Raw].load.split(sep=None)[0].decode('UTF8')
```

[20]:
```
'REGISTER'
```

5 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)

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 :

[3]:
```
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)
```

```
Emission d'un paquet ICMP Echo vers 8.8.8.8
Réception d'un paquet ICMP Echo-Reply en provenance de 8.8.8.8
Emission d'un paquet ICMP Echo vers 8.8.8.8
Réception d'un paquet ICMP Echo-Reply en provenance de 8.8.8.8
```

[3]:
```
<Sniffed: TCP:0 UDP:0 ICMP:0 Other:0>
```

[ ]: