Excerpts from
Sections 1, 6-10, 12-13 and 15

Scapy documentation (!)

Philippe BIONDI
phil(at)secdev.org

1 Introduction

1.1 About this document

1.2 What is Scapy

Scapy is a Python program that enables you to forge, dissect, emit or sniff network packets, probe, scan or attack networks.

1.3 Scapy's concepts

1.3.1 Fast packet design

Many tools stick to the *program-that-you-run-from-a-shell* paradigm. The result is an awful syntax to describe a packet. The solution adopted was to use a higher but less powerful description, in the form of scenarii imagined by the tool's author. For instance you only need to give and IP to a port scanner to trigger the *port scanning* scenario. Even if you can tweak the scenario a bit, you still are stuck to a port scan.

Scapy paradigm is to propose a DSL[1] that enables a powerful and fast description of any kind of packet. The trick to use the Python syntax and a Python interpreter as the DSL syntax and the DSL interpreter has many advantages: there is no need to write the interpreter, users don't need to learn another language again and they benefit from a complete, concise and very powerful language.

Scapy enables you to describe a packet or a set of packets as layers that you stack one upon the other. Fields of each layer have useful default values that you can overload.

Scapy does not oblige you to use predetermined scenarii or templates. This means that each time you want to send packets, you have to write a new tool. In C, it would take an average of 60 lines only to describe the packet. With scapy, you only need one line to describle the packets you want to send, and one line to print the result. 90% of the network probing tools can be rewritten in 2 lines of Scapy.

1.3.2 Probe once, interpret many

Network discovery is blackbox testing. When probing a network, you will send many stimuli, and some of them will be answered. If you choose the right stimuli, you can obtain the information you need from the responses or from the lack of responces. Unlike many tools, Scapy will give you all the information, i.e. all the stimuli you sent and all the responses you got. You will have to look into them to get the information you are looking for. When the dataset is small, you can just dig for it. In other cases, you will have to choose a point of view on this data. Most tools choose the point of view for you and loose all the data not related to the given point of view. Because Scapy give you the whole raw data, you can use them many times and have your point of view evolve during your analysis. For example, you can probe for a TCP port scan, visualize the data like the result of a port scan, then decide you would like to also visualize the TTL of response packet. You do not need to do a new probe each time you want to visualize other data.

1.3.3 Scapy decodes, it does not interpret

A common problem in network probing tools is that they try to interpret the answers they got instead of only decoding and giving facts. Saying something like *I received a TCP Reset on port 80* is not subject to interpretation errors. Saying *The port 80 is closed* is an interpretation that can be right most of the

[1] Domain Specific Language.

time but wrong in some specific contexts the tool's author did not thought of. For instance, some scanners tend to report a filtered TCP port when they receive an ICMP destination unreachable packet. This may be right, but in some cases it means the packet was not fitered by the firewall but there was no host to forward the packet to.

Interpretating results can help people that don't know what a port scan is. But it can also make more harm than good, as it bias the results. In fact, what happen to people that know exactly what they are doing and that know very well thier tools is that they try to reverse the tool's interpretation to get the facts that triggered the interpretation, in order to do the interpretation themeselves. Too bad so much information has been lost in the operation.

6

Packet manipulation

6.0.4 User's methods

constructor

stacking

testing

reaching

haslayer

haslayer(self, cls)
True if self has a layer that is an instance of cls. Superseded by `cls in self` syntax.

getlayer

getlayer(self, cls, nb=*1*)
Returns the nbth layer that is an instance of cls.

psdump

psdump(self, filename=*None*) Creates an EPS file describing a packet. If filename is not provided a temporary file is created and gs is called.

pdfdump

pdfdump(self, filename=*None*)
Creates a PDF file describing a packet. If filename is not provided a temporary file is created and xpdf is called.

hide_defaults

hide_defaults(self)
Remove fields' values that are the same as default values.

show

show(self, indent=*3*, lvl=*''*, label_lvl=*''*)
Print a hierarchical view of the packet.

indent: gives the size of indentation for each layer.

show2

show2(self)
Prints a hierarchical view of an assembled version of the packet, so that automatic fields are calculated (checksums, etc.)

sprintf

sprintf(self, fmt, relax=*1*)
where format is a string that can include directives. A directive begins and ends by % and has the following format `%[fmt[r],][cls[:nb].]field%`.

fmt is a classic printf directive, "r" can be appended for raw substitution (ex: `IP.flags=0x18` instead of `SA`), nb is the number of the layer we want (ex: for IP/IP packets, `IP:2.src` is the `src` of the upper IP layer). Special case : `%.time%` is the creation time.

```
p.sprintf("%.time% %-15s,IP.src% -> %-15s,IP.dst% %IP.chksum% "
          "%03xr,IP.proto% %r,TCP.flags%")
```

Moreover, the format string can include conditionnal statements. A conditionnal statement looks like : {`layer:string`} where layer is a layer name, and string is the string to insert in place of the condition if it is true, i.e. if layer is present. If layer is preceded by a "!", the result si inverted. Conditions can be imbricated. A valid statement can be :

```
p.sprintf("This is a{TCP: TCP}{UDP: UDP}{ICMP:n ICMP} packet")
p.sprintf("{IP:%IP.dst% {ICMP:%ICMP.type%}{TCP:%TCP.dport%}}")
```

A side effect is that, to obtain { and } characters, you must use `%(` and `%)`.

decode_payload_as

decode_payload_as(self, cls)
Reassembles the payload and decode it using another packet class.

command

command(self)
Returns a string representing the command you have to type to obtain the same packet.

copy
copy(self)
Returns a deep copy of the instance.

lastlayer
lastlayer(self, layer=*None*)
Returns the uppest layer of the packet.

6.0.5 Developer's methods

hashret
hashret(self)
Must returns a string that has the same value for a request and its answer. The result of this function is used as a hash value to speed up the look for a the request matching a given response.

guess_payload_class
guess_payload_class(self, payload)
Guesses the next payload class from layer bonds. Can be overloaded to use a different mechanism.

default_payload_class
default_payload_class(self, payload)
Returns the default payload class if nothing has been found by the **guess_payload_class**() method.

my_summary
mysummary(self)
Can be overloaded to return a string that summarizes the layer. Only one **mysummary**() is used in a whole packet summary: the one of the upper layer, except if a **mysummary**() also returns (as a couple) a list of layers whose **mysummary**() must be called if they are present.

dissection_done
dissection_done(self, pkt)
will be called after a dissection is completed

post_dissection
post_dissection(self, pkt)
is called right after the dissection of the current layer

post_build
post_build(self, pkt)
called right after the current layer is build

extract_padding
get_field(self, fld)
returns the field instance from the name of the field

answers

answers(self, other)
True if self is an answer from other

haslayer_str

haslayer_str(self, cls)
True if self has a layer that whose class name is cls.

7

Packet list manipulation

7.1 User's methods

show

show(self)
Best way to display the packet list. Defaults to nsummary() method

nzpadding

nzpadding(self, lfilter=*None*)
same as **padding**() but only non null padding

make_tex_table

make_tex_table(self)
same as **make_table**(), but print a table with LaTeXsyntax

plot

plot(self, f, lfilter=*None*)
Apply a function to each packet to get a value that will be plotted with GnuPlot. A gnuplot object is returned.

lfilter: a truth function that decides whether a packet must be ploted

hexdump

hexdump(self)
Print an hexadecimal dump of each packet in the list

hexraw

hexraw(self, lfilter=*None*)
Same as **nsummary**(), except that if a packet has a **Raw** layer, it will be hexdumped

lfilter: a truth function that decides whether a packet must be ploted

pdfdump

pdfdump(self, filename=*None*)
create a PDF file with a **pdfdump**() of every packet.

filename: name of the file to write to. If empty, a temporary file is used and conf.prog.pdfreader is called.

nsummary

nsummary(self, prn=*None*, lfilter=*None*)
print a summary of each packet with the packet's number

prn: function to apply to each packet instead of `lambda x:x.summary()`

lfilter: truth function to apply to each packet to decide whether it will be displayed

conversations

conversations(self, getsrcdst=*None*, prog=*None*, type=*'svg'*, target=*'| display'*)
Graph a conversations between sources and destinations and display it (using graphviz and imagemagick)

getsrcdst: a function that takes an element of the list and return the source and dest by defaults, return source and destination IP

type: output type (svg, ps, gif, jpg, etc.), passed to dot's `-T` option

target: filename or redirect. Defaults pipe to Imagemagick's display program

make_lined_table

make_lined_table(self)
Same as **make_table**(), but print a table with lines.

padding

padding(self, lfilter=*None*)
Same as **hexraw**(), for **Padding** layer.

psdump

psdump(self, filename=*None*)
Creates a multipage poscript file with a **psdump**() of every packet

filename: name of the file to write to. If empty, a temporary file is used and conf.prog.psreader is called.

sr

sr(self, multi=*0*)
Match packets in the list and return ((*matchedcouples*), (*unmatchedpackets*))

summary

summary(self, prn=*None*, lfilter=*None*)
print a summary of each packet

prn: function to apply to each packet instead of `lambda x:x.summary()`

lfilter: truth function to apply to each packet to decide whether it will be displayed

filter

filter(self, func)
Return a packet list filtered by a truth function

make_table

make_table(self)
Print a table using a function that returs for each packet its head column value, head row value and displayed value.

```
p.make_table(lambda x:(x[IP].dst, x[TCP].dport, x[TCP].sprintf("%flags%"))
```

display

display(self)
deprecated. is show()

timeskew_graph

timeskew_graph(self, ip)
Try to graph the timeskew between the timestamps and real time for a given IP.

8 Commands

8.1 Tools

ls

ls(obj=*None*)
Lists available layers, or infos on a given layer

```
>>> ls()
ARP : ARP
BOOTP : BOOTP
CookedLinux :  cooked linux
DNS : DNS
GRE : GRE
[...]
>>> ls(Ether)
dst :  DestMACField = (None)
src :  SourceMACField = (None)
type :  XShortEnumField = (0)
>>> a=Ether()/Dot1Q(type=0x1234)
>>> ls(a)
dst :  DestMACField = 'ff:ff:ff:ff:ff:ff' (None)
src :  SourceMACField = None (None)
type :  XShortEnumField = 33024 (0)
--
prio :  BitField = 0 (0)
id :  BitField = 0 (0)
vlan :  BitField = 1 (1)
type :  XShortEnumField = 4660 (0)
```

lsc

lsc(cmd=*None*)
Lists documented commands

```
>>> lsc()
sr :  Send and receive packets at layer 3
sr1 :  Send packets at layer 3 and return only the first answer
[...]
```

hexdump

hexdump(x)
Prints an hexadecimal dump of a string or a packet

```
>>> hexdump(Ether(type=0x1234,dst="ba:be:fe:ed:be:ef")/IP())
0000 BA BE FE ED BE EF 00 00 00 00 00 00 12 34 45 00 .............4E.
0010 00 14 00 01 00 00 40 00 7C E7 7F 00 00 01 7F 00 ......@.|.......
0020 00 01 ..
```

linehexdump

linehexdump(x, onlyasc=*0*)
Prints a one line hexadecimal view of a string or packet. If onlyasc is not null, it can be used as a filter for safe printing strings from untrusted source (SSID, etc).

```
>>> linehexdump("\x01\x23\x45\x67")
01 23 45 67 .#Eg
>>> linehexdump("\x01\x23\x45\x67",onlyasc=1)
.
```

save_session

save_session(fname, session=*None*, pickleProto=*-1*)
Saves in a file all the variables in the user scope (everything seen with the `dir()` command). This command is very handy but there are some annoying caveats that come from the Python `cpickle` module. For example, lambda functions can't be saved and loaded modules won't be reloaded. Another problem can arise from automatic fields like source IP that can change from one machine to another, so the session will seem altered.

```
>>> save_session("/tmp/session.scapy")
```

load_session

load_session(fname)
Loads a previously saved session, smashing everything in the current session.

```
>>> load_session("/tmp/session.scapy")
```

update_session

update_session(fname)
Tries to merge the current session with a previously saved session.

```
>>> update_session("/tmp/session.scapy")
```

import_hexcap

import_hexcap()

Expects lines on standard input copy/pasted from a `tcpdump -xX` output, parse them, and output a string you can dissect with the protocol of your choice.

```
>>> IP(import_hexcap())
0x0000:  4500 0054 0000 4000 4001 242a c0a8 080e E..T..@.@.$*....
0x0010:  4266 0b63 0800 81e0 112a 0000 442e 9ca2 Bf.c.....*..D...
0x0020:  0007 991a 0809 0a0b 0c0d 0e0f 1011 1213 ................
0x0030:  1415 1617 1819 1a1b 1c1d 1e1f 2021 2223 .............!"#
0x0040:  2425 2627 2829 2a2b 2c2d 2e2f 3031 3233 $%&'()*+,-./0123
0x0050:  3435 3637 4567
^D
<IP version=4L ihl=5L tos=0x0 len=84 id=0 flags=DF frag=0L ttl=64
proto=ICMP chksum=0x242a src=192.168.8.14 dst=66.102.11.99 options=''
|<ICMP type=echo-request code=0 chksum=0x81e0 id=0x112a seq=0x0 |<Raw
load='D.\x9c\xa2\x00\x07\x99\x1a\x08\t\n\x0b\x0c\r\x0e\x0f\x10\x11\x12\x13\x14\x15
\x16\x17\x18\x19\x1a\x1b\x1c\x1d\x1e\x1f !"#$%&\'()*+,-./01234567' |>>>
```

export_object

export_object(obj)

Prints a bas64-encoded bzipped pickled version of the object that can be copy/pasted and imported to another Scapy session. If the two sessions run on different machines, automatic fields like source IP may not be the same.

```
>>> export_object(IP(ttl=3))
eNpdVWtbE1cQluwmgShV6q2ttVaruAgJEvACeIOQwDoSloSxsRXpQhZma5LNZDcUqNtqC62l2ruX
3r/7U/qt/6izEDX1yZPn7HPOeed9Z85c7oYW5ubKpl2Zm4vpRoxbuu5xyGdFw8iibZWKLqs+h1Hx
vBJHQHExZpZs0/VWq5bLOVlupRYTY/VKOaqVzFWrxm1ZbNtCBlc45vNOTXtJMWp7meAwxrscbvf5
NQ2jRWvRrJc83g07MOSu8R4IobJY9rgDQ0cn+HVUK2bZ4r2oLtbMJd6HqmuvWbwf2uvz1N5kPF2u
UowPOHzQ5zfERKnIbzYd5516bcHSjYaAtxw+5PPbGu3OOh5QqQOVowMuH6a9qLi1BX5HlLneFvUR
VOST363Pu/NOsNmdVc9KV+rlhsmjDh/z+T1N7LWIkaTNx30+oaGayxsZ7gQdVd1YPssnoQuV9LjB
GrRiW7AVzwSudUEPhtP6eM7gU1DAUCbF3bAhW9nUiME9MB9Yum5wHBICGzNy8dTkjMEJ6BTDqUmD
e4VX1fN6nk/DHVQMfZL7YAkVXbiSOIbKhFzqh3ON0i3MAAyhMpMy+AxEMFrI5uNimc/CAVSv5OTJ
OahiNDOzkc7pBp8HEOOyDspbqfqYHA9BHKNT4l/AMQw3ZHtcrF4IHjEnhi7CIVQKovISHEc1DwK+
DIOojMnZFTiBUTO/FZ8xBngEDqMynkvzKPSiei0pkBS42Kob+XQqPjLBY9CPsS3dWSe74nEahjE8
Ukgkz3AGZgUrAsZBadzJOXXP4gnoRgWFSoeOwOnAVDpv8FVIBm8hYgBuYSg/w9cgjBHdSE2J+Eko
NSIOVfVczsIFUZ4SQVOwLlEJYm7AtMQXDZ6GXYHG+PYb5UCtSwKFqzXHczhPktMUpPEoz6BiJ11G
n69rsjkILdQLIboACo1L9uVgg7ohQmeglTRooyTsomnoIB32UT8coLOwTlNwiC7CYRqBMF2D43QJ
TtAV6KQEdNFJ6KYJ6KEuiNMQJCgOvTQKSboK/TQGg3QZhmgAhikNFygLOnXCNBlQoFNwg4ZhljJw
iwBMOg/z1ANL1AclmoQqnQOXUnCHTtel2I41pX4hyP1G3r/vcMHnG9qWu5L7DZc/CErJc1z+ULAF
jaQyb/o8S81F+aIcbzk85/NHYgPb+pLnEqfl18dmc2nOO17arsGbzULy5NSed5UFh4s+W42iDtF2
C1kUIZEFuu3Wy7wkWorNXaEJTQ7bPn/8Cvp24EbJqnBJoPb/tDtVz3YqbgNedrjisxO4sKMJXxV8
tHGTGSM1K8BzDXaIvYpGOgldn7OgemoTLMij6LJVcwXHy7RfMmue3Cb6TMlcek79icMrPq8GnSeE
4bKOU5vX7geZuBjc4k9fZOIE38FwONVc9mf5M22WP8fQZIbvmrN8DONjGf5Cvr5E1Vq2S7xuWsKs
CPPKltANn796JTxfB+GxqcT3Gxo3XmmQDYnfOLzp87eBm1eakuSBPGpEUJsaSfgf+vxdcKOlieB7
IQjZRf5Bbj2so+rZOpJ/HB/5d+ez9WeJB7jTrtieLWNpzSryT9IBOxwJW8kxi1ZxrjHEfvb5F4xW
zdVgmx+9FJh1jO3NGD+WOffE56caycj7lWS4/UaPTWrLkkyx32mL8cjdp5v/kJCQWPyDHtETfKOx
v55T/enzX6JUlr81apeoH8zSMVkK8r9JJhWzZGepIiniynuvyO5GljYlJg+hRYos8R8yYmBq
```

import_object

import_object(obj=*None*)

Reads a base64-encoded bzipped pickled object on standard input.

```
>>> import_object()
eNpdVWtbE1cQluwmgShV6q2ttVaruAgJEvACeIOQwDoSloSxsRXpQhZma5LNZDcUqNtqC62l2ruX
3r/7U/qt/6izEDX1yZPn7HPOeed9Z85c7oYW5ubKpl2Zm4vpRoxbuu5xyGdFw8iibZWKLqs+h1Hx
vBJHQHExZpZsO/VWq5bLOVlupRYTY/VKOaqVzFWrxm1ZbNtCBlc45vNOTXtJMWp7meAwxrscbvf5
NQ2jRWvRrJc83g07MOSu8R4IobJY9rgDQOcn+HVUK2bZ4r2oLtbMJd6HqmuvWbwf2uvz1N5kPF2u
UowPOHzQ5zfERKnIbzYd5516bcHSjYaAtxw+5PPbGu3OOh5QqQOVowMuH6a9qLi1BX5HlLneFvUR
VOST363Pu/NOsNmdVc9KV+rlhsmjDh/z+T1N7LWIkaTNx3O+oaGayxsZ7gQdVd1YPssnoQuV9LjB
GrRiW7AVzwSudUEPhtP6eM7gU1DAUCbF3bAhW9nUiME9MB9Yum5wHBICGzNy8dTkjMEJ6BTDqUmD
e4VX1fN6nk/DHVQMfZL7YAkVXbiSOIbKhFzqh3ONOi3MAAyhMpMy+AxEMFrI5uNimc/CAVSv5OTJ
OahiND0zkc7pBp8HEOOyDspbqfqYHA9BHKNT4l/AMQw3ZHtcrF4IHjEnhi7CIVQKovISHEc1DwK+
DIOojMnZFTiBUTO/FZ8xBngEDqMynkvzKPSieiOpkBS42Kob+XQqPjLBY9CPsS3dWSe74nEahjE8
Ukgkz3AGZgUrAsZBadzJOXXP4gnoRgWFSoeOwOnAVDpv8FVIBm8hYgBuYSg/w9cgjBHdSE2J+Eko
NSIOVfVczsIFUZ4SQVOwLlEJYm7AtMQXDZ6GXYHG+PYb5UCtSwKFqzXHczhPktMUpPEoz6BiJ11G
n69rsjkILdQLIboACo1L9uVgg7ohQmeglTRooyTsomnoIB32UT8coLOwTlNwiC7CYRqBMF2D43QJ
TtAV6KQEdNFJ6KYJ6KEuiNMQJCgOvTQKSboK/TQGg3QZhmgAhikNFygLOnXCNBlQoFNwg4ZhljJw
iwBMOg/z1ANL1AclmoQqnQOXUnCHTtel2I41pX4hyP1G3r/vcMHnG9qWu5L7DZc/CErJc1z+ULAF
jaQyb/o8S81F+aIcbzk85/NHYgPb+pLnEqfl18dmc2nO017arsGbzULy5NSed5UFh4s+W42iDtF2
C1kUIZEFuu3Wy7wkWorNXaEJTQ7bPn/8Cvp24EbJqnBJoPb/tDtVz3YqbgNedrjisxO4sKMJXxV8
tHGTGSM1K8BzDXaIvYpGOgldn70gemoTLMij6LJVcwXHy7RfMmue3Cb6TM1cek79icMrPq8GnSeE
4bKOU5vX7geZuBjc4k9fZOIE38FwONVc9mf5M22WP8fQZIbvmrN8DONjGf5Cvr5E1Vq2S7xuWsKs
CPPKltANn796JTxfB+GxqcT3Gxo3XmmQDYnfOLzp87eBm1eakuSBPGpEUJsaSfgf+vxdcKOlieB7
IQjZRf5Bbj2so+rZOpJ/HB/5d+ez9WeJB7jTrtieLWNpzSryT9IBOxwJW8kxi1ZxrjHEfvb5F4xW
zdVgmx+9FJh1jO3NGD+WOffE56caycj7lWS4/UaPTWrLkkyx32mL8cjdp5v/kJCQWPyDHtETfKOx
v55T/enzX6JUlr81apeoH8zSMVkK8r9JJhWzZGepIiniynuvyO5GljYlJg+hRYos8R8yYmBq
^D
<IP ttl=3 |>
```

rdpcap

rdpcap(filename, count=*-1*)
reads a pcap file and returns the list of read packets. If count is positive, only the first count packets are read.

```
>>> rdpcap("ike.cap")
<ike.cap:  UDP:45 TCP:0 ICMP:0 Other:0>
>>> rdpcap("ike.cap", count=10)
<ike.cap:  UDP:10 TCP:0 ICMP:0 Other:0>
```

wrpcap

wrpcap(filename, pkt, linktype=*None*)
Write a packet or list of packets to a pcap file. linktype can be used to force the link type value written into the file.

```
>>> wrpcap("my.cap", packet_list)
```

fragment

fragment(pkt, fragsize=*1480*)
Fragments an IP packet with a large payload into many IP packets with a smaller payload and with the `fragofs` and `flags` fields correctly set up.

```
>>> fragment(IP(dst="1.2.3.4")/ICMP()/("X"*50), fragsize=10)
[<IP flags=MF frag=0 proto=ICMP dst=1.2.3.4 |<Raw load='\x08\x00W_\x00\x00\x00\x00XXXXXXXX'
|>>, <IP flags=MF frag=2 proto=ICMP dst=1.2.3.4 |<Raw load='XXXXXXXXXXXXXXXX'
|>>, <IP flags=MF frag=4 proto=ICMP dst=1.2.3.4 |<Raw load='XXXXXXXXXXXXXXXX'
|>>, <IP flags= frag=6 proto=ICMP dst=1.2.3.4 |<Raw load='XXXXXXXXXX' |>>]
```

fuzz

fuzz(p) Transform a packet into a fuzzed packet. Whas is done is replacing each field whose default value is not automatically calculated by a random value compatible with the field. Values that have to be automatically calculated (like checksums or lengths) are identified by the fact their default value is `None`. This characterization is a good approximation but keep in mind it could be wrong. Because the random value is considered as a default value on fuzzed packets, values set by the user or the upper layer overload the random value.

```
>>> ls( UDP() )
sport :  ShortEnumField = 53 (53)
dport :  ShortEnumField = 53 (53)
len :  ShortField = None (None)
chksum :  XShortField = None (None)
>>> ls( fuzz(UDP()) )
sport :  ShortEnumField = <RandShort> (53)
dport :  ShortEnumField = <RandShort> (53)
len :  ShortField = None (None)
chksum :  XShortField = None (None)
```

```
>>> a=IP(version=4)/UDP()
>>> hexdump(a)
0000 45 00 00 1C 00 01 00 00 40 11 7C CE 7F 00 00 01 E.......@.|.....
0010 7F 00 00 01 00 35 00 35 00 08 01 72 .....5.5...r
>>> hexdump(a)
0000 45 00 00 1C 00 01 00 00 40 11 7C CE 7F 00 00 01 E.......@.|.....
0010 7F 00 00 01 00 35 00 35 00 08 01 72 .....5.5...r
>>> b=fuzz(a)
>>> hexdump(b)
0000 47 DE 00 24 D3 E9 40 00 DE 11 42 FF C0 A8 08 0E G..$..@...B.....
0010 63 7E DF 17 0E ED 74 01 F4 C6 00 00 97 14 24 5E c~....t.......$^
0020 00 08 39 1F ..9.
>>> hexdump(b)
0000 4B 0D 00 34 64 CE 60 00 95 11 FD CE C0 A8 08 0E K..4d.`.........
0010 80 5E B7 6B 57 B9 BA 1D 58 C4 91 BD E6 E4 DB 39 .^.kW...X......9
0020 61 58 0E E6 FF 3E CF 98 5F 00 00 00 2B F3 B5 C5 aX...>.._...+...
0030 00 08 1D A5 ....
```

interact

interact(mydict=*None*, argv=*None*, mybanner=*None*, loglevel=*1*)
Runs an interactive session with completion and history. Use it in your own tools.

bind_top_down

bind_top_down(lower, upper, fval)
Informs upper layer that, when stacked on lower, it must overload lower's fields whose names are the keys of the fval dictionnary with their associated values.

```
>>> a=IP()/UDP()
>>> a.ttl
64
>>> bind_top_down(IP, UDP, {'ttl':12} )
>>> a=IP()/UDP()
>>> a.ttl
12
```

bind_bottom_up

bind_bottom_up(lower, upper, fval)
Informs lower layer that, when dissected, if all of its fields match the fval dictionnary, the payload is upper

```
>>> UDP("ABCDEFGHIJKL")
<UDP sport=16706 dport=17220 len=17734 chksum=0x4748 |<Raw load='IJKL' |>>
>>> bind_bottom_up(UDP, Dot1Q, {"dport":17220} )
>>> UDP("ABCDEFGHIJKL")
<UDP sport=16706 dport=17220 len=17734 chksum=0x4748 |<Dot1Q prio=2L id=0L
vlan=2378L type=0x4b4c |>>
```

bind_layers

bind_layers(lower, upper, fval)
Does a **bind_bottom_up** and a **bind_top_down**.

8.2 Communication commands

8.2.1 Sniff family

sniff

sniff(prn=*None*, lfilter=*None*, count=*0*, store=*1*, offline=*None*, L2socket=*None*, timeout=*None*) Sniffs packets from the network and return them in a packet list. This function can have many parameters:

count: number of packets to capture. 0 means infinity.

store: wether to store sniffed packets or discard them. When you only want to monitor your network forever, set store to *0*.

prn: function to apply to each packet. If something is returned, it is displayed. For instance you can use `prn = lambda x: x.summary()`.

lfilter: python function applied to each packet to determine. if further action may be done. For instance, you can use `lfilter = lambda x: x.haslayer(Padding)`

offline: pcap file to read packets from, instead of sniffing them. In this case, BPF filter won't work.

timeout: stop sniffing after a given time (default: *None*).

L2socket: you can provide a supersocket for sniffing instead of the one from `conf.L2listen`.

8.2.2 Send family

send

send(pkts, inter=*0*, loop=*0*, verbose=*None*)
Send packets at layer 3, using the `conf.L3socket` supersocket. pkts can be a packet, an implicit packet or a list of them.

loop: send the packets endlessly if not *0*.

inter: time in seconds to wait between 2 packets

verbose: override the level of verbosity. Make the function totally silent when *0*

sendp

sendp(pkts, inter=*0*, loop=*0*, iface=*None*, iface_hint=*None*, verbose=*None*)
Send packets at layer 2 using the `conf.L2socket` supersocket. pkts can be a packet, an implicit packet or a list of them.

loop: send the packets endlessly if not *0*.

inter: time in seconds to wait between 2 packets

verbose: override the level of verbosity. Make the function totally silent when *0*

8.2.3 Send and receive family

The *send and receive* functions family is central to interact with the network with Scapy.

sr

sr(pkts, filter=*None*, iface=*None*, timeout=*2*, inter=*0*, verbose=*None*, chainCC=*0*, retry=*0*, multi=*0*) Send and receive packets at layer 3 using the `conf.L3socket` supersocket.

nofilter: put 1 to avoid use of bpf filters on systems that don't support it

retry: if positive, how many times to resend unanswered packets if negative, how many consecutive unanswered probes before giving up. Only the negative value is really useful.

timeout: how much time to wait after the last packet has been sent. By default, sr will wait forever and the user will have to interrupt (Ctrl-C) it when he expects no more answers.

verbose: set verbosity level.

multi: whether to accept multiple answers for the same stimulus.

filter: provide a BPF filter.

iface: listen answers only on the provided interface.

inter: time in seconds to wait between each packet sent.

chainCC: when Ctrl-C is pressed, raise the `KeyboardInterrupt` exception again so that callers can know it and behave accordingly.

sr1

sr1(pkts, filter=*None*, iface=*None*, timeout=*2*, inter=*0*, verbose=*None*, chainCC=*0*, retry=*0*, multi=*0*) Same as sr except only the first answer is returned. This command is very useful for one packet probes like pinging an IP.

```
>>> sr1(IP(dst="192.168.0.1")/ICMP())
<IP version=4L ihl=5L tos=0x0 len=28 id=34897 flags= frag=0L ttl=64 proto=ICMP
chksum=0x713d src=192.168.0.1 dst=192.168.0.1 options='' |<ICMP type=echo-reply
code=0 chksum=0xffff id=0x0 seq=0x0 |>>
```

srp

srp(pkts, filter=*None*, iface=*None*, timeout=*2*, inter=*0*, verbose=*None*, chainCC=*0*, retry=*0*, multi=*0*, iface_hint=*None*)
Same as srp but for working at layer 2 with `conf.L2socket` supersocket. There is also an additional parameter, iface_hint, which give an hint that can help choosing the right output interface. By default, if not specified by iface, `conf.iface` is chosen. The hint takes the form of an IP to which the layer 2 packet might be destinated. The Scapy routing table (`conf.route`) is used to determine which interface to use to reach this IP.

srp1

srp(pkts, filter=*None*, iface=*None*, timeout=*2*, inter=*0*, verbose=*None*, chainCC=*0*, retry=*0*, multi=*0*, iface_hint=*None*)
Same as srp, except only the first answer is returned. This command is very useful for one packet probes like ARP pinging an IP.

srbt

srbt(peer, pkts, inter=*0.1*)
Same as sr but for the `conf.BTsocket` supersocket. It is a Bluetooth supersocket that needs the peer's address, provided by the peer parameter.

srloop

srloop(pkts, prn=*lambda x:x[1].summary()*,, prnfail=*lambda x:x.summary()*, inter=*1*, timeout=*None*, count=*None*, verbose=*0*)

Send in loop a packet or a set of packets with `conf.L3socket` supersocket and print the results at each round.

prn: a function to be applied to each couple (*packet sent*, *packet received*) whose result will be displayed.

prnfail: a function to be applied to each unanswered packet sent whose result will be displayed.

inter: time interval in seconds between two rounds

timeout: time to wait for answers after the last packet has been sent. By default, the timeout will be $\min(2 \times \text{inter}, 5)$

count: number of rounds. By default, runs forever.

verbose: control verbosity.

```
>>> srloop(IP(dst="192.168.0.1")/TCP(dport=[21,22,25,80]),count=3)
RECV 2:  IP / TCP 192.168.0.1:ssh > 192.168.0.1:ftp-data SA
IP / TCP 192.168.0.1:www > 192.168.0.1:ftp-data RA
fail 2:  IP / TCP 192.168.0.1:ftp-data > 192.168.0.1:smtp S
IP / TCP 192.168.0.1:ftp-data > 192.168.0.1:ftp S
RECV 2:  IP / TCP 192.168.0.1:ssh > 192.168.0.1:ftp-data SA
IP / TCP 192.168.0.1:www > 192.168.0.1:ftp-data RA
fail 2:  IP / TCP 192.168.0.1:ftp-data > 192.168.0.1:smtp S
IP / TCP 192.168.0.1:ftp-data > 192.168.0.1:ftp S
RECV 2:  IP / TCP 192.168.0.1:ssh > 192.168.0.1:ftp-data SA
IP / TCP 192.168.0.1:www > 192.168.0.1:ftp-data RA
fail 2:  IP / TCP 192.168.0.1:ftp-data > 192.168.0.1:smtp S
IP / TCP 192.168.0.1:ftp-data > 192.168.0.1:ftp S
Sent 12 packets, received 6 packets.  50.0% hits.
```

srploop

srploop(pkts)
Same as srloop but for layer 2.

8.3 high-level commands

ikescan

ikescan(ip)

traceroute

traceroute(target, dport=*80*, minttl=*1*, maxttl=*30*, sport=*¡RandShort¿*, l4=*None*, filter=*None*, timeout=*2*)

arping

arping(net, timeout=*2*)
Send ARP who-has requests to determine which hosts are up arping(net, iface=conf.iface) -¿ None

is_promisc

is_promisc(ip, fake_bcast=*'ff:ff:00:00:00:00'*)
Try to guess if target is in Promisc mode. The target is provided by its ip.

dhcp_request

dhcp_request(iface=*None*)

fragleak
fragleak(target, sport=*123*, dport=*123*, timeout=*0.2*, onlyasc=*0*)

fragleak2
fragleak2(target, timeout=*0.4*, onlyasc=*0*)

report_ports
report_ports(target, ports)
portscan a target and output a LaTeXtable

arpcachepoison
arpcachepoison(target, victim, interval=*60*)
Poison target's cache with (your MAC,victim's IP) couple

8.4 Answering machines

dhcpd

dns_spoof

airpwn

bootpd

farpd

9

Adding a new protocol

Adding new layer in Scapy is very easy. All the magic is in the fields. If the fields you need are already there and the protocol is not too brain-damaged, this should be a matter of minutes.

9.1 Definition of a layer

A layer is a subclass of the Packet class. All the logic behind layer manipulation is hold by the Packet class and will be inherited.

A simple layer is compounded by a list of fields that will be either concatenated when assembling the layer or dissected one by one when desassembling a string. The list of fiels is hold in an attribute named fields_desc. Each field is an instance of a field class.

```
1 class Disney(Packet):
2     name = "Disney Packet"
3     fields_desc = [ ShortField("mickey", 5),
4                     XByteField("minnie", 3),
5                     IntEnumField("donald", 1, {1:"happy",2:"cool",3:"angry"}) ]
```

In this example, our layer has three fields. The first one is an 2 byte integer field named mickey and whose default value is 5. The second one is a 1 byte

integer field named minnie and whose default value is 3. The difference between a vanilla ByteField and a XByteField is only the fact that the prefered human representation of the field's value is in hexadecimal. The last field is a 4 byte integer field named s donald. It is different from a vanilla IntField by the fact that some of the possible values of the field have litterate representations. For example, if it is worth 3, the value will be displayed as *angry*. Moreover, if the `"cool"` value is assigned to this field, it will understand that it has to take the value 2.

If your protocol is as simple as this, it is ready to use.

```
>>> d=Disney(mickey=1)
>>> ls(d)
mickey :  ShortField = 1 (5)
minnie :  XByteField = 3 (3)
donald :  IntEnumField = 1 (1)
>>> d.show()
###[ Disney Packet ]###
mickey= 1
minnie= 0x3
donald= happy
>>> d.donald="cool"
>>> str(d)
'\x00\x01\x03\x00\x00\x00\x02'
>>> Disney(_)
<Disney mickey=1 minnie=0x3 donald=cool |>
```

9.2 Fields

Many fields already exist. Some are very generic, as ByteField, and some very specific and used only in one layer, as TCPOptionsField.

FieldLenField The FieldLenField is a field whose value gives the length of another field. If the other field is a FieldListField or a PacketListField, the value is the number of elements of the list. Else, it correspond to the number of bytes belonging to the other field. The third parameter is the name of the other field. The default value should be `None` to indicate that it should be calculated automatically. A shift value can be given that need to be added to the value of the field to obtain the field value. This is needed when a field holds the length of a set of fields and must be adjusted to get only the variable field's length.

9.3 Layers' methods

9.4 Binding layers

9.5 Layers' design patterns

9.5.1 For a string whose length is given by another field

To do this, we will use the FieldLenField and the StrLenField. The FieldLenField is a field whose value gives the length of another field. Here, the other field is "the_string". The string field is also special because it needs to know its length from the lengh field when the packet is dissected. So, it references "the_length" as its third argument.

To make it short, when dissecting, "the_string" knows its lenght from freshly dissected "the_length" field and when assembling, "the_length" can be automatically computed from the length of "the_string".

```
1 class VarStr(Packet):
2     name = "Variable String"
3     fields_desc = [ FieldLenField("the_length",None,"the_string","I"),
4                     StrLenField("the_string","The default value", "the_length") ]
```

```
>>> p=VarStr()
>>> p.show2()
###[ Variable String ]###
the_length= 17L
the_string= 'The default value'
>>> p.the_string="The new value"
>>> p.show2()
###[ Variable String ]###
the_length= 13L
the_string= 'The new value'
>>> hexdump(p)
0000 00 00 00 0D 54 68 65 20 6E 65 77 20 76 61 6C 75 ....The new valu
0010 65 e
```

10

Adding a new field

A field is a class whose instance holds all the meta-informations relating to a given field in an layer class and is used as translating box.

The class field is responsible for both extracting the field value from the raw packet string being dissected and for adding the field in a raw packet string being assembled.

The value held into a field can have many forms. We have the assembled form, the internal form, the human readable form and a rich representation. For example, the TCP flags for a SYN-ACK packet will have the `"SA"` string as rich representation. In the packet, we will find the character `"\x12"`. But the internal useful value will be the integer 18. Each field will provide functions to translate values between all those representations, even if, in most of the cases, some of the representations will be identical. A special function is also here to try guess its input form to enable the user to fill a field with human readable or rich representation.

10.1 The **Field** API

addfield

addfield(self, pkt, s, val)
Adds the value val to the raw string packet s. The field belongs to the

Packet instance pkt.

getfield

getfield(self, pkt, s)
Extracts and returns the value of the field from the raw string packet s. The field belongs to the Packet instance pkt.

randval

randval(self)
Returns a VolatileValue subclass' instance whose values will be randomly chosen in the domain of the field.

copy

copy(self)
Returns a deep copy of the instance.

i2h

i2h(self, pkt, x)
Translates the internal value representation into the human readable representation.

h2i

h2i(self, pkt, x)
Translates the human readable representation into the internal representation.

m2i

m2i(self, pkt, x)
Translates the machine representation into the internal representation. The machine representation is the raw bytes found into the raw string packet.

any2i

any2i(self, pkt, x)
Try to guess the input representation and returns the internal representation.

i2m

i2m(self, pkt, x)
Translates the internal representation into the machine representation. The machine representation is the raw bytes found into the raw string packet.

i2repr

i2repr(self, pkt, x)
Translates the internal value to the rich representation.

i2len

i2en(self, pkt, x)

Computes the length of the field for it to be used in another field (usually, a length field). Depending on the field and the layer, the value can be for example a byte count or a number of elements in a list, and can even be shifted to fit the needs of the the length field.

12

Making your own tools

```
#! /usr/bin/env python

from scapy import *

class Test(Packet):
    name = "Test packet"
    fields_desc = [ ShortField("test1", 1),
                    ShortField("test2", 2) ]

def make_test(x,y):
    return Ether()/IP()/Test(test1=x,test2=y)

if __name__ == "main":
    interact(mydict=globals(), mybanner="Test add-on v3.14")
```

13

Scripting Scapy

```
1  #! /usr/bin/env python
2
3  import sys
4  if len(sys.argv) != 2:
5      print "Usage: arping <net>\n eg: arping 192.168.1.0/24"
6      sys.exit(1)
7
8  from scapy import srp,Ether,ARP,conf
9  conf.verb=0
10 ans,unans=srp(Ether(dst="ff:ff:ff:ff:ff:ff")
11                 /ARP(pdst=sys.argv[1]),
12                timeout=2)
13
14 for s,r in ans:
15     print r.sprintf("%Ether.src% %ARP.psrc%")
```

15

Networking code

15.1 Supersockets

There are many different ways to access the network even on the same OS. The abstraction for all of these in Scapy is supersockets.

A supersocket can be initialized with a BPF filter. It can be read with the `recv()` method and written with the `send()` method. In both cases, packets are provided, and the `recv()` method has to determine which layer is suitable for the considered link type. A supersocket also has a `fileno()` method for it to be `select`ed for reading. In the case there is one file descriptor for reading and one for writing, the first one must be returned.

If a supersocket is supposed to work at a given layer, it has to handle all the missing layers. For instance a layer 3 supersocket working with layer 2 sockets must handle the layer 2.

Functions that use supersockets (sr, sendp, sniff, etc.) choose the supersocket layer type they need in the `conf` variable. If they need a layer 3 supersocket, they will use `conf.L3socket`.

Here are some of the available supersockets:

L3RawSocket: a supersocket working for IP packets and using a `PF_INET/SOCK_RAW`

socket for sending and a `PF_PACKET` for receiving. Hence many limitations apply to sending. For example, if IP checksum is 0, it will be calculated, packets can be blocked by the local firewall, etc. But it has the advantage to work on the loopback interface.

L3PacketSocket: a supersocket working at layer 3 and using `PF_PACKET/SOCK_RAW` sockets for sending and receiving. It handles layer 2. This supersocket is the default choice on Linux.

L2Socket: a supersocket working at layer 2 and using `PF_PACKET/SOCK_RAW` sockets for sending and receiving. This supersocket is the default choice on Linux.

L2ListenSocket: This socket uses `PF_PACKET/SOCK_RAW` for sniffing use only.

L3dnetSocket: This supersocket uses libpcap for receiving and libdnet for sending. It works at layer 3 and handles layer 2.

L2dnetSocket: This supersocket uses libpcap for receiving and libdnet for sending. It works at layer 2.

L2pcapListenSocket: This supersocket use libpcap for sniffing uses only.

StreamSocket: This socket uses a kernel stream socket (TCP connexion, etc.) as a link layer. The layer class to use as the link layer protocol must be provided.

BluetoothL2CAPSocket: This socket is used to handle Bluetooth sockets at the L2CAP level.

BluetoothHCISocket: This socket is used to handle Bluetooth sockets at the HCI level. This level is not supported yet by Python socket module.

15.2 Routing packets

When Scapy is launched, its routing tables are synchronized with host's routing table. For a packet sent at layer 3, the destination IP determine the output interface, source address and gateway to be used. For a layer 2 packet, the output interface can be precised, or an hint can be given in the form of an IP to determine the output interface. If no output interface nor hint are given, `conf.iface` is used.

```
>>> conf.route
Network Netmask Gateway Iface Output IP
127.0.0.0 255.0.0.0 0.0.0.0 lo 127.0.0.1
172.16.15.0 255.255.255.0 0.0.0.0 eth0 172.16.15.42
0.0.0.0 0.0.0.0 172.16.15.1 eth0 172.16.15.42
>>> conf.route.add(net="192.168.1.0/24",gw="172.16.15.23")
>>> conf.route.add(host="192.168.4.5",gw="172.16.15.24")
>>> conf.route
Network Netmask Gateway Iface Output IP
127.0.0.0 255.0.0.0 0.0.0.0 lo 127.0.0.1
172.16.15.0 255.255.255.0 0.0.0.0 eth0 172.16.15.42
192.168.1.0 255.255.255.0 172.16.15.23 eth0 172.16.15.42
192.168.4.5 255.255.255.255 172.16.15.24 eth0 172.16.15.42
0.0.0.0 0.0.0.0 172.16.15.1 eth0 172.16.15.42
>>> conf.route.delt(net="192.168.1.0/24",gw="172.16.15.23")
>>> conf.route.resync()
>>> conf.route
Network Netmask Gateway Iface Output IP
127.0.0.0 255.0.0.0 0.0.0.0 lo 127.0.0.1
172.16.15.0 255.255.255.0 0.0.0.0 eth0 172.16.15.42
0.0.0.0 0.0.0.0 172.16.15.1 eth0 172.16.15.42
```