First steps

Let’s build a packet and play with it:

```
>>> a=IP(ttl=10)
>>> a
< IP ttl=10 |>
>>> a.src
’127.0.0.1’
>>> a.dst="192.168.1.1"
>>> a
< IP ttl=10 dst=192.168.1.1 |>
>>> a.src
’192.168.8.14’
>>> del(a.ttl)
>>> a
< IP dst=192.168.1.1 |>
>>> a.ttl
64
```

Stacking layers

The `/` operator has been used as a composition operator between two layers. When doing so, the lower layer can have one or more of its defaults fields overloaded according to the upper layer. (You still can give the value you want). A string can be used as a raw layer.

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

Sending packets¶

```
>>> send(IP(dst="1.2.3.4")/ICMP())
Exit
```

```
Sr1
```

SYN Scans

```
SYN scanning is a tactic that a malicious hacker (or cracker) can
use to determine the state of a communications port without
establishing a full connection.
```

From the above output, we can see Google returned “SA” or SYN-ACK flags indicating an open port.

Classic SYN Scan can be initialized by executing the following command from Scapy’s prompt:

```
>>> sr1(IP(dst=“72.14.207.99")/TCP(dport=80,flags="S"))
```

From the above output, we can see Google returned “SA” or SYN-ACK flags indicating an open port.
Use either notations to scan ports 400 through 443 on the system:

```
>>> sr(IP(dst="192.168.1.1")/
TCP(sport=666,dport=(440,443),flags="S"))
```

or

```
>>> sr(IP(dst=“192.168.1.1")/
TCP(sport=RandShort(),dport=[440,441,442,443],flags="S"))
```

```
In order to quickly review responses simply request a summary of
collected packets:
```

Sniffing

We can easily capture some packets or even clone tcpdump or tshark. Either one interface or a list of interfaces to sniff on can be provided. If no interface is given, sniffing will happen on every interface:

```
>>>  sniff(filter="icmp and host 66.35.250.151", count=2)
<Sniffed: UDP:0 TCP:0 ICMP:2 Other:0>
>>>  a=_
>>>  a.nsummary()
0000 Ether / IP / ICMP 192.168.5.21 echo-request 0 / Raw
0001 Ether / IP / ICMP 192.168.5.21 echo-request 0 / Raw
>>>  a[1]
```

```
sniff(iface="wifi0", prn=lambda x: x.summary())

sniff(iface="eth1", prn=lambda x: x.show())
```

For even more control over displayed information we can use the `sprintf()` function:

```
>>> pkts = sniff(prn=lambda x:x.sprintf("{IP:%IP.src% -> %IP.dst%
\n}{Raw:%Raw.load%\n}"))
```

Filters

Demo of both bpf filter and sprintf() method:

a=sniff(filter="tcp and (port 25 or port 110)")

Send and receive in a loop

Here is an example of a (h)ping-like functionality : you always send the same set of packets to see if something change:

```
>>> srloop(IP(dst="www.target.com/30")/TCP())
```

Importing and Exporting Data

PCAP

It is often useful to save capture packets to pcap file for use at later time or with different applications:

```
>>> wrpcap("temp.cap",pkts)
```

To restore previously saved pcap file:

```
>>> pkts = rdpcap("temp.cap")
```

or

```
>>> pkts = sniff(offline="temp.cap")
```

Making tables

Now we have a demonstration of the `make_table()` presentation function. It takes a list as

```
>>> ans, unans = sr(IP(dst="172.20.80.192/28")/
TCP(dport=[20,21,22,25,53,80]))
Received 142 packets, got 25 answers, remaining 71 packets
>>> ans.make_table(lambda (s,r): (s.dst, s.dport,
r.sprintf("%IP.id%")))
   172.20.80.196 172.20.80.197 172.20.80.198 172.20.80.200
172.20.80.201
20 0             4203          7021          -             11562
21 0             4204          7022          -             11563
22 0             4205          7023          11561         11564
```

Routing

Now scapy has its own routing table, so that you can have your packets routed differently than the system:

```
>>> conf.route
Network         Netmask         Gateway         Iface
127.0.0.0       255.0.0.0       0.0.0.0         lo
192.168.8.0     255.255.255.0   0.0.0.0         eth0
0.0.0.0         0.0.0.0         192.168.8.1     eth0
>>> conf.route.delt(net="0.0.0.0/0",gw="192.168.8.1")
>>> conf.route.add(net="0.0.0.0/0",gw="192.168.8.254")
>>> conf.route.add(host="192.168.1.1",gw="192.168.8.1")
>>> conf.route
Network         Netmask         Gateway         Iface
127.0.0.0       255.0.0.0       0.0.0.0         lo
192.168.8.0     255.255.255.0   0.0.0.0         eth0
0.0.0.0         0.0.0.0         192.168.8.254   eth0
192.168.1.1     255.255.255.255 192.168.8.1     eth0
>>> conf.route.resync()
>>> conf.route
Network         Netmask         Gateway         Iface
127.0.0.0       255.0.0.0       0.0.0.0         lo
192.168.8.0     255.255.255.0   0.0.0.0         eth0
0.0.0.0         0.0.0.0         192.168.8.1     eth0
```

Gnuplot

We can easily plot some harvested values using Gnuplot. (Make sure that you have exit-py and Gnuplot installed.) For example, we can observe the IP ID patterns to know how many distinct IP stacks are used behind a load balancer:

```
>>> a, b = sr(IP(dst="www.target.com")/
TCP(sport=[RandShort()]*1000))
>>> a.plot(lambda x:x[1].id)
<Gnuplot._Gnuplot.Gnuplot instance at 0xb7d6a74c>
```

TCP traceroute (2)

Scapy also has a powerful TCP traceroute function. Unlike other traceroute programs that wait for each node to reply before going to the next, scapy sends all the packets at the same time. This has the disadvantage that it can't know when to stop (thus the maxttl parameter) but the great advantage that it took less than 3 seconds to get this multi-target traceroute result:

```
>>>
traceroute(["www.yahoo.com","www.altavista.com","www.wisenut.com",
"www.copernic.com"],maxttl=20)
Received 80 packets, got 80 answers, remaining 0 packets
   193.45.10.88:80    216.109.118.79:80  64.241.242.243:80
66.94.229.254:80
1  192.168.8.1        192.168.8.1        192.168.8.1
192.168.8.1
2  82.243.5.254       82.243.5.254       82.243.5.254
82.243.5.254
```

```
>>> res, unans =
traceroute(["www.microsoft.com","www.cisco.com","www.yahoo.com","w
ww.wanadoo.fr","www.pacsec.com"],dport=[80,443],maxttl=20,retry=-2
)
Received 190 packets, got 190 answers, remaining 10 packets
   193.252.122.103:443 193.252.122.103:80 198.133.219.25:443
198.133.219.25:80  207.46...
1  192.168.8.1         192.168.8.1        192.168.8.1
192.168.8.1        192.16...
2  82.251.4.254        82.251.4.254       82.251.4.254
82.251.4.254       82.251...
3  213.228.4.254       213.228.4.254      213.228.4.254
213.228.4.254      213.22...
[...]
>>> res.graph()                          # piped to ImageMagick's
display program. Image below.
>>> res.graph(type="ps",target="| lp")   # piped to postscript
printer
>>> res.graph(target="> /tmp/graph.svg") # saved to file
```

```
res.trace3D()
```

Simple one-liners

ACK Scan

Using Scapy’s powerful packet crafting facilities we can quick replicate classic TCP Scans. For example, the following string will be sent to simulate an ACK Scan:

```
>>> ans, unans = sr(IP(dst="www.slashdot.org")/
TCP(dport=[80,666],flags="A"))
```

We can find unfiltered ports in answered packets:

```
>>> for s,r in ans:
...     if s[TCP].dport == r[TCP].sport:
...        print str(s[TCP].dport) + " is unfiltered"
```

Similarly, filtered ports can be found with unanswered packets:

```
>>> for s in unans:
...     print str(s[TCP].dport) + " is filtered"
```

IP Scan

A lower level IP Scan can be used to enumerate supported protocols:

```
>>> ans, unans =
sr(IP(dst="192.168.1.1",proto=(0,255))/"SCAPY",retry=2)
```

ARP Ping

The fastest way to discover hosts on a local ethernet network is to use the ARP Ping method:

```
>>> ans, unans = srp(Ether(dst="ff:ff:ff:ff:ff:ff")/
ARP(pdst="192.168.1.0/24"),timeout=2)
```

Answers can be reviewed with the following command:

```
>>> ans.summary(lambda (s,r): r.sprintf("%Ether.src%
%ARP.psrc%") )
```

Scapy also includes a built-in arping() function which performs similar to the above two commands:

```
>>> arping("192.168.1.*")
```

ICMP Ping

Classical ICMP Ping can be emulated using the following command:

```
>>> ans, unans = sr(IP(dst="192.168.1.1-254")/ICMP())
```

Information on live hosts can be collected with the following request:

```
>>> ans.summary(lambda (s,r): r.sprintf("%IP.src% is alive") )
```

TCP Ping

In cases where ICMP echo requests are blocked, we can still use various TCP Pings such as TCP SYN Ping below:

```
>>> ans, unans = sr( IP(dst="192.168.1.*")/TCP(dport=80,flags="S")
)
```

Any response to our probes will indicate a live host. We can collect results with the following command:

```
>>> ans.summary( lambda(s,r) : r.sprintf("%IP.src% is alive") )
```

UDP Ping

If all else fails there is always UDP Ping which will produce ICMP Port unreachable errors from live hosts. Here you can pick any port which is most likely to be closed, such as port 0:

```
>>> ans, unans = sr( IP(dst="192.168.*.1-10")/UDP(dport=0) )
```

Once again, results can be collected with this command:

```
>>> ans.summary( lambda(s,r) : r.sprintf("%IP.src% is alive") )
```

Python security

If you want this to work, you have to send your packets through the kernel using PF_ INET/SOCK_RAW sockets; i.e., by using L3RawSocket instead of L3PacketSocket or L3dnetSocket:

```
>>> sr1(IP()/ICMP(), timeout=1) Begin emission:
.Finished to send 1 packets.
Received 1 packets, got 0 answers, remaining 1 packets
```