

Python Scapy Quick-Reference

The scapy module is extremely useful, but the documentation is somewhat lacking. Consequently, here is a simple cookbook of handy scapy recipes.

Importing Scapy

Unlike most modules, scapy requires a global import to be useful:

```
from scapy.all import *
```

This imports all exported symbols from the scapy.all submodule into the global namespace. The rest of our examples assume your program has done this.

Reading and Writing Packet Capture Files

Your life will be easiest if all of your captures are in pure pcap format, not a format like pcap-ng. Wireshark, tshark, and dumpcap will *generally* produce pcap, unless you capture on all interfaces, in which case you will get pcap-ng files. That is, unless you override the default behavior. For dumpcap, which is our recommended way to capture packets (when feasible), the `-P` option will force normal pcap output.

Having said all that:

```
frames = rdpcap('file.cap')
```

This opens a pcap file named `file.cap` for reading, and returns an iterable, which we've called `frames`, because it potentially contains layer-2 frames, rather than layer-3 packets. That will depend on the capture file, however.

We can iterate over these as follows:

```
for f in frames:
    pass
```

This loop does nothing (`pass` is a nop in python).

To write packets to a file, we would call:

```
wrpcap('outfile.pcap', pkts)
```

`pkts` may be packets or frames, and should be an iterable, such as a list.

Dissecting Packets and Frames

Scapy stores everything in dict-like objects, which is handy. The objects are actually built as a series of *layers*. Consider:

```
for f in frames:
    if IP not in f:
        continue
    pkt = f[IP]
```

First, we verify that there's an IP layer in this object, and if not we skip to the next one. Then we get the IP layer of `f`, which may be identically `f` or it may be a layer (at any depth).

We can also do more complex things, skipping over layers we don't care about:

```

for f in frames:
    if DNS not in f:
        continue
    d = f[DNS]

```

This might be a DNS layer within a UDP layer within an IP layer within an Ether layer. The nice thing is that we don't have to care.

At a given layer, there are a number of *fields* that we can access:

```

for f in frames:
    if DNS not in f:
        continue
    d = f[DNS]
    d.opcode

```

The easiest way to get a feel for what's in a scapy object is to call the `display` method:

```

for f in frames:
    f.display()
    if DNS in f:
        f[DNS].display()

```

The first call will print (to stdout) all of the layers, including their fields, while the second will only print information about the DNS layer.

Creating Scapy Objects

Scapy has fairly normal constructors:

```
pkt = IP(dst='1.2.3.4')
```

It also has a layering operator:

```

pkt = IP(dst='1.2.3.4')
udp = UDP(dport=123)
p = pkt/udp
pkt.display()
udp.display()
p.display()

```

We can simplify this:

```

pkt = IP(dst='1.2.3.4')
udp = UDP(dport=123)
pkt /= udp
pkt.display()

```

We can even simplify it further:

```

pkt = IP(dst='1.2.3.4')/UDP(dport=123)
pkt.display()

```

Here are some layers that might interest you:

- Ether
- IP
- ICMP
- UDP
- TCP
- DNS

- DNSQR
- DNSRR
- Raw

What's this Raw layer? It's literally a raw sequence of bytes:

```
pkt = IP(dst='1.2.3.4')/UDP(dport=123)/Raw('This is a test')
pkt.display()
```

This just wraps the bytes in an appropriate scapy layer object, and we can shorten this:

```
pkt = IP(dst='1.2.3.4')/UDP(dport=123)/'This is a test'
pkt.display()
```

We can also nest things further:

```
pkt = IP(dst='1.2.3.4')/UDP(dport=123)/IP(
    src='2.3.4.5',dst='3.4.5.6')/ICMP()/ 'This is a test'
pkt.display()
```

Sending and Receiving Packets

Finally, how do we connect to the actual network, rather than just working with files?

If you're creating packets at layer 3 (that is, starting from the IP layer), you can just call:

```
send(pkt)
```

You can send and then wait for a response, as well:

```
new_pkt = sr1(pkt)
```

Here, `new_pkt` is the response received.

To just receive packets from an interface:

```
def my_callback(pkt):
    pass
```

```
sniff(iface=None, count=0, prn=my_callback)
```

Specifying `None` for `iface` (the default) captures on all interfaces. A count of 0 (the default) captures forever; nonzero will stop after that number of packets have been received. See `help(sniff)` for more parameters, including filters.