Scapy p.05 – Sending our First Packet; ARP Response

29 Oct 2013 » Scapy (/category/Scapy)

With a good understanding of how to view our packets we can now move onto some packet generation. Let's talk a bit about sniffing first and how existing packets are our best tool for creating new ones.

Sniff() function arguments

We've used the `sniff()` function a couple times already to capture some packets for viewing. I'm going to explain a little bit more about the `sniff()` function and its arguments. The arguments we will be talking about are:

- **count:** Number of packets to capture. 0 means infinity.
- **iface:** Sniff for packets only on the provided interface.
- **prn:** Function to apply to each packet. If something is returned, it is displayed. For instance you can use prn = lambda x: x.summary().
- **store:** Whether to store sniffed packets or discard them. When you only want to monitor your network forever, set store to 0.
- **timeout:** Stop sniffing after a given time (default: None).

These should all be self-explanatory except for the `filter` and `prn` arguments. The `filter` argument takes BPF syntax filters (http://biot.com/capstats/bpf.html), just like Wireshark or tcpdump capture filters. The `prn` argument is a very cool capability of the `sniff()` function and you can read more about it here: Scapy and custom actions. (http://thepacketgeek.com/scapy-sniffing-with-custom-actions-part-1/)

Since we want to generate our first ARP packet we should go ahead and sniff one to see what it takes to recreate one using the `.show()` and `.command()` method. Here's a sniff using the `count` and `filter` arguments:

```
>>> pkts = sniff(count=5, filter="arp")
>>> pkts.summary()
Ether / ARP who has 172.16.20.255 says 172.16.20.40 / Padding
Ether / ARP who has 172.16.20.244 says 172.16.20.40 / Padding
Ether / ARP who has 172.16.20.252 says 172.16.20.40 / Padding
Ether / ARP who has 172.16.20.253 says 172.16.20.40 / Padding
Ether / ARP who has 172.16.20.80 says 172.16.20.74 / Padding
>>> pkts[0].show()
###[ Ethernet ]###
  dst= ff:ff:ff:ff:ff:ff
  src= 00:11:22:aa:bb:cc
  type= 0x806
###[ ARP ]###
     hwtype= 0x1
     ptype= 0x800
     hwlen= 6
     plen= 4
     op= who-has
     hwsrc= 00:11:22:aa:bb:cc
     psrc= 172.16.20.40
     hwdst= 00:00:00:00:00:00
     pdst= 172.16.20.255
###[ Padding ]###
>>> pkts[0].command()
"Ether(src='00:11:22:aa:bb:cc', dst='ff:ff:ff:ff:ff:ff', type=205
4)/ARP(hwdst='00:00:00:00:00:00', ptype=2048, hwtype=1, psrc='17
2.16.20.40', hwlen=6, plen=4, pdst='172.16.20.255', hwsrc='00:11:
22:aa:bb:cc', op=2)"
```

Building a Packet

It looks like ARP packets only have 2 layers plus padding that we have to worry about. We can use the `ls()` function on the Ether and ARP layers to see what options are available to us:

```
>>> ls(Ether)
dst        : DestMACField         = (None)
src        : SourceMACField       = (None)
type       : XShortEnumField      = (0)
>>> ls(ARP)
hwtype     : XShortField          = (1)
ptype      : XShortEnumField      = (2048)
hwlen      : ByteField            = (6)
plen       : ByteField            = (4)
op         : ShortEnumField       = (1)
hwsrc      : ARPSourceMACField    = (None)
psrc       : SourceIPField        = (None)
hwdst      : MACField             = ('00:00:00:00:00:00')
pdst       : IPField              = ('0.0.0.0')
```

Let's create our ARP packet and start assigning some values. We construct a new ARP packet, and use the assignment operator customize specific fields of our packet:

```
>>> arppkt = Ether()/ARP()
>>> arppkt[ARP].hwsrc = "00:11:22:aa:bb:cc"
>>> arppkt[ARP].pdst = "172.16.20.1"
>>> arppkt[Ether].dst = "ff:ff:ff:ff:ff:ff"
>>> arppkt
<Ether  dst=ff:ff:ff:ff:ff:ff type=0x806 |<ARP  hwsrc=00:11:22:a
a:bb:cc pdst=172.16.20.1 |>>
```

The layers we want are defined with the with the `Layer()` notation. This will work for any layer in the `ls()` command output. That's a lot of options! You can also define the packet from scratch with all the options in one statement by passing in the fields as arguments to the related layer.

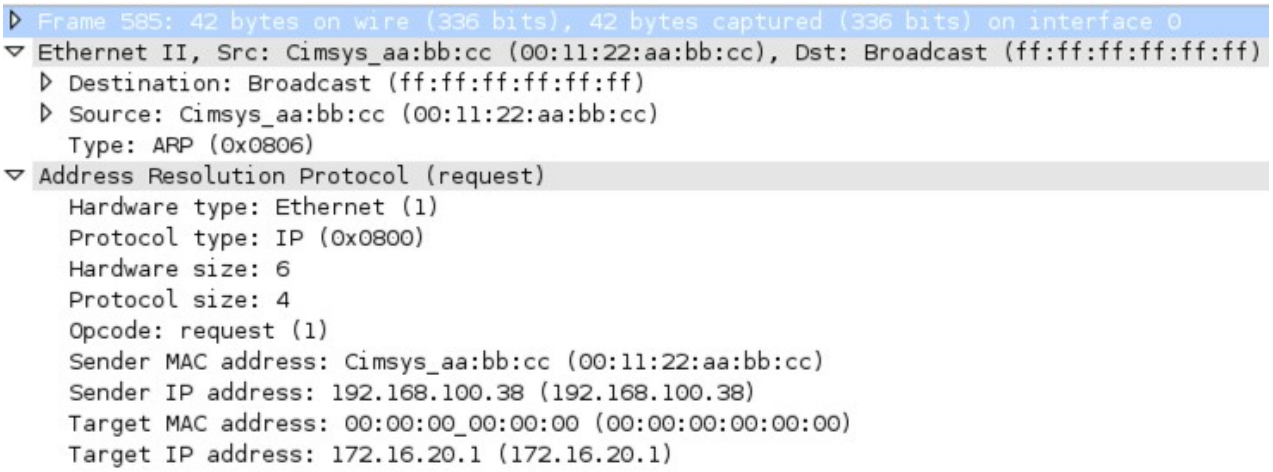
Note that the special glue holding these packets together is the `/` operator. If you happen to forget a layer when you're first defining the packet, you can add on a layer very easily using the existing packet and the `/` operator like this:

```
>>> tcppkt = Ether()/IP()
>>> tcppkt
<Ether  type=0x800 |<IP  |>>
>>> tcppkt = tcppkt/TCP()
>>> tcppkt
<Ether  type=0x800 |<IP  frag=0 proto=tcp |<TCP  |>>>
```

Sending a packet

Yup, you guessed it, its finally time to send this ARP packet out on the wire! Since ARP is a L2 protocol we're going to use the `sendp()` function as the `send()` function only works with L3 Packets (IP or IPv6 headers):

```
>>> arppkt
<Ether  dst=ff:ff:ff:ff:ff:ff src=00:11:22:aa:bb:cc type=0x806 |<
ARP  hwsrc=00:11:22:aa:bb:cc pdst=172.16.20.1 |>>
>>> sendp(arppkt)
.
Sent 1 packets.
```



Screenshot of capture packet in Wireshark

What, what! Check that out! Our packet out from the scapy console and in the wire! Pretty cool, right? Well, here's a fun fact. We don't need to create and build the packet before sending it, we can define the packet right there in the `send()` or `sendp()` function like this:

```
>>> sendp(Ether(dst="ff:ff:ff:ff:ff:ff",src="00:11:22:aa:bb:cc")/
ARP(hwsrc="00:11:22:aa:bb:cc",pdst="172.16.20.1"))
.
Sent 1 packets.
```

In fact, we can do some other cool things with these send functions. If we had an array of packets (such as one created with Python loops and some random or incrementing values for IP address/TCP port), the send function would send each packet in that array:

```
>>> pkts
<Sniffed: TCP:0 UDP:5 ICMP:4 Other:1>
>>> send(pkts)
..........
Sent 10 packets.
```

The send commands have some arguments to control the packet sending, here's the main ones you might consider using:

send(pkts, inter=0, loop=0)
sendp(pkts, inter=0, loop=0)

- **iface:** The interface to send the packets out from.
- **inter:** Time in seconds to wait between 2 packets.
- **loop:** Send the packets endlessly if not 0.
- **pkts:** Can be a packet, an implicit packet or a list of them.

Related Posts

- Writing Packets to Trace File with Scapy (https://thepacketgeek.com/writing-packets-to-trace-file-with-scapy/)
- Importing packets from trace files with scapy (https://thepacketgeek.com/importing-packets-from-trace-files/)
- Scapy p.01 – Scapy Introduction and Overview (https://thepacketgeek.com/scapy-p-01-scapy-introduction-and-overview/)
- Scapy p.02 – Installing Python and Scapy (https://thepacketgeek.com/scapy-p-02-installing-python-and-scapy/)
- Scapy p.03 – Scapy Interactive Mode (https://thepacketgeek.com/scapy-p-03-scapy-interactive-mode/)
- Scapy p.04 – Looking at Packets (https://thepacketgeek.com/scapy-p-04-looking-at-packets/)

« Scapy p.06 – Sending and Receiving with Scapy (/scapy-p-06-sending-and-receiving-with-scapy/)

Scapy p.04 – Looking at Packets » (/scapy-p-04-looking-at-packets/)

Scapy p.06 – Sending and Receiving with Scapy

29 Oct 2013 » Scapy (/category/Scapy)

We've sniffed some packets, dig down into packet layers and fields, and even sent some packets. Great job! It's time to step up our game with Scapy and start really using some of the power Scapy contains. Please Note: this next example is for education and example only. Please be responsible on your network, especially at work!

Scapy Send/Receive Function

Let's get familiar with the `sr()`, `sr1()`, `srp()`, and `srp1()` functions. Just like the `send()`, function, the `p` at the end of the function name means that we're sending at L2 instead of L3. The functions with a `1` in them mean that Scapy will send the specified packet and end after receiving 1 answer/response instead of continuing to listen for answers/responses. I'll reference both functions as `sr()`, but the examples will use the correct function.

Sending an ICMP Echo Request (ping)

The `sr()` function is used to send a packet or group of packets when you expect a response back. We'll be sending an ICMP Echo Request (ping) since we can expect some sort of a response back from that. First let's use the `sniff()` function to figure out what an ICMP Echo Request looks like in Scapy:

```
> p = sniff(count=10,filter="icmp and ip host 4.2.2.1")
> p
<Sniffed: TCP:0 UDP:0 ICMP:10 Other:0>
> p[0]
<Ether  dst=00:07:7d:6d:b4:9e src=b8:f6:b1:11:65:35 type=0x800 |<IP  version=4L ihl=5L to
s=0x0 len=84 id=14488 flags= frag=0L ttl=64 proto=icmp chksum=0x7bd6 src=172.16.20.40 dst
=4.2.2.1 options=[] |<ICMP  type=echo-request code=0 chksum=0xaba6 id=0x55d3 seq=0x0 |<Ra
w |>>
```

In the previous ARP example we changed the dst and src MAC address, but since we're expecting a response back from another network device we'll have to leave it up to Scapy to fill those in when it sends the packets. Since we're building a L3 packet, we can actually leave off the Ether layer since Scapy will handle the generation of that. So let's start building the `IP` and `ICMP` layers. To see the available fields for each layer, and what the default values will be if we don't specify, use the `ls('layer')` command:

```
>>> 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    : PacketListField      = ([])
>>> ls(ICMP)
type       : ByteEnumField        = (8)
code       : MultiEnumField       = (0)
chksum     : XShortField          = (None)
id         : ConditionalField     = (0)
seq        : ConditionalField     = (0)
ts_ori     : ConditionalField     = (13940582)
ts_rx      : ConditionalField     = (13940582)
ts_tx      : ConditionalField     = (13940582)
gw         : ConditionalField     = ('0.0.0.0')
ptr        : ConditionalField     = (0)
reserved   : ConditionalField     = (0)
addr_mask  : ConditionalField     = ('0.0.0.0')
unused     : ConditionalField     = (0)
```

Most of those default values are fine, and src addresses will be filled out automatically by Scapy when it sends the packets. We can spoof those if desired, but again, since we're expecting a response we need to leave those alone. We'll be sending a L3 packet and we only expect one response, so we'll build and send our ICMP packet using the `sr1()` function:

```
>>> pingr = IP(dst="192.168.200.254")/ICMP()
>>> sr1(pingr)
Begin emission:
..Finished to send 1 packets.
.*
Received 84 packets, got 1 answers, remaining 0 packets
<IP  version=4L ihl=5L tos=0x0 len=28 id=1 flags= frag=0L ttl=255 proto=icmp chksum=0xa7c
4 src=4.2.2.1 dst=172.16.20.40 options=[] |<ICMP  type=echo-reply code=0 chksum=0xffff id
=0x0 seq=0x0 |<Padding |>>>
```

Scapy prints out the response packet

The `Received 84 packets` is referring to the number of non-response packets Scapy sniffed while waiting for the response. It's not anything to be alarmed about, but just note that on a busy host you might see a big number of packets there. We can also define the ICMP packet directly in the `sr1()` function like this:

```
>>> sr1(IP(dst="192.168.200.254")/ICMP())
Begin emission:
..Finished to send 1 packets.
.*
Received 97 packets, got 1 answers, remaining 0 packets
<IP  version=4L ihl=5L tos=0x0 len=28 id=1 flags= frag=0L ttl=255 proto=icmp chksum=0xa7c
4 src=4.2.2.1 dst=172.16.20.40 options=[] |<ICMP  type=echo-reply code=0 chksum=0xffff id
=0x0 seq=0x0 |<Padding |>>>
```

We can save the response packet into a variable just like we do when creating a packet:

```
>>> resp = sr1(pingr)
Begin emission:
..Finished to send 1 packets.
.*
Received 147 packets, got 1 answers, remaining 0 packets
>>> resp[0].summary()
'IP / ICMP 4.2.2.1 > 172.16.20.40 echo-reply 0 / Padding'
```

If we're saving the response, Scapy won't print it out by default

Two other Scapy functions related to sending and receiving packets are the `srloop()` and `srploop()`. The `srloop()` will send the L3 packet and continue to resend the packet after each response is received. The `srploop()` does the same thing except for… you guessed it, L2 packets! This let's us simulate the `ping` command, and with the `count` argument, we can also define the number of times to loop:

```
>>> resp = srloop(pingr, count=5)
RECV 1: IP / ICMP 4.2.2.1 > 172.16.20.10 echo-reply 0 / Padding
RECV 1: IP / ICMP 4.2.2.1 > 172.16.20.10 echo-reply 0 / Padding
RECV 1: IP / ICMP 4.2.2.1 > 172.16.20.10 echo-reply 0 / Padding
RECV 1: IP / ICMP 4.2.2.1 > 172.16.20.10 echo-reply 0 / Padding
RECV 1: IP / ICMP 4.2.2.1 > 172.16.20.10 echo-reply 0 / Padding

Sent 5 packets, received 5 packets. 100.0% hits.
>>> resp
(<Results: TCP:0 UDP:0 ICMP:5 Other:0>, <PacketList: TCP:0 UDP:0 ICMP:0 Other:0>)
```

Saving the responses puts our packets into an array

As you can see, our Scapy skills are building and you might already have some ideas about how you can use these functions in your own network tools. In the next article, we'll see how you can build an ARP monitor to keep an ear on the network for possible spoofed ARP replies.

Related Posts

- Writing Packets to Trace File with Scapy (https://thepacketgeek.com/writing-packets-to-trace-file-with-scapy/)
- Importing packets from trace files with scapy (https://thepacketgeek.com/importing-packets-from-trace-files/)
- Scapy p.01 – Scapy Introduction and Overview (https://thepacketgeek.com/scapy-p-01-scapy-introduction-and-overview/)
- Scapy p.02 – Installing Python and Scapy (https://thepacketgeek.com/scapy-p-02-installing-python-and-scapy/)
- Scapy p.03 – Scapy Interactive Mode (https://thepacketgeek.com/scapy-p-03-scapy-interactive-mode/)
- Scapy p.04 – Looking at Packets (https://thepacketgeek.com/scapy-p-04-looking-at-packets/)

« Scapy p.07 – Monitoring ARP (/scapy-p-07-monitoring-arp/)

Scapy p.05 – Sending our First Packet; ARP Response » (/scapy-p-05-sending-our-first-packet-arp-response/)

Scapy p.04 – Looking at Packets

29 Oct 2013 » Scapy (/category/Scapy)

Packets, Layers, and Fields. Oh My!

Scapy uses Python dictionaries as the data structure for packets. Each packet is a collection of nested dictionaries with each layer being a child dictionary of the previous layer, built from the lowest layer up. Visualizing the nested packet layers would look something like this:

Each field (such as the Ethernet `dst' value or ICMP` type' value) is a key:value pair in the appropriate layer. These fields (and nested layers) are all mutable so we can reassign them in place using the assignment operator. Scapy has packet methods for viewing the layers and fields that I will introduce next.

Packet summary() and show() Methods

Now let's go back to our `pkt` and have some fun with it using Scapy's Interactive mode. We already know that using the `summary()` method will give us a quick look at the packet's layers:

```
>>> pkt[0].summary()
'Ether / IP / ICMP 172.16.20.10 > 4.2.2.1 echo-request 0 / Raw'
```

But what if we want to see more of the packet contents? That's what the

`show()` method is for:

```
>>> pkt[0].show()
###[ Ethernet ]###
  dst= 00:24:97:2e:d6:c0
  src= 00:00:16:aa:bb:cc
  type= 0x800
###[ IP ]###
     version= 4L
     ihl= 5L
     tos= 0x0
     len= 84
     id= 57299
     flags=
     frag= 0L
     ttl= 64
     proto= icmp
     chksum= 0x0
     src= 172.16.20.10
     dst= 4.2.2.1
     \options\
###[ ICMP ]###
        type= echo-request
        code= 0
        chksum= 0xd8af
        id= 0x9057
        seq= 0x0
###[ Raw ]###
```

Very cool, that's some good info. If you're familiar with Python you have probably noticed the list index, `[0]`, after the `pkt` variable name. Remember that our sniff only returned a single packet, but if we increase the `count` argument value, we will get back an list with multiple packets:

```
>>> pkts = sniff(count=10)
>>> pkts
<Sniffed: TCP:0 UDP:0 ICMP:10 Other:0>
```

Getting the value of the list returns a quick glance at what type of packets were sniffed.

```
>>> pkts.summary()
Ether / IP / ICMP 172.16.20.10 > 4.2.2.1 echo-request 0 / Raw
Ether / IP / ICMP 4.2.2.1 > 172.16.20.10 echo-reply 0 / Raw
Ether / IP / ICMP 172.16.20.10 > 4.2.2.1 echo-request 0 / Raw
Ether / IP / ICMP 4.2.2.1 > 172.16.20.10 echo-reply 0 / Raw
Ether / IP / ICMP 172.16.20.10 > 4.2.2.1 echo-request 0 / Raw
Ether / IP / ICMP 4.2.2.1 > 172.16.20.10 echo-reply 0 / Raw
Ether / IP / ICMP 172.16.20.10 > 4.2.2.1 echo-request 0 / Raw
Ether / IP / ICMP 4.2.2.1 > 172.16.20.10 echo-reply 0 / Raw
Ether / IP / ICMP 172.16.20.10 > 4.2.2.1 echo-request 0 / Raw
Ether / IP / ICMP 4.2.2.1 > 172.16.20.10 echo-reply 0 / Raw
```

And we can show the summary or packet contents of any single packet by using the list index with that packet value. So, let's look at the contents of the 4th packet (Remember, list indexes start counting at 0):

```
>>> pkts[3]
<Ether  dst=00:00:16:aa:bb:cc src=00:24:97:2e:d6:c0 type=0x800 |<IP  version=4L ihl=5L to
s=0x20 len=84 id=47340 flags= frag=0L ttl=57 proto=icmp chksum=0x3826 src=4.2.2.1 dst=17
2.16.20.10 options=[] |<ICMP  type=echo-reply code=0 chksum=0xcfbf id=0x3060 seq=0x1 |<Ra
w |>>>>
```

Getting the value of a single packet returns a quick glance of the contents of that packet.

The `show()` method will give us a cleaner print out:

```
>>> pkts[3].show()
###[ Ethernet ]###
  dst= 00:00:16:aa:bb:cc
  src= 00:24:97:2e:d6:c0
  type= 0x800
###[ IP ]###
     version= 4L
     ihl= 5L
     tos= 0x20
     len= 84
     id= 47340
     flags=
     frag= 0L
     ttl= 57
     proto= icmp
     chksum= 0x3826
     src= 4.2.2.1
     dst= 172.16.20.10
     \options\
###[ ICMP ]###
        type= echo-reply
        code= 0
        chksum= 0xcfbf
        id= 0x3060
        seq= 0x1
###[ Raw ]###
```

Digging into Packets by Layer

Scapy builds and dissects packets by the layers contained in each packet, and then by the fields in each layer. Each layer is nested inside the parent layer as can be seen with the nesting of the < and > brackets:

```
>>> pkts[4]
<Ether  dst=00:24:97:2e:d6:c0 src=00:00:16:aa:bb:cc type=0x800 |<IP  version=4L ihl=5L to
s=0x0 len=84 id=17811 flags= frag=0L ttl=64 proto=icmp chksum=0x0 src=192.168.201.203 dst
=4.2.2.1 options=[] |<ICMP  type=echo-request code=0 chksum=0xc378 id=0x3060 seq=0x2 |<Ra
w |>>>>
```

You can also dig into a specific layer using an list index. If we wanted to get to the `ICMP` layer of `pkts[3]`, we could do that using the layer name or index number:

```
>>> pkts[3][ICMP].summary()
'ICMP 4.2.2.1 > 192.168.201.203 echo-reply 0 / Raw'
>>> pkts[3][2].summary()
'ICMP 4.2.2.1 > 192.168.201.203 echo-reply 0 / Raw'
```

Since the first index chooses the packet out of the `pkts` list, the second index chooses the layer for that specific packet. Looking at the summary of this packet from an earlier example, we know that the `ICMP` layer is the 3rd layer.

Packet .command() Method

If you're wanting to see a reference of how a packet that's been sniffed or received might look to create, Scapy has a packet method for you! Using the `.command()` packet method will return a string of the command necessary to recreate that packet, like this:

```
>>> pkts[2].command()
'Ether(src=\'00:11:22:aa:bb:cc\', dst=\'c0:c1:c0:b7:ce:63\', type=2048)/IP(frag=0L, src=
\'172.16.20.10\', proto=1, tos=0, dst=\'4.2.2.1\', chksum=51457, len=84, options=[], vers
ion=4L, flags=0L, ihl=5L, ttl=64, id=59755)/ICMP(gw=None, code=0, ts_ori=None, addr_mask=
None, seq=3, ptr=None, unused=None, ts_rx=None, chksum=50424, reserved=None, ts_tx=None,
type=8, id=59999)/Raw(load=\'Rk\\xe8\\x02\\x00\\x0c#\\\'\\x08\\t\\n\\x0b\\x0c\\r\\x0e\\x0
f\\x10\\x11\\x12\\x13\\x14\\x15\\x16\\x17\\x18\\x19\\x1a\\x1b\\x1c\\x1d\\x1e\\x1f !"#$%&
\\\'()*+,-./01234567\')'
```

Digging into Layers by Field

Within each layer, Scapy parses out the value of each field if it has support for the layer's protocol. Depending on the type of field, Scapy may replace the value with a more friendly text value for the summary views, but not in the values returned for an individual field. Here are some examples:

```
>>> pkts[3]
<Ether  dst=00:00:16:aa:bb:cc src=00:24:97:2e:d6:c0 type=0x800 |\
<IP  version=4L ihl=5L tos=0x20 len=84 id=47340 flags= frag=0L ttl=57 proto=icmp chksum=0
x3826 src=4.2.2.1 dst=192.168.201.203 options=[] |\
<ICMP  type=echo-reply code=0 chksum=0xcfbf id=0x3060 seq=0x1 |<Raw |>>>>
>>> pkts[3][Ether].src
'00:24:97:2e:d6:c0'
>>> pkts[3][IP].ttl
57
>>> pkts[3][IP].proto
1
>>> pkts[3][ICMP].type
0
```

Using Python control statements with Scapy

The awesome thing about Scapy being a module of Python is that we can use the power of Python to do stuff with our packets. Here's a tip of the iceberg example using a Python `for` statement along with some new Scapy packet methods:

```
>>> for packet in pkts:
...     if (packet.haslayer(ICMP)):
...         print(f"ICMP code: {packet.getlayer(ICMP).code}")
...
ICMP code: 0
ICMP code: 0
ICMP code: 0
ICMP code: 0
ICMP code: 0
ICMP code: 0
ICMP code: 0
ICMP code: 0
ICMP code: 0
ICMP code: 0
```

As you can guess, the `haslayer()` and `getlayer()` methods will test for the existence of a layer and return the layer (and any nested layers) respectively. This is just a very basic use of Python statements with Scapy and we'll see a lot more when it comes to packet generation and custom actions.

Related Posts

- Writing Packets to Trace File with Scapy (https://thepacketgeek.com/writing-packets-to-trace-file-with-scapy/)

- Importing packets from trace files with scapy (https://thepacketgeek.com/importing-packets-from-trace-files/)
- Scapy p.01 – Scapy Introduction and Overview (https://thepacketgeek.com/scapy-p-01-scapy-introduction-and-overview/)
- Scapy p.02 – Installing Python and Scapy (https://thepacketgeek.com/scapy-p-02-installing-python-and-scapy/)
- Scapy p.03 – Scapy Interactive Mode (https://thepacketgeek.com/scapy-p-03-scapy-interactive-mode/)
- Scapy p.05 – Sending our First Packet; ARP Response (https://thepacketgeek.com/scapy-p-05-sending-our-first-packet-arp-response/)

« Scapy p.05 – Sending our First Packet; ARP Response (/scapy-p-05-sending-our-first-packet-arp-response/)

Scapy p.03 – Scapy Interactive Mode » (/scapy-p-03-scapy-interactive-mode/)