About:

- Scapy is a network packet manipulation and creation tool
- It allows you to send, sniff, dissect, and forge network packets
- This capability allows a user to probe, scan, or attack networks
- Primary usage and dependency is Python
    - This means that the previous two bullet points can be done automatically in a script
- When running Scapy, make sure to run as an administrator to have access to some functionalities like sending packets otherwise you will be blocked

Building a Simple Packet:

- Simple as stating `>>> packetA=IP(ttl=10)` defines a packet **packetA** that has a Time-To-Live of 10 hops ready to be sent on the IP layer.
- This packet can be built upon now that it has been defined
- For example, you can add a source flag and destination flag to it:

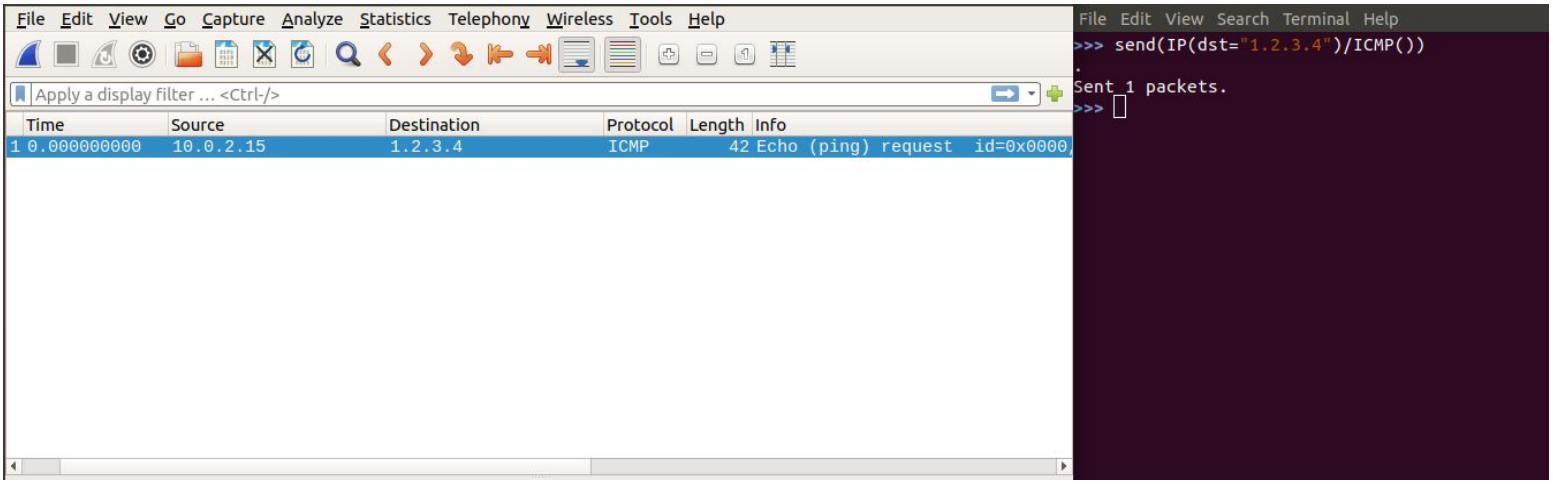
```
>>> packetA.src="127.0.0.1"
>>> packetA.dst="1.2.3.4"
```

Now when **packetA** is called in Scapy, this is the resulting output given:

```
>>> <IP  ttl='10' src=127.0.0.1 dst=5.6.7.8 |>
```

Sending a Packet:

- There are two functions available that allow you to send packets: **send()** and **sendp()**
- **send()** allows you to send packets on Layer 3, Network Layer, (while handling routing and Layer 2 for you), while **sendp()** sends packets on Layer 2, Data-Link Layer.
- For example: `>>> send(IP(dst="1.2.3.4")/ICMP())` sends an ICMP ping over the network to the IP 1.2.3.4.
    - The **/** in the above code segment refers to a composition of two layers in which the lower layer (IP) can have one or more of its components overloaded by the upper layer (ICMP)
- Using Wireshark, we are able to capture the ping on the next page

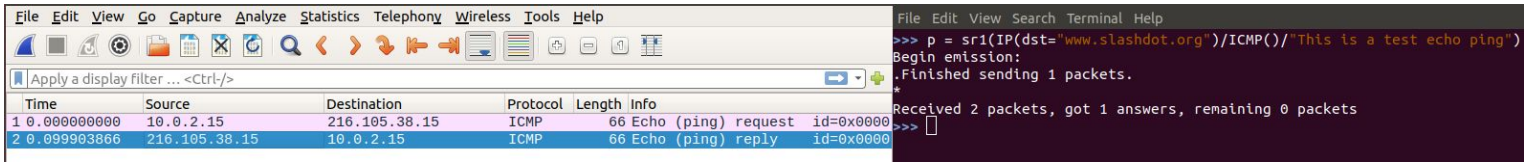


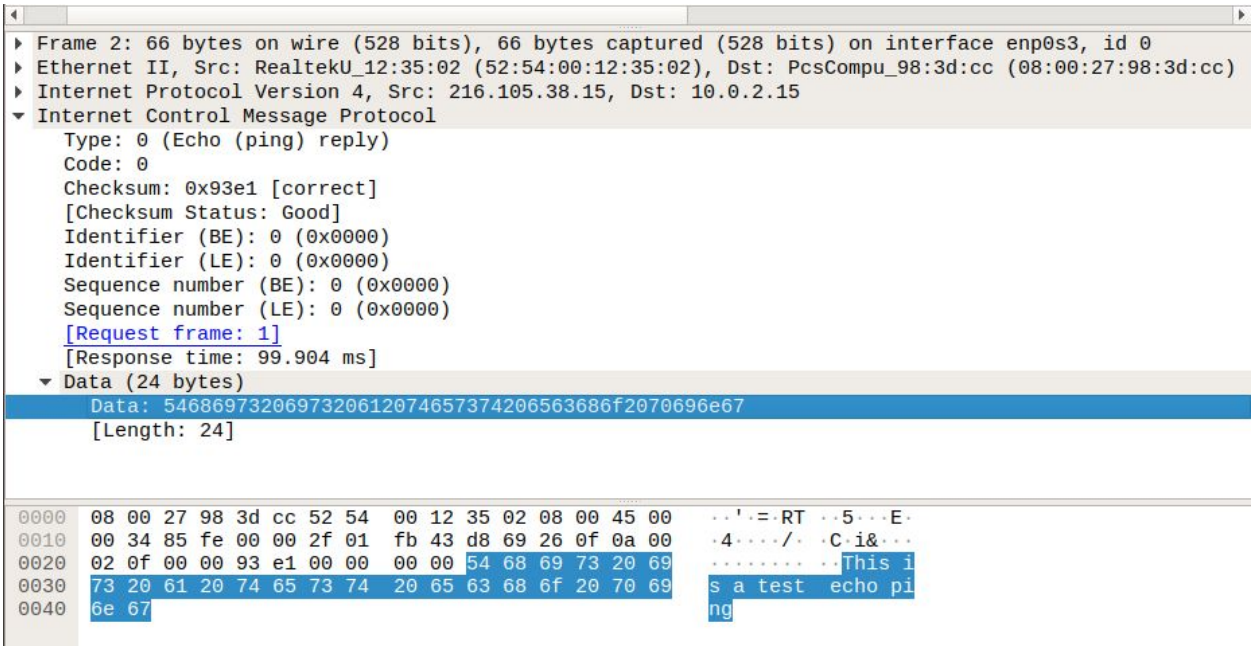
Sending and Receiving Packets:

- There are a couple functions that can be used for sending and receiving packets. **sr()**, **sr1()**, and **srp()**
- **sr()** sends a couple packets and receives the answer packets back along with unanswered packets
- **sr1()** is a variant of **sr()** that only sends one packet and gets back one answer packet. They are also both Layer 3 functions
- **srp()** is another variant of **sr()** that sends packets over Layer 2.
- For this example we will be using **sr1()** to send a ping over to **slashdot.org**:

```
>>> p = sr1(IP(dst="www.slashdot.org")/ICMP()/"This is a test echo ping")
```

- We are able to capture the request and reply along with the data in Wireshark below:





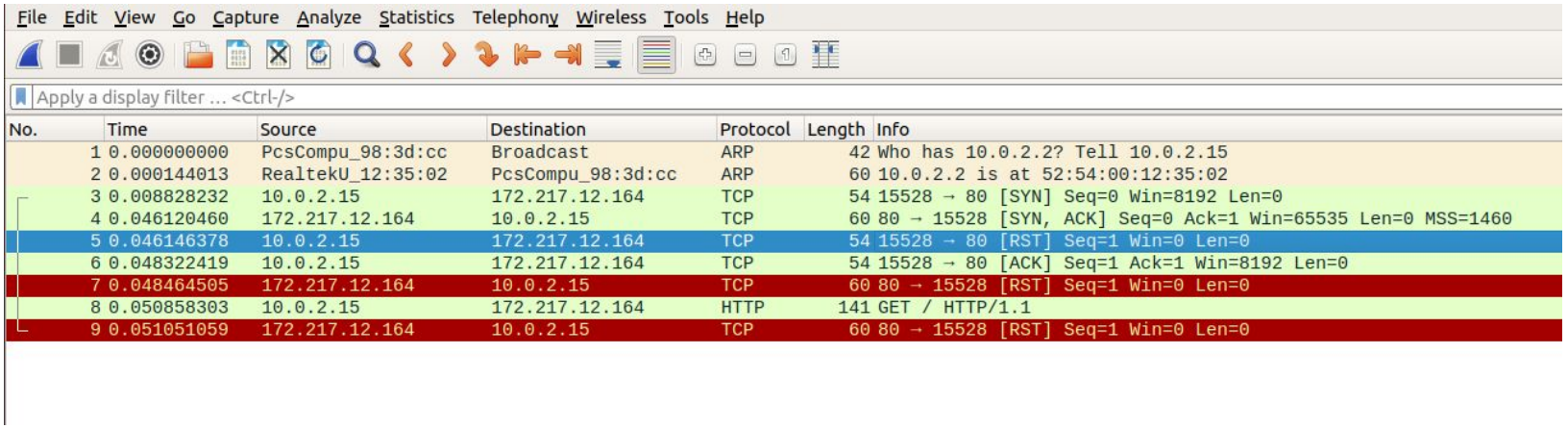
Establishing a HTTP Connection:

- One small example we will go over is the automating the process of establishing a HTTP connection with Scapy
- There are two ways we can do this
    - Using HTTP()/HTTPRequest()
    - With the predefined function: http_request()
- We first load the http layer with `load_layer("http")`
- Then we use the following setup:

```
load_layer("http")

req = HTTP()/HTTPRequest(
    Accept_Encoding=b'gzip',
    Connection=b"keep-alive",
    Host=b'www.google.com'
    )

pkt = TCP_client.tcplink(HTTP, "www.google.com", 80)
ans = pkt.sr1(req)
pkt.close()
```

- This breakdown should look familiar from Computer Networks
- What this does is sends an HTTP SYN packet to www.google.com
- Once the Three-Way Handshake is established, the connection closes

- This is shown below in the captured Wireshark packets below



- The other method of sending the same packet is by using the following:

```
load_layer("http")
http_request("www.google.com", "/", display=True)
```

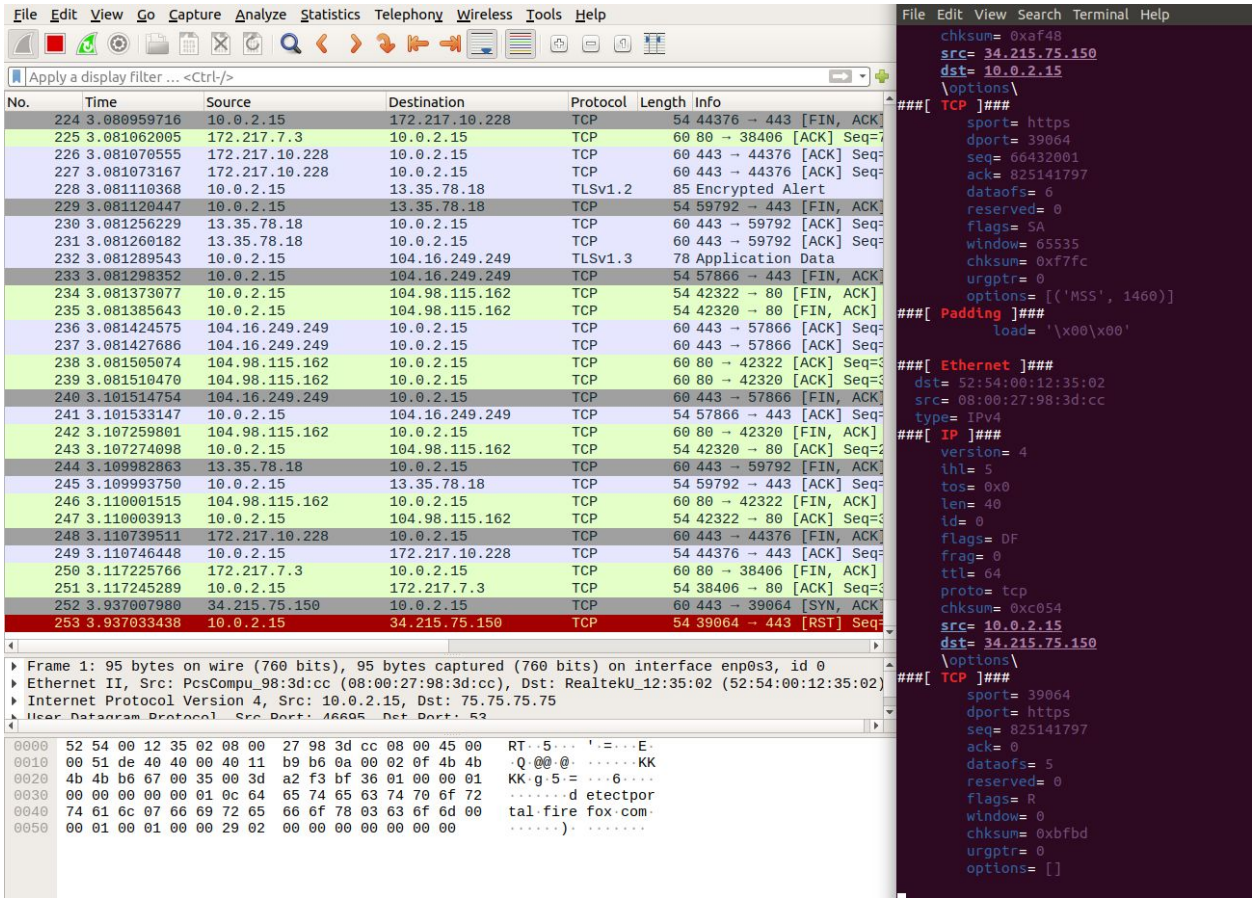
- The use of http_request() follows the same formatting as above and gets the same response, just shorter and more simple
- This method is quicker if you are not looking to set up your HTTP packet a specific way

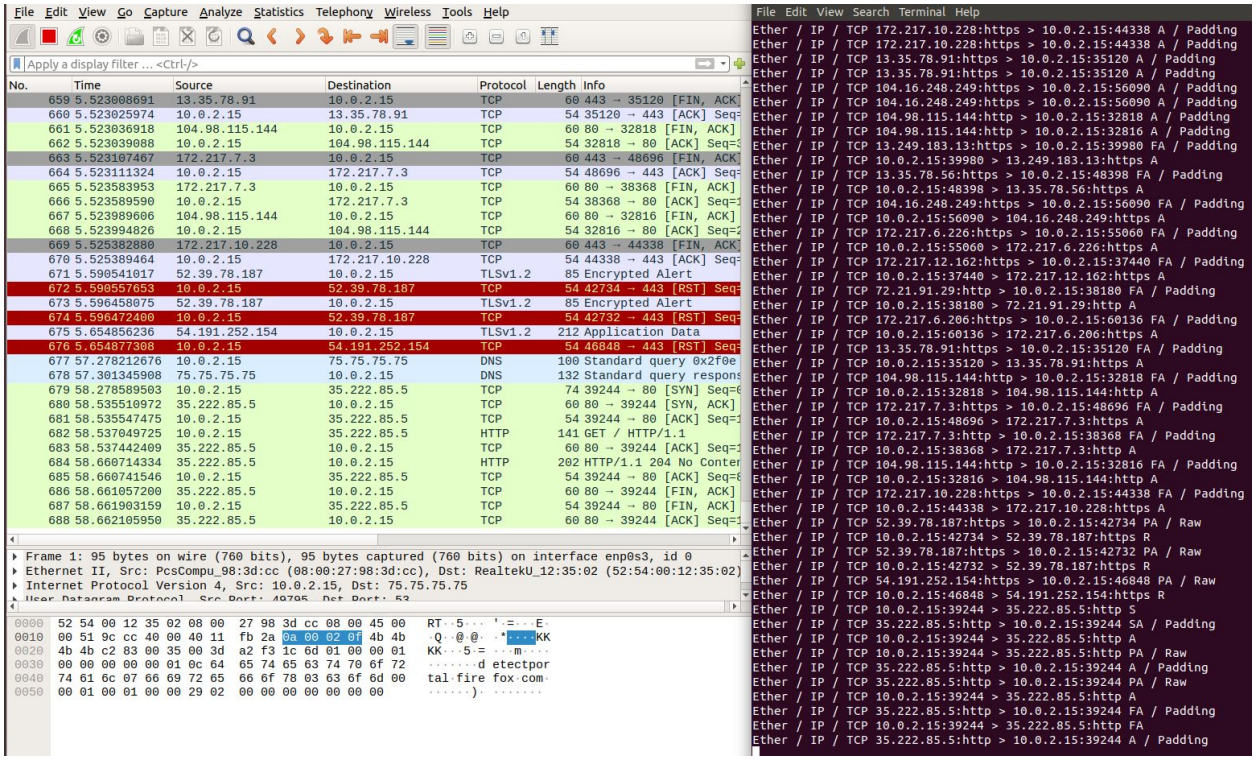
Sniffing:

- Captures packets being sent over the network to be analyzed (using **tcpdump** in this case with Ubuntu)
- Very useful as it provides a way to capture packets and analyze them
- There are a couple ways packets can be analyzed: **show()** and **summary()**
- Below are images of Wireshark and Scapy sniffing side-by-side to see the differences. The first is using **show()**, the other **summary()**.

```
sniff(iface="enp0s3", prn=lambda x: x.show())
sniff(iface="enp0s3", prn=lambda x: x.summary())
```

- **iface** refers to the network that is to be analyzed. This is not required but used here as an example.
- **prn** is an argument that applies a function to be executed with each packet discovered
    - In this case, a lambda function that uses the show/summary function based on the packet input **x**


File Edit View Go Capture Analyze Statistics Telephony Wireless Tools Help
Apply a display filter … <Ctrl-/>
No. Time Source Destination Protocol Length Info
224 3.080959716 10.0.2.15 172.217.10.228 TCP 54 44376 → 443 [FIN, ACK]
225 3.081062005 172.217.7.3 10.0.2.15 TCP 60 80 → 38406 [ACK] Seq=7
226 3.081070555 172.217.10.228 10.0.2.15 TCP 60 443 → 44376 [ACK] Seq=
227 3.081073167 172.217.10.228 10.0.2.15 TCP 60 443 → 44376 [ACK] Seq=
228 3.081110368 10.0.2.15 13.35.78.18 TLSv1.2 85 Encrypted Alert
229 3.081120447 10.0.2.15 13.35.78.18 TCP 54 59792 → 443 [FIN, ACK]
230 3.081256229 13.35.78.18 10.0.2.15 TCP 60 443 → 59792 [ACK] Seq=
231 3.081260182 13.35.78.18 10.0.2.15 TCP 60 443 → 59792 [ACK] Seq=
232 3.081289543 10.0.2.15 104.16.249.249 TLSv1.3 78 Application Data
233 3.081298352 10.0.2.15 104.16.249.249 TCP 54 57866 → 443 [FIN, ACK]
234 3.081373077 10.0.2.15 104.98.115.162 TCP 54 42322 → 80 [FIN, ACK]
235 3.081385643 10.0.2.15 104.98.115.162 TCP 54 42320 → 80 [FIN, ACK]
236 3.081424575 104.16.249.249 10.0.2.15 TCP 60 443 → 57866 [ACK] Seq=
237 3.081427686 104.16.249.249 10.0.2.15 TCP 60 443 → 57866 [ACK] Seq=
238 3.081505074 104.98.115.162 10.0.2.15 TCP 60 80 → 42322 [ACK] Seq=3
239 3.081510470 104.98.115.162 10.0.2.15 TCP 60 80 → 42320 [ACK] Seq=3
240 3.101514754 104.16.249.249 10.0.2.15 TCP 60 443 → 57866 [FIN, ACK]
241 3.101533147 10.0.2.15 104.16.249.249 TCP 54 57866 → 443 [ACK] Seq=
242 3.107259801 104.98.115.162 10.0.2.15 TCP 60 80 → 42320 [FIN, ACK]
243 3.107274098 10.0.2.15 104.98.115.162 TCP 54 42320 → 80 [ACK] Seq=2
244 3.109982863 13.35.78.18 10.0.2.15 TCP 60 443 → 59792 [FIN, ACK]
245 3.109993750 10.0.2.15 13.35.78.18 TCP 54 59792 → 443 [ACK] Seq=
246 3.110001515 104.98.115.162 10.0.2.15 TCP 60 80 → 42322 [FIN, ACK]
247 3.110003913 10.0.2.15 104.98.115.162 TCP 54 42322 → 80 [ACK] Seq=3
248 3.110739511 172.217.10.228 10.0.2.15 TCP 60 443 → 44376 [FIN, ACK]
249 3.110746448 10.0.2.15 172.217.10.228 TCP 54 44376 → 443 [ACK] Seq=
250 3.117225766 172.217.7.3 10.0.2.15 TCP 60 80 → 38406 [FIN, ACK]
251 3.117245289 10.0.2.15 172.217.7.3 TCP 54 38406 → 80 [ACK] Seq=3
252 3.937007980 34.215.75.150 10.0.2.15 TCP 60 443 → 39064 [SYN, ACK]
253 3.937033438 10.0.2.15 34.215.75.150 TCP 54 39064 → 443 [RST] Seq=
Frame 1: 95 bytes on wire (760 bits), 95 bytes captured (760 bits) on interface enp0s3, id 0
Ethernet II, Src: PcsCompu_98:3d:cc (08:00:27:98:3d:cc), Dst: RealtekU_12:35:02 (52:54:00:12:35:02)
Internet Protocol Version 4, Src: 10.0.2.15, Dst: 75.75.75.75
0000 52 54 00 12 35 02 08 00 27 98 3d cc 08 00 45 00
0010 00 51 de 40 40 00 40 11 b9 b6 0a 00 02 0f 4b 4b
0020 4b 4b b6 67 00 35 00 3d a2 f3 bf 36 01 00 00 01
0030 00 00 00 00 00 01 0c 64 65 74 65 63 74 70 6f 72
0040 74 61 6c 07 66 69 72 65 66 6f 78 03 63 6f 6d 00
0050 00 01 00 01 00 00 29 02 00 00 00 00 00 00 00
File Edit View Search Terminal Help
chksum= 0xaf48
src= 34.215.75.150
dst= 10.0.2.15
\options\
###[TCP]###
sport= https
dport= 39064
seq= 66432001
ack= 825141797
dataofs= 6
reserved= 0
flags= SA
window= 65535
chksum= 0xf7fc
urgptr= 0
options= [('MSS', 1460)]
###[Padding]###
load= '\x00\x00'
###[Ethernet]###
dst= 52:54:00:12:35:02
src= 08:00:27:98:3d:cc
type= IPv4
###[IP]###
version= 4
ihl= 5
tos= 0x0
len= 40
id= 0
flags= DF
frag= 0
ttl= 64
proto= tcp
chksum= 0xc054
src= 10.0.2.15
dst= 34.215.75.150
\options\
###[TCP]###
sport= 39064
dport= https
seq= 825141797
ack= 0
dataofs= 5
reserved= 0
flags= R
window= 0
chksum= 0xbfbd
urgptr= 0
options= []


File Edit View Go Capture Analyze Statistics Telephony Wireless Tools Help
Apply a display filter … <Ctrl-/>
No. Time Source Destination Protocol Length Info
659 5.523008691 13.35.78.91 10.0.2.15 TCP 60 443 → 35120 [FIN, ACK]
660 5.523025974 10.0.2.15 13.35.78.91 TCP 54 35120 → 443 [ACK] Seq=
661 5.523036918 104.98.115.144 10.0.2.15 TCP 60 80 → 32818 [FIN, ACK]
662 5.523039088 10.0.2.15 104.98.115.144 TCP 54 32818 → 80 [ACK] Seq=3
663 5.523107467 172.217.7.3 10.0.2.15 TCP 60 443 → 48696 [FIN, ACK]
664 5.523111324 10.0.2.15 172.217.7.3 TCP 54 48696 → 443 [ACK] Seq=
665 5.523583953 172.217.7.3 10.0.2.15 TCP 60 80 → 38368 [FIN, ACK]
666 5.523589590 10.0.2.15 172.217.7.3 TCP 54 38368 → 80 [ACK] Seq=1
667 5.523989606 104.98.115.144 10.0.2.15 TCP 60 80 → 32816 [FIN, ACK]
668 5.523994826 10.0.2.15 104.98.115.144 TCP 54 32816 → 80 [ACK] Seq=2
669 5.525382880 172.217.10.228 10.0.2.15 TCP 60 443 → 44338 [FIN, ACK]
670 5.525389464 10.0.2.15 172.217.10.228 TCP 54 44338 → 443 [ACK] Seq=
671 5.590541017 52.39.78.187 10.0.2.15 TLSv1.2 85 Encrypted Alert
672 5.590557653 10.0.2.15 52.39.78.187 TCP 54 42734 → 443 [RST] Seq=
673 5.596458075 52.39.78.187 10.0.2.15 TLSv1.2 85 Encrypted Alert
674 5.596472400 10.0.2.15 52.39.78.187 TCP 54 42732 → 443 [RST] Seq=
675 5.654856236 54.191.252.154 10.0.2.15 TLSv1.2 212 Application Data
676 5.654877308 10.0.2.15 54.191.252.154 TCP 54 46848 → 443 [RST] Seq=
677 57.278212676 10.0.2.15 75.75.75.75 DNS 100 Standard query 0x2f0e
678 57.301345908 75.75.75.75 10.0.2.15 DNS 132 Standard query respons
679 58.278589503 10.0.2.15 35.222.85.5 TCP 74 39244 → 80 [SYN] Seq=0
680 58.535510972 35.222.85.5 10.0.2.15 TCP 60 80 → 39244 [SYN, ACK]
681 58.535547475 10.0.2.15 35.222.85.5 TCP 54 39244 → 80 [ACK] Seq=1
682 58.537049725 10.0.2.15 35.222.85.5 HTTP 141 GET / HTTP/1.1
683 58.537442409 35.222.85.5 10.0.2.15 TCP 60 80 → 39244 [ACK] Seq=1
684 58.660714334 35.222.85.5 10.0.2.15 HTTP 202 HTTP/1.1 204 No Conter
685 58.660741546 10.0.2.15 35.222.85.5 TCP 54 39244 → 80 [ACK] Seq=8
686 58.661057200 35.222.85.5 10.0.2.15 TCP 60 80 → 39244 [FIN, ACK]
687 58.661903159 10.0.2.15 35.222.85.5 TCP 54 39244 → 80 [FIN, ACK]
688 58.662105950 35.222.85.5 10.0.2.15 TCP 60 80 → 39244 [ACK] Seq=1
Frame 1: 95 bytes on wire (760 bits), 95 bytes captured (760 bits) on interface enp0s3, id 0
Ethernet II, Src: PcsCompu_98:3d:cc (08:00:27:98:3d:cc), Dst: RealtekU_12:35:02 (52:54:00:12:35:02)
Internet Protocol Version 4, Src: 10.0.2.15, Dst: 75.75.75.75
0000 52 54 00 12 35 02 08 00 27 98 3d cc 08 00 45 00
0010 00 51 9c cc 40 00 40 11 fb 2a 0a 00 02 0f 4b 4b
0020 4b 4b c2 83 00 35 00 3d a2 f3 1c 6d 01 00 00 01
0030 00 00 00 00 00 01 0c 64 65 74 65 63 74 70 6f 72
0040 74 61 6c 07 66 69 72 65 66 6f 78 03 63 6f 6d 00
0050 00 01 00 01 00 00 29 02 00 00 00 00 00 00 00
File Edit View Search Terminal Help
Ether / IP / TCP 172.217.10.228:https > 10.0.2.15:44338 A / Padding
Ether / IP / TCP 172.217.10.228:https > 10.0.2.15:44338 A / Padding
Ether / IP / TCP 13.35.78.91:https > 10.0.2.15:35120 A / Padding
Ether / IP / TCP 13.35.78.91:https > 10.0.2.15:35120 A / Padding
Ether / IP / TCP 104.16.248.249:https > 10.0.2.15:56090 A / Padding
Ether / IP / TCP 104.16.248.249:https > 10.0.2.15:56090 A / Padding
Ether / IP / TCP 104.98.115.144:http > 10.0.2.15:32818 A / Padding
Ether / IP / TCP 104.98.115.144:http > 10.0.2.15:32816 A / Padding
Ether / IP / TCP 13.249.183.13:https > 10.0.2.15:39980 FA / Padding
Ether / IP / TCP 10.0.2.15:39980 > 13.249.183.13:https A
Ether / IP / TCP 13.35.78.56:https > 10.0.2.15:48398 FA / Padding
Ether / IP / TCP 10.0.2.15:48398 > 13.35.78.56:https A
Ether / IP / TCP 104.16.248.249:https > 10.0.2.15:56090 FA / Padding
Ether / IP / TCP 10.0.2.15:56090 > 104.16.248.249:https A
Ether / IP / TCP 172.217.6.226:https > 10.0.2.15:55060 FA / Padding
Ether / IP / TCP 10.0.2.15:55060 > 172.217.6.226:https A
Ether / IP / TCP 172.217.12.162:https > 10.0.2.15:37440 FA / Padding
Ether / IP / TCP 10.0.2.15:37440 > 172.217.12.162:https A
Ether / IP / TCP 72.21.91.29:http > 10.0.2.15:38180 FA / Padding
Ether / IP / TCP 10.0.2.15:38180 > 72.21.91.29:http A
Ether / IP / TCP 172.217.6.206:https > 10.0.2.15:60136 FA / Padding
Ether / IP / TCP 10.0.2.15:60136 > 172.217.6.206:https A
Ether / IP / TCP 13.35.78.91:https > 10.0.2.15:35120 FA / Padding
Ether / IP / TCP 10.0.2.15:35120 > 13.35.78.91:https A
Ether / IP / TCP 104.98.115.144:http > 10.0.2.15:32818 FA / Padding
Ether / IP / TCP 10.0.2.15:32818 > 104.98.115.144:http A
Ether / IP / TCP 172.217.7.3:https > 10.0.2.15:48696 FA / Padding
Ether / IP / TCP 10.0.2.15:48696 > 172.217.7.3:https A
Ether / IP / TCP 172.217.7.3:http > 10.0.2.15:38368 FA / Padding
Ether / IP / TCP 10.0.2.15:38368 > 172.217.7.3:http A
Ether / IP / TCP 104.98.115.144:http > 10.0.2.15:32816 FA / Padding
Ether / IP / TCP 10.0.2.15:32816 > 104.98.115.144:http A
Ether / IP / TCP 172.217.10.228:https > 10.0.2.15:44338 FA / Padding
Ether / IP / TCP 10.0.2.15:44338 > 172.217.10.228:https A
Ether / IP / TCP 52.39.78.187:https > 10.0.2.15:42734 PA / Raw
Ether / IP / TCP 10.0.2.15:42734 > 52.39.78.187:https R
Ether / IP / TCP 52.39.78.187:https > 10.0.2.15:42732 PA / Raw
Ether / IP / TCP 10.0.2.15:42732 > 52.39.78.187:https R
Ether / IP / TCP 54.191.252.154:https > 10.0.2.15:46848 PA / Raw
Ether / IP / TCP 10.0.2.15:46848 > 54.191.252.154:https R
Ether / IP / TCP 10.0.2.15:39244 > 35.222.85.5:http S
Ether / IP / TCP 35.222.85.5:http > 10.0.2.15:39244 SA / Padding
Ether / IP / TCP 10.0.2.15:39244 > 35.222.85.5:http A
Ether / IP / TCP 10.0.2.15:39244 > 35.222.85.5:http PA / Raw
Ether / IP / TCP 35.222.85.5:http > 10.0.2.15:39244 A / Padding
Ether / IP / TCP 35.222.85.5:http > 10.0.2.15:39244 PA / Raw
Ether / IP / TCP 10.0.2.15:39244 > 35.222.85.5:http A
Ether / IP / TCP 35.222.85.5:http > 10.0.2.15:39244 FA / Padding
Ether / IP / TCP 10.0.2.15:39244 > 35.222.85.5:http FA
Ether / IP / TCP 35.222.85.5:http > 10.0.2.15:39244 A / Padding

Controlled Sniffing Display:

- There are a bunch of filters and methods that one can use to simplify what is seen from the sniffing similar to how Wireshark is done
- Below is an example of using **sprintf()** to format how data is displayed from each packet being sent. In this case, it prints the sending and receiving IP addresses along with any Raw data being sent in the packet and is then stored in the variable **pkts**

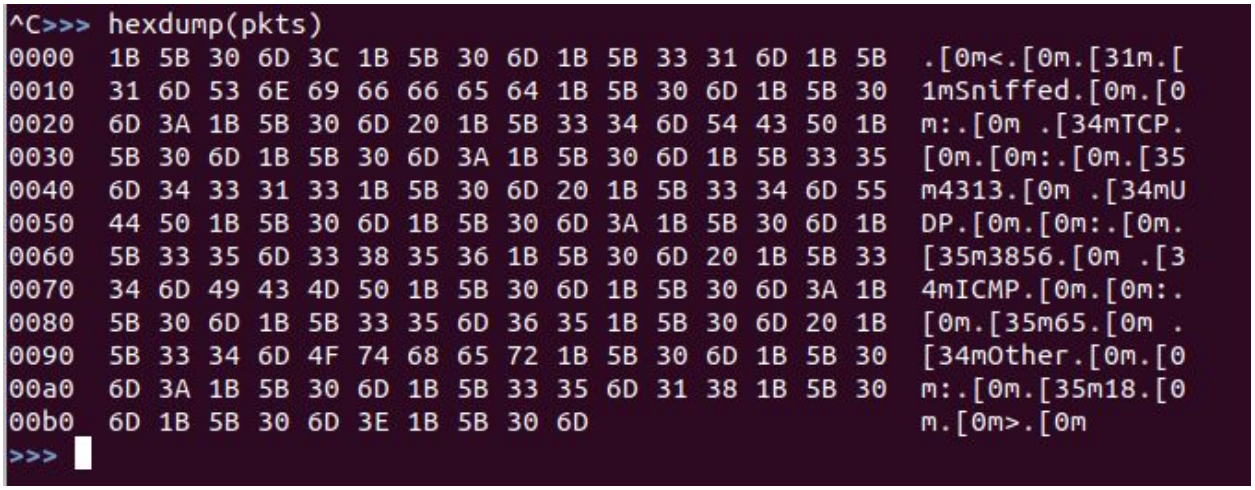
```
pkts = sniff(prn=lambda x:x.sprintf("{IP:%IP.src% ->
%IP.dst%\n}{Raw:%Raw.load%\n}"))
```

```
172.217.10.65 -> 10.0.2.15

172.217.10.65 -> 10.0.2.15
'\x16\x03\x03\x00\x80\x02\x00\x00|\x03\x03\xcbfWn\xa2\xd3\x97\x13\xff\x84\x96,\x9b\x1f7\x88\x08Q\xe7\
x15co\xb7\xe1K\xab\xe3\xa0m"\x87R a\x0c\xf8\\\'#\x9bJ\x80RP\xb3\xfc\xb5\xaa\xbd\x9cp\xf0\xf0\xcc\xbf\
xf0\x95\xac\x93\xc9\x92\x8aNpx\x13\x01\x00\x004\x00)\x00\x02\x00\x00\x003\x00$\x00\x1d\x00 T@\x11+\xd
7W\x8f\xdcl\xa0\xf9d\xe7\'\xd2g\xb3Y\xe30\x82\x14\xdb\xc6B\x16I\x8d\xb5\xce\x84v\x00+\x00\x02\x03\x04
\x14\x03\x03\x00\x01\x01\x17\x03\x03\x00D\xc6\x12\xc4\x0b\xf1\xd6\xda\xb8\xb8\xd6r\xe5`\xfe\xda\xac\x
1c\x08\xd1\x8a\xb0\x18\xbb\xcd}\x91\x92\xcbu!\xd8\\\xf9\xaaOE\xa3\x0b\xb9\x1f\xe4\x8c\x11\x1b\x1a\x8a
\x07\x0e\xf3\xe8\xef\x15\x9a\x87\x97\xec\xdc\xb4\xf6\xcf[\xad\x1a\xc0\xd4\xc7\xfa\xf6'

10.0.2.15 -> 172.217.10.65

10.0.2.15 -> 172.217.10.65
'\x14\x03\x03\x00\x01\x01\x17\x03\x03\x005\xe5\xcf4<\r\x02w\x7f!\xcb#\xbb\x8f\xaf\x08\xd1--\xf6\xfe+i
\x99\x02g\xd6\x8f\r\x06m\xd4\xbb\x80\x0bd\xbe\xb6p)\xae\xa6\xe9\xd2A_\xa0<D\xe2RS;\x9d'

172.217.10.65 -> 10.0.2.15

10.0.2.15 -> 172.217.10.65
"\x17\x03\x03\x00\xa54G\x86m\x10lTu\x08\xa2\x00\xc8^\xd5\x06\xa2\x0e~5\xe1EN!\x8c{\n\x94\xa1\xc5\x85\
xfdlh\x05\x8e\x1e\x10Z\x81\xde?P\xe6\xcc\xbe\n\xa0%\xbd:\x8fz\xfa'\xa8%9\xe1/\xf5b\xc2\xaeU\x08{%qg\x
a2<H\xe8\x01\xc1\xde\xf7X\x02\xe2h9A'X\x1b/\xe6\xbaa\xff\x94\xbb\xb1\x08\x8a\xbaN:\xe3U\xe5\xb6\xf0\x
c8\xe9gdiW\xdf\x1c!\xed\xbb\x8d\xf4\x9cJ\x0e\xa95\xcaN*#V>\x8e\xecuyn\xd9\xfes\x9c3\xe2\xeb\xae\xbb\x
13V\x1f\xeb\xb3!\xf2 S3\xef\xe4r\xd9\xbe\xd4X\xfa\xd1!\t\xf0\xb8"

172.217.10.65 -> 10.0.2.15
```

- This can be stored/read for later use using the following lines:

```
wrpcap("Session1.cap",pkts)
pkts = rdpcap("Session1.cap")
```

- Using **hexdump(pkts)** displays **pkts** data in a hexdump format like below



Exporting to Wireshark:

- Scapy has the capability of calling Wireshark when you want to use it to analyze packets stored in a variable
- Just simply use `wireshark(pkts)` and Scapy will open Wireshark up for you to analyze the packets

Using Scapy in Python Scripting:

- Because Scapy is built on Python, everything being used above, can be used in a Python script as well
- All you need is to import from scapy.all import * and you have access to everything you need
    - Do note that functionality like sending packets requires Sublime or whatever editor you're using to be run as administrator to work
- Below is an example of running the **sr1()** example in a Python script

File Edit Selection Find View Goto Tools Project Preferences Help

ScapyLab.py

```
1  from scapy.all import *
2  p = sr1(IP(dst="www.slashdot.org")/ICMP()/"Raw text being sent for testing")
3
4  if p:
5      p.show()
```

```
Begin emission:
.Finished sending 1 packets.
*
Received 2 packets, got 1 answers, remaining 0 packets
###[ IP ]###
  version   = 4
  ihl       = 5
  tos       = 0x0
  len       = 59
  id        = 53332
  flags     =
  frag      = 0
  ttl       = 47
  proto     = icmp
  chksum    = 0xb0e6
  src       = 216.105.38.15
  dst       = 10.0.2.15
  \options   \
###[ ICMP ]###
     type      = echo-reply
     code      = 0
     chksum    = 0xca96
     id        = 0x0
     seq       = 0x0
###[ Raw ]###
        load      = 'Raw text being sent for testing'

[Finished in 0.9s]
```