

Exercise 1

Traffic capture

The ethernet connection is associated with `0c:37:96:5f:8a:16` (from the perspective of the Linux machine) under `enx0c37965f8a16`. On the raspberry pi, the connection is labelled with the much nicer `eth0` (if you like Minecraft this is also the name of a popular youtuber).

Starting Wireshark and looking at this interface, we see this message:

15	30.106941338	Raspberr_8d:c8:32	BizlinkT_5f:8a:16	ARP	60	Who has 192.168.10.1? Tell 192.168.10.2
16	30.106964187	BizlinkT_5f:8a:16	Raspberr_8d:c8:32	ARP	42	192.168.10.1 is at 0c:37:96:5f:8a:16

which matches the MAC address that we took earlier.

If we use the `ping` command,

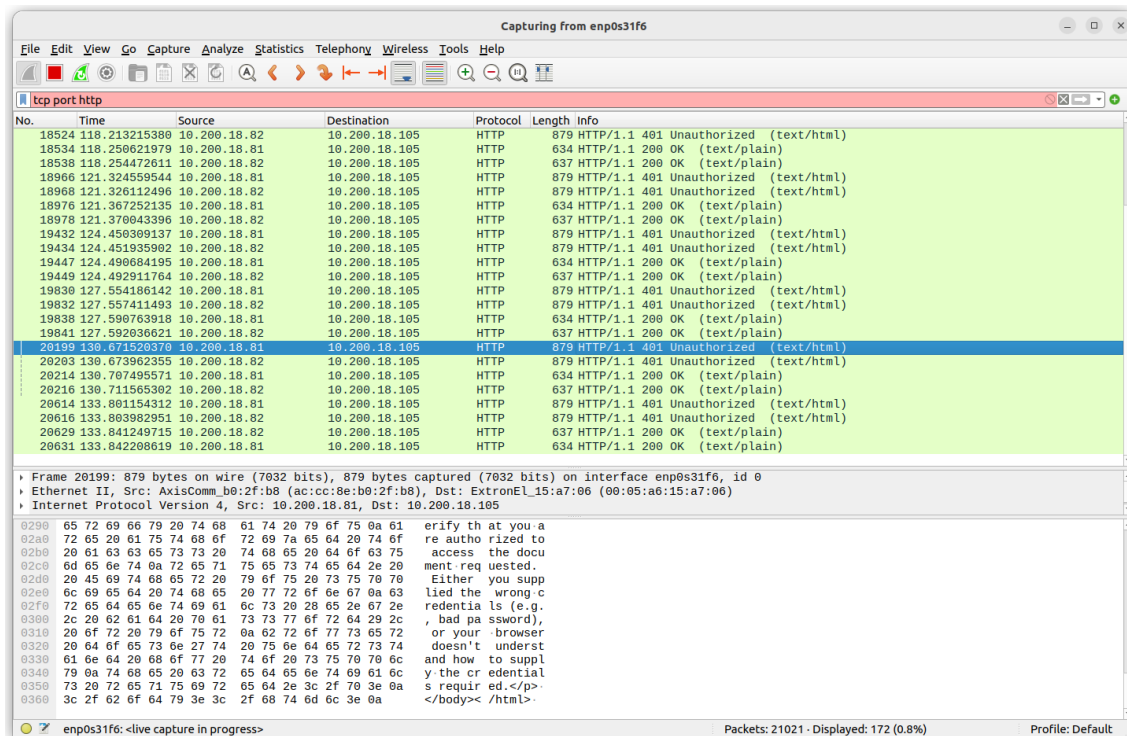
Source	Destination	Protocol	Length	Info
192.168.10.2	192.168.10.1	SSH	110	Server: Encrypted packet (len=44)
192.168.10.1	192.168.10.2	TCP	66	34124 → 22 [ACK] Seq=1081 Ack=1321 Win=501 Len=0 TSval=379983081
192.168.10.1	192.168.10.2	SSH	102	Client: Encrypted packet (len=36)
192.168.10.2	192.168.10.1	SSH	110	Server: Encrypted packet (len=44)
192.168.10.1	192.168.10.2	TCP	66	34124 → 22 [ACK] Seq=1117 Ack=1365 Win=501 Len=0 TSval=379983977
192.168.10.1	192.168.10.2	SSH	102	Client: Encrypted packet (len=36)
192.168.10.2	192.168.10.1	SSH	110	Server: Encrypted packet (len=52)
192.168.10.1	192.168.10.2	TCP	66	34124 → 22 [ACK] Seq=1153 Ack=1417 Win=501 Len=0 TSval=379984319
192.168.10.2	192.168.10.1	ICMP	98	Echo (ping) request id=0x0001, seq=1/256, ttl=64 (reply in 216)
192.168.10.2	192.168.10.1	SSH	158	Server: Encrypted packet (len=92)
192.168.10.1	192.168.10.2	ICMP	98	Echo (ping) reply id=0x0001, seq=1/256, ttl=64 (request in 214)
192.168.10.1	192.168.10.2	TCP	66	34124 → 22 [ACK] Seq=1153 Ack=1509 Win=501 Len=0 TSval=379984325
192.168.10.2	192.168.10.1	SSH	166	Server: Encrypted packet (len=100)
192.168.10.1	192.168.10.2	TCP	66	34124 → 22 [ACK] Seq=1153 Ack=1609 Win=501 Len=0 TSval=379984325
192.168.10.2	1.1.1.1	DNS	81	Standard query 0x33a9 A 2.debian.pool.ntp.org
192.168.10.2	1.1.1.1	DNS	81	Standard query 0x0eae AAAA 2.debian.pool.ntp.org
192.168.10.2	192.168.10.1	ICMP	98	Echo (ping) request id=0x0001, seq=2/512, ttl=64 (reply in 223)
192.168.10.1	192.168.10.2	ICMP	98	Echo (ping) reply id=0x0001, seq=2/512, ttl=64 (request in 222)
192.168.10.2	192.168.10.1	SSH	166	Server: Encrypted packet (len=100)
192.168.10.1	192.168.10.2	TCP	66	34124 → 22 [ACK] Seq=1153 Ack=1709 Win=501 Len=0 TSval=379985355
192.168.10.2	192.168.10.1	ICMP	98	Echo (ping) request id=0x0001, seq=3/768, ttl=64 (reply in 227)
192.168.10.1	192.168.10.2	ICMP	98	Echo (ping) reply id=0x0001, seq=3/768, ttl=64 (request in 226)
192.168.10.2	192.168.10.1	SSH	166	Server: Encrypted packet (len=100)

Here we see that the raspberry pi (ip: 192.168.10.2) sends a ping request, 64 bytes long, to the Linux machine (ip:192.168.10.1) which returns a reply to the raspberry pi.

Over 560 pings, the raspberry pi reports a mean time between request and response of 0.516 ms with standard deviation 0.075 ms. The packet loss reported is zero (all 560 requests received a reply), so the ethernet cable seems to be working well.

Switching to the departmental connection (ip: 10.200.17.151/22), the network is much busier. We see many different protocols appearing, such as TCP, ARP, GVCP and STP.

To filter for `http` requests, we first lookup the filter under the *Capture* tab and find that we need the filter expression `tcp port http`. Here is a view of the result:



We can see that the packet's payload contains a message formatted in HTML. I'm not entirely sure where it comes from though.

```

pi@p4pi: ~
pi@p4pi: ~
ubuntu@ubuntu: ~

, options [nop,nop,TS val 382177162 ecr 2705141579], length 36
16:56:55.182725 IP 192.168.10.2.ssh > 192.168.10.1.34124: Flags [P.], seq 304:340, ack 109, win 50
1, options [nop,nop,TS val 2705141667 ecr 382177162], length 36
pi@p4pi:~ $ sudo tcpdump -l eth0 -c 10 -w captured2.pcap
tcpdump: listening on eth0, link-type EN10MB (Ethernet), snapshot length 262144 bytes
10 packets captured
10 packets received by filter
0 packets dropped by kernel
pi@p4pi:~ $ tcpdump -r captured2.pcap
reading from file captured2.pcap, link-type EN10MB (Ethernet), snapshot length 262144
16:58:00.959601 IP 192.168.10.2.ssh > 192.168.10.1.34124: Flags [P.], seq 1510838147:1510838191, a
ck 3783456519, win 501, options [nop,nop,TS val 2705207443 ecr 382242885], length 44
16:58:00.959716 IP 192.168.10.2.ssh > 192.168.10.1.34124: Flags [P.], seq 44:96, ack 1, win 501, o
ptions [nop,nop,TS val 2705207444 ecr 382242885], length 52
16:58:00.959804 IP 192.168.10.2.ssh > 192.168.10.1.34124: Flags [P.], seq 96:164, ack 1, win 501,
options [nop,nop,TS val 2705207444 ecr 382242885], length 68
16:58:00.959894 IP 192.168.10.2.ssh > 192.168.10.1.34124: Flags [P.], seq 164:232, ack 1, win 501,
options [nop,nop,TS val 2705207444 ecr 382242885], length 68
16:58:00.960258 IP 192.168.10.1.34124 > 192.168.10.2.ssh: Flags [.], ack 44, win 679, options [nop
,nop,TS val 382242938 ecr 2705207443], length 0
16:58:00.960259 IP 192.168.10.1.34124 > 192.168.10.2.ssh: Flags [.], ack 96, win 679, options [nop
,nop,TS val 382242938 ecr 2705207444], length 0
16:58:00.960259 IP 192.168.10.1.34124 > 192.168.10.2.ssh: Flags [.], ack 164, win 679, options [no
p,nop,TS val 382242938 ecr 2705207444], length 0
16:58:00.960319 IP 192.168.10.1.34124 > 192.168.10.2.ssh: Flags [.], ack 232, win 679, options [no
p,nop,TS val 382242938 ecr 2705207444], length 0

```

Here we capture 10 packets on the raspberry pi using *tcpdump*. I'm not sure what to say about these, either.

Sending traffic

Using the script provided in the Github repo, we may send 100 packets with the command :

```
sudo python3 send.py 100 enx0c37965f8a16 192.168.10.1 192.168.10.2
```

No.	Time	Source	Destination	Protocol	Length	Info
91	17.215226826	192.168.10.1	192.168.10.2	UDP	64	50000 → 1024 Len=22
92	17.270977564	192.168.10.1	192.168.10.2	UDP	64	50000 → 1024 Len=22
93	17.331097158	192.168.10.1	192.168.10.2	UDP	64	50000 → 1024 Len=22
94	17.403559648	192.168.10.1	192.168.10.2	UDP	64	50000 → 1024 Len=22
95	17.463010437	192.168.10.1	192.168.10.2	UDP	64	50000 → 1024 Len=22
96	17.523177822	192.168.10.1	192.168.10.2	UDP	64	50000 → 1024 Len=22
97	17.608053785	192.168.10.1	192.168.10.2	UDP	64	50000 → 1024 Len=22
98	17.671056388	192.168.10.1	192.168.10.2	UDP	64	50000 → 1024 Len=22
99	17.739078884	192.168.10.1	192.168.10.2	UDP	64	50000 → 1024 Len=22
100	17.815054279	192.168.10.1	192.168.10.2	UDP	64	50000 → 1024 Len=22
101	17.879095008	192.168.10.1	192.168.10.2	UDP	64	50000 → 1024 Len=22
102	17.935070268	192.168.10.1	192.168.10.2	UDP	64	50000 → 1024 Len=22
103	17.982391448	192.168.10.1	192.168.10.2	UDP	64	50000 → 1024 Len=22
104	18.030429840	192.168.10.1	192.168.10.2	UDP	64	50000 → 1024 Len=22
105	18.095254620	192.168.10.1	192.168.10.2	UDP	64	50000 → 1024 Len=22
106	18.167952640	192.168.10.1	192.168.10.2	UDP	64	50000 → 1024 Len=22
107	18.223232219	192.168.10.1	192.168.10.2	UDP	64	50000 → 1024 Len=22
108	18.275152460	192.168.10.1	192.168.10.2	UDP	64	50000 → 1024 Len=22
109	18.350319353	192.168.10.1	192.168.10.2	UDP	64	50000 → 1024 Len=22
110	18.423231660	192.168.10.1	192.168.10.2	UDP	64	50000 → 1024 Len=22
111	18.475081066	192.168.10.1	192.168.10.2	UDP	64	50000 → 1024 Len=22
112	18.551221999	192.168.10.1	192.168.10.2	UDP	64	50000 → 1024 Len=22
113	18.623361184	192.168.10.1	192.168.10.2	UDP	64	50000 → 1024 Len=22
114	18.691392835	192.168.10.1	192.168.10.2	UDP	64	50000 → 1024 Len=22
▶ Frame 109: 64 bytes on wire (512 bits), 64 bytes captured (512 bits) on interface enx0c37965f8a16, id 0 ▶ Ethernet II, Src: 00:00:00_00:00:02 (00:00:00:00:00:02), Dst: 00:00:00_00:00:01 (00:00:00:00:00:01) ▶ Internet Protocol Version 4, Src: 192.168.10.1, Dst: 192.168.10.2						
0000	00 00 00 00 00 01 00 00	00 00 00 02 08 00 45 00E.			
0010	00 32 00 01 00 00 40 11	e5 66 c0 a8 0a 01 c0 a8	.2...@.f.....			
0020	0a 02 c3 50 04 00 00 1e	c9 55 75 64 78 70 6c 74	...P.....Uudxp1t			
0030	75 74 6b 6b 67 70 74 69	72 6b 79 67 6f 67 67 7a	utkkgpti rkygoggz			

We can repeat the same with our duplicated repo on the raspberry pi (switching the IP addresses and changing the connection name to *eth0*):

The screenshot shows the Wireshark interface with a packet capture of a network traffic. The top pane displays a list of packets. The middle pane shows the details of the selected packet (No. 44), including the User Datagram Protocol (UDP) section. The bottom pane shows the raw data of the packet in hexadecimal and ASCII format.

The packet list shows the following details for packet 44:

No.	Time	Source	Destination	Protocol	Length	Info
44	3.649809890	192.168.10.2	192.168.10.1	UDP	64	50000 → 1024 Len=22

The details pane for the selected packet shows the following information:

- Ethernet II, Src: Intel (08:00:00:00:00:00), Dst: Intel (08:00:00:00:00:00)
- Internet Protocol Version 4, Src: 192.168.10.2, Dst: 192.168.10.1
- User Datagram Protocol, Src Port: 50000, Dst Port: 1024

The raw data pane shows the following hexadecimal and ASCII representation of the packet data:

```

0000 00 00 00 00 00 01 00 00 00 00 02 08 00 45 00 .....E
0010 00 32 00 01 00 00 40 11 e5 66 c0 a8 0a 02 c0 a8 -2...@..f....
0020 0a 01 c3 50 04 00 00 1e 00 32 62 7a 67 73 6c 7a ...P....2bzgslz
0030 6c 6c 72 72 61 66 73 65 74 6f 79 75 68 78 62 6b llrrafse toyuhxbk
  
```

The bottom pane shows the Python script used to generate the traffic:

```

def randomword(length):
    return ''.join(random.choice(string.ascii_lowercase) for i in range(length))

def send_random_traffic(num_packets, interface, src_ip, dst_ip):
    dst_mac = "00:00:00:00:00:01"
    src_mac = "00:00:00:00:00:02"
    total_pkts = 0
    port = 1024
    for i in range(num_packets):
        data = randomword(22)
        p = Ether(dst=dst_mac, src=src_mac)/IP(dst=dst_ip, src=src_ip)
        p = p/UDP(sport=50000, dport=port)/Raw(load=data)
        sendp(p, iface=interface, inter=0.01)
        # If you want to see the contents of the packet, uncomment the line below
        # print(p.show())
        total_pkts += 1
    print("Sent %s packets in total" % total_pkts)
  
```

Looking at the code, we see that the protocol used to send the strings generated in python is UDP. Indeed, the number of bytes in the payload of each UDP packet is 22, matching the value in line 17 of the python code:

The screenshot shows the Wireshark interface with the details pane expanded for the selected packet (No. 44). The details pane shows the following information:

- Ethernet II, Src: Intel (08:00:00:00:00:00), Dst: Intel (08:00:00:00:00:00)
- Internet Protocol Version 4, Src: 192.168.10.2, Dst: 192.168.10.1
- User Datagram Protocol, Src Port: 50000, Dst Port: 1024
- Data (22 bytes): 627a67736c7a6c6c727261667365746f79756878626b

The raw data pane shows the following hexadecimal and ASCII representation of the packet data:

```

0000 00 00 00 00 00 01 00 00 00 00 02 08 00 45 00 .....E
0010 00 32 00 01 00 00 40 11 e5 66 c0 a8 0a 02 c0 a8 -2...@..f....
0020 0a 01 c3 50 04 00 00 1e 00 32 62 7a 67 73 6c 7a ...P....2bzgslz
0030 6c 6c 72 72 61 66 73 65 74 6f 79 75 68 78 62 6b llrrafse toyuhxbk
  
```

Hence, the relevant filter expression in WireShark is *udp*.

*enx0c37965f8a16

File Edit View Go Capture Analyze Statistics Telephony Wireless Tools Help

udp

No.	Time	Source	Destination	Protocol	Length	Info
462	10.416860943	192.168.10.2	192.168.10.1	UDP	64	50000 → 1024 Len=22
468	10.464879593	192.168.10.2	192.168.10.1	UDP	64	50000 → 1024 Len=22
473	10.524751872	192.168.10.2	192.168.10.1	UDP	64	50000 → 1024 Len=22
478	10.580753727	192.168.10.2	192.168.10.1	UDP	64	50000 → 1024 Len=22
483	10.640747699	192.168.10.2	192.168.10.1	UDP	64	50000 → 1024 Len=22
489	10.700736231	192.168.10.2	192.168.10.1	UDP	64	50000 → 1024 Len=22
495	10.758508882	192.168.10.2	192.168.10.1	UDP	64	50000 → 1024 Len=22
501	10.808721116	192.168.10.2	192.168.10.1	UDP	64	50000 → 1024 Len=22
507	10.860662920	192.168.10.2	192.168.10.1	UDP	64	50000 → 1024 Len=22
513	10.920815597	192.168.10.2	192.168.10.1	UDP	64	50000 → 1024 Len=22
519	10.980676327	192.168.10.2	192.168.10.1	UDP	64	50000 → 1024 Len=22
525	11.040755166	192.168.10.2	192.168.10.1	UDP	64	50000 → 1024 Len=22
531	11.104810898	192.168.10.2	192.168.10.1	UDP	64	50000 → 1024 Len=22
536	11.152818353	192.168.10.2	192.168.10.1	UDP	64	50000 → 1024 Len=22

Data: 626f7a7373697277272661766379676a666d6367787a
[Length: 22]

```

0000 00 00 00 00 00 01 00 00 00 00 00 02 08 00 45 00 .....E.
0010 00 32 00 01 00 00 40 11 e5 66 c0 a8 0a 02 c0 a8 .2...@.f.....
0020 0a 01 c3 50 04 00 00 1e ff 29 62 6f 7a 73 73 69 ...P.....)bozss1
0030 72 77 72 76 61 76 63 79 67 6a 66 6d 63 67 78 7a rwravcy gjfmcgxz

```

Data (data.data), 22 byte(s) Packets: 620 · Displayed: 126 (20.3%) Profile: Default

The total packet size is 64 bytes, since there are 64 hexadecimal digit pairs. Then we know that the header takes up $64 - 22 = 42$ bytes. So initially I used a string length of $512 - 42 = 470$ bytes, but I forgot that TCP is a different protocol and hence might have a different header! At a string length of 470 this yielded packets of size 524, so I actually needed a string length of 458.

Capturing from enx0c37965f8a16

File Edit View Go Capture Analyze Statistics Telephony Wireless Tools Help

Apply a display filter ... <Ctrl-/>

No.	Time	Source	Destination	Protocol	Length	Info
1076	157.433516142	192.168.10.2	192.168.10.1	TCP	524	[TCP Retransmission] [TCP Port numbers reused] 50000 → 5555 [SYN] Seq=0...
1077	157.481495813	192.168.10.2	192.168.10.1	TCP	524	[TCP Retransmission] [TCP Port numbers reused] 50000 → 5555 [SYN] Seq=0...
1078	157.541661897	192.168.10.2	192.168.10.1	TCP	524	[TCP Retransmission] [TCP Port numbers reused] 50000 → 5555 [SYN] Seq=0...
1079	157.589649347	192.168.10.2	192.168.10.1	TCP	524	[TCP Retransmission] [TCP Port numbers reused] 50000 → 5555 [SYN] Seq=0...
1080	157.637587582	192.168.10.2	192.168.10.1	TCP	524	[TCP Retransmission] [TCP Port numbers reused] 50000 → 5555 [SYN] Seq=0...
1081	157.694096292	192.168.10.2	192.168.10.1	TCP	524	[TCP Retransmission] [TCP Port numbers reused] 50000 → 5555 [SYN] Seq=0...
1082	157.745486237	192.168.10.2	192.168.10.1	TCP	524	[TCP Retransmission] [TCP Port numbers reused] 50000 → 5555 [SYN] Seq=0...
1083	157.797554567	192.168.10.2	192.168.10.1	TCP	524	[TCP Retransmission] [TCP Port numbers reused] 50000 → 5555 [SYN] Seq=0...
1084	157.849465431	192.168.10.2	192.168.10.1	TCP	524	[TCP Retransmission] [TCP Port numbers reused] 50000 → 5555 [SYN] Seq=0...
1085	157.901595354	192.168.10.2	192.168.10.1	TCP	524	[TCP Retransmission] [TCP Port numbers reused] 50000 → 5555 [SYN] Seq=0...
1086	157.953653167	192.168.10.2	192.168.10.1	TCP	524	[TCP Retransmission] [TCP Port numbers reused] 50000 → 5555 [SYN] Seq=0...
1087	158.005594206	192.168.10.2	192.168.10.1	TCP	524	[TCP Retransmission] [TCP Port numbers reused] 50000 → 5555 [SYN] Seq=0...
1088	158.065569233	192.168.10.2	192.168.10.1	TCP	524	[TCP Retransmission] [TCP Port numbers reused] 50000 → 5555 [SYN] Seq=0...
1089	158.117649379	192.168.10.2	192.168.10.1	TCP	524	[TCP Retransmission] [TCP Port numbers reused] 50000 → 5555 [SYN] Seq=0...

The erroneous packets

Capturing from enx0c37965f8a16

File Edit View Go Capture Analyze Statistics Telephony Wireless Tools Help

Apply a display filter ... <Ctrl-/>

No.	Time	Source	Destination	Protocol	Length	Info
420	36.711857065	192.168.10.2	192.168.10.1	TCP	512	[TCP Retransmission] [TCP Port numbers reused] 50000 → 5555 [SYN] Seq=0...
421	36.763866829	192.168.10.2	192.168.10.1	TCP	512	[TCP Retransmission] [TCP Port numbers reused] 50000 → 5555 [SYN] Seq=0...
422	36.815882840	192.168.10.2	192.168.10.1	TCP	512	[TCP Retransmission] [TCP Port numbers reused] 50000 → 5555 [SYN] Seq=0...
423	36.872420678	192.168.10.2	192.168.10.1	TCP	512	[TCP Retransmission] [TCP Port numbers reused] 50000 → 5555 [SYN] Seq=0...
424	36.927847727	192.168.10.2	192.168.10.1	TCP	512	[TCP Retransmission] [TCP Port numbers reused] 50000 → 5555 [SYN] Seq=0...
425	36.979932797	192.168.10.2	192.168.10.1	TCP	512	[TCP Retransmission] [TCP Port numbers reused] 50000 → 5555 [SYN] Seq=0...
426	37.027838369	192.168.10.2	192.168.10.1	TCP	512	[TCP Retransmission] [TCP Port numbers reused] 50000 → 5555 [SYN] Seq=0...
427	37.071885737	192.168.10.2	192.168.10.1	TCP	512	[TCP Retransmission] [TCP Port numbers reused] 50000 → 5555 [SYN] Seq=0...
428	37.127923299	192.168.10.2	192.168.10.1	TCP	512	[TCP Retransmission] [TCP Port numbers reused] 50000 → 5555 [SYN] Seq=0...
429	37.195909867	192.168.10.2	192.168.10.1	TCP	512	[TCP Retransmission] [TCP Port numbers reused] 50000 → 5555 [SYN] Seq=0...
430	37.251928404	192.168.10.2	192.168.10.1	TCP	512	[TCP Retransmission] [TCP Port numbers reused] 50000 → 5555 [SYN] Seq=0...
431	37.303837819	192.168.10.2	192.168.10.1	TCP	512	[TCP Retransmission] [TCP Port numbers reused] 50000 → 5555 [SYN] Seq=0...
432	37.356422481	192.168.10.2	192.168.10.1	TCP	512	[TCP Retransmission] [TCP Port numbers reused] 50000 → 5555 [SYN] Seq=0...
433	37.407887232	192.168.10.2	192.168.10.1	TCP	512	[TCP Retransmission] [TCP Port numbers reused] 50000 → 5555 [SYN] Seq=0...

.....0..... = IG bit: Individual address (unicast)

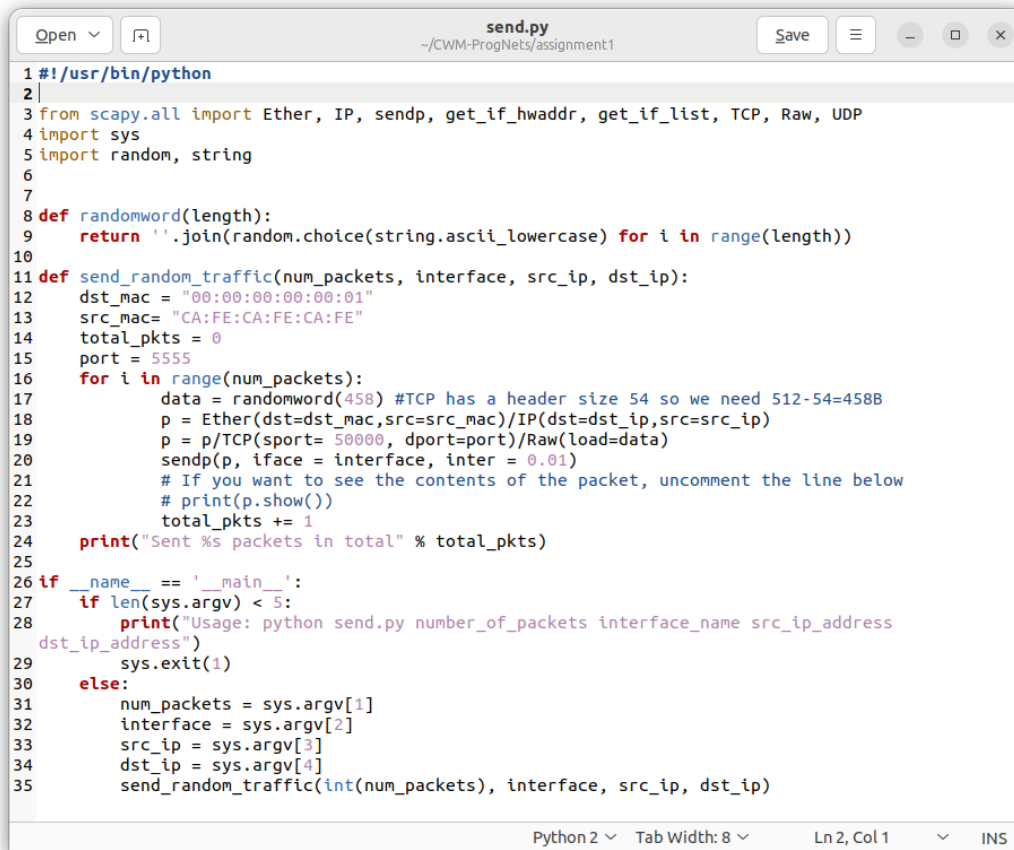
Source: ca:fe:ca:fe:ca:fe (ca:fe:ca:fe:ca:fe)

0000	00 00 00 00 01 ca fe	ca fe ca fe 08 00 45 00E.....
0010	01 f2 00 01 00 00 40 06	e3 b1 c0 a8 0a 02 c0 a8@.....
0020	0a 01 c3 50 15 b3 00 00	00 00 00 00 00 50 02	...P.....P...
0030	20 00 13 17 00 00 71 09	62 65 6c 6e 61 71 72 70qi belnaqrp
0040	6a 66 72 62 7a 76 76 6e	78 74 6a 76 6f 6d 62 61	jfrbzvvn xtjvomba
0050	6e 6b 72 6f 71 6e 6e 73	79 65 74 70 79 75 6e 79	nkroqnnn yetpyuny
0060	7a 76 6a 63 6d 68 7a 66	70 61 68 6f 63 65 69 65	zvjcmbzf pahoceie
0070	62 77 75 79 78 61 62 76	76 61 6b 73 6e 63 6b 73	bwuyxabv vaksncks
0080	77 6c 68 77 68 74 74 6a	77 6a 74 69 75 6c 79 71	wlhwhtttj wjtulyq

Specifies if this is an individual (unicast) or group (broadcast/multicast) address (eth.dst.ig), 3 byte(s) Packets: 479 · Displayed: 479 (100.0%) Profile: Default

The correct packet size!

Edited python code



```
1#!/usr/bin/python
2|
3from scapy.all import Ether, IP, sendp, get_if_hwaddr, get_if_list, TCP, Raw, UDP
4import sys
5import random, string
6
7
8def randomword(length):
9    return ''.join(random.choice(string.ascii_lowercase) for i in range(length))
10
11def send_random_traffic(num_packets, interface, src_ip, dst_ip):
12    dst_mac = "00:00:00:00:00:01"
13    src_mac = "CA:FE:CA:FE:CA:FE"
14    total_pkts = 0
15    port = 5555
16    for i in range(num_packets):
17        data = randomword(458) #TCP has a header size 54 so we need 512-54=458B
18        p = Ether(dst=dst_mac,src=src_mac)/IP(dst=dst_ip,src=src_ip)
19        p = p/TCP(sport= 50000, dport=port)/Raw(load=data)
20        sendp(p, iface = interface, inter = 0.01)
21        # If you want to see the contents of the packet, uncomment the line below
22        # print(p.show())
23        total_pkts += 1
24    print("Sent %s packets in total" % total_pkts)
25
26if __name__ == '__main__':
27    if len(sys.argv) < 5:
28        print("Usage: python send.py number_of_packets interface_name src_ip_address dst_ip_address")
29        sys.exit(1)
30    else:
31        num_packets = sys.argv[1]
32        interface = sys.argv[2]
33        src_ip = sys.argv[3]
34        dst_ip = sys.argv[4]
35        send_random_traffic(int(num_packets), interface, src_ip, dst_ip)
```

Python 2 ▾ Tab Width: 8 ▾ Ln 2, Col 1 ▾ INS