VirtualBox tweak

- By default, VirtualBox places your virtual machine in a private network. We need to tweak the VirtualBox configuration a bit to make the virtual machine appear "alongside" your host machine on the same network.

  Halt your virtual machine (ie, power it off—saving state is not sufficient here). Go to Settings -¿ Network. Set the "Attached to" box to "Bridged Adapter". Under the Advanced settings on the same tab, set "Promiscuous Mode" to "Allow VMs".

```
[@majd 7]$ ifconfig
enp0s3: flags=4163<UP,BROADCAST,RUNNING,MULTICAST>  mtu 1500
        inet 140.233.103.98  netmask 255.255.255.0  broadcast
   255.255.255.255
        inet6 fe80::353d:4b9d:680e:7fd7  prefixlen 64  scopeid 0x20<link>
        ether 08:00:27:75:2d:0b  txqueuelen 1000  (Ethernet)
        RX packets 8088  bytes 3427677 (3.2 MiB)
        RX errors 0  dropped 0  overruns 0  frame 0
        TX packets 5648  bytes 799742 (780.9 KiB)
        TX errors 0  dropped 0 overruns 0  carrier 0  collisions 0

lo: flags=73<UP,LOOPBACK,RUNNING>  mtu 65536
        inet 127.0.0.1  netmask 255.0.0.0
        inet6 ::1  prefixlen 128  scopeid 0x10<host>
        loop  txqueuelen 1000  (Local Loopback)
        RX packets 40  bytes 2000 (1.9 KiB)
        RX errors 0  dropped 0  overruns 0  frame 0
        TX packets 40  bytes 2000 (1.9 KiB)
        TX errors 0  dropped 0 overruns 0  carrier 0  collisions 0
```

1 Observe DHCP

- Start up Wireshark, listen on your virtual machine's external interface (ie, enp0s3), and set the filter for udp. Now, in a terminal, run this command to restart the dhcpcd process (which is what sends DHCP requests and processes DHCP responses):

  ```
  $ sudo systemctl restart dhcpcd@enp0s3
  ```

  Observe the sequence of packets in Wireshark. Note the encapsulation. Note the various pieces of information contained in the DHCP response.

When running the above command, two DHCP packets one request and the other is ack as follows:

```
696 325.913807075 0.0.0.0 255.255.255.255 DHCP  342 DHCP Request  -
   Transaction ID 0xd1d1a37e
```

```
698 325.921262953 140.233.103.1 255.255.255.255 DHCP  371 DHCP ACK      -
   Transaction ID 0xd1d1a37e
```

Wireshark Packet List

When we expand the DHCP request, we see:

```
Frame 696: 342 bytes on wire (2736 bits), 342 bytes captured (2736 bits)
   on interface enp0s3, id 0
Ethernet II, Src: PcsCompu_75:2d:0b (08:00:27:75:2d:0b), Dst: Broadcast (
   ff:ff:ff:ff:ff:ff)
Internet Protocol Version 4, Src: 0.0.0.0, Dst: 255.255.255.255
    0100 .... = Version: 4
    .... 0101 = Header Length: 20 bytes (5)
    Differentiated Services Field: 0x00 (DSCP: CS0, ECN: Not-ECT)
    Total Length: 328
    Identification: 0xf17b (61819)
    Flags: 0x00
    ...0 0000 0000 0000 = Fragment Offset: 0
    Time to Live: 64
    Protocol: UDP (17)
    Header Checksum: 0x882a [validation disabled]
    [Header checksum status: Unverified]
    Source Address: 0.0.0.0
    Destination Address: 255.255.255.255
User Datagram Protocol, Src Port: 68, Dst Port: 67
    Source Port: 68
    Destination Port: 67
    Length: 308
    Checksum: 0xd300 [unverified]
    [Checksum Status: Unverified]
    [Stream index: 41]
    [Timestamps]
    UDP payload (300 bytes)
Dynamic Host Configuration Protocol (Request)
    Message type: Boot Request (1)
    Hardware type: Ethernet (0x01)
    Hardware address length: 6
    Hops: 0
    Transaction ID: 0xd1d1a37e
    Seconds elapsed: 0
    Bootp flags: 0x0000 (Unicast)
    Client IP address: 0.0.0.0
    Your (client) IP address: 0.0.0.0
    Next server IP address: 0.0.0.0
    Relay agent IP address: 0.0.0.0
    Client MAC address: PcsCompu_75:2d:0b (08:00:27:75:2d:0b)
    Client hardware address padding: 00000000000000000000
    Server host name not given
    Boot file name not given
    Magic cookie: DHCP
    Option: (50) Requested IP Address (140.233.103.98)
```

```
    Option: (53) DHCP Message Type (Request)
    Option: (55) Parameter Request List
    Option: (57) Maximum DHCP Message Size
    Option: (61) Client identifier
    Option: (145) Forcerenew Nonce Capable
    Option: (255) End
    Padding: 0000
```

Wireshark Packet details

and for the DHCP ACK, we get:

```
Frame 698: 371 bytes on wire (2968 bits), 371 bytes captured (2968 bits)
   on interface enp0s3, id 0
Ethernet II, Src: PaloAlto_00:01:12 (00:1b:17:00:01:12), Dst: Broadcast (
   ff:ff:ff:ff:ff:ff)
Internet Protocol Version 4, Src: 140.233.103.1, Dst: 255.255.255.255
    0100 .... = Version: 4
    .... 0101 = Header Length: 20 bytes (5)
    Differentiated Services Field: 0x00 (DSCP: CS0, ECN: Not-ECT)
    Total Length: 357
    Identification: 0x0100 (256)
    Flags: 0x00
    ...0 0000 0000 0000 = Fragment Offset: 0
    Time to Live: 127
    Protocol: UDP (17)
    Header Checksum: 0x459e [validation disabled]
    [Header checksum status: Unverified]
    Source Address: 140.233.103.1
    Destination Address: 255.255.255.255
User Datagram Protocol, Src Port: 67, Dst Port: 68
    Source Port: 67
    Destination Port: 68
    Length: 337
    Checksum: 0x973f [unverified]
    [Checksum Status: Unverified]
    [Stream index: 42]
    [Timestamps]
    UDP payload (329 bytes)
Dynamic Host Configuration Protocol (ACK)
    Message type: Boot Reply (2)
    Hardware type: Ethernet (0x01)
    Hardware address length: 6
    Hops: 0
    Transaction ID: 0xd1d1a37e
    Seconds elapsed: 0
    Bootp flags: 0x8000, Broadcast flag (Broadcast)
    Client IP address: 0.0.0.0
    Your (client) IP address: 140.233.103.98
    Next server IP address: 0.0.0.0
    Relay agent IP address: 140.233.103.1
    Client MAC address: PcsCompu_75:2d:0b (08:00:27:75:2d:0b)
```

```
    Client hardware address padding: 00000000000000000000
    Server host name not given
    Boot file name not given
    Magic cookie: DHCP
    Option: (53) DHCP Message Type (ACK)
    Option: (54) DHCP Server Identifier (140.233.2.204)
    Option: (51) IP Address Lease Time
    Option: (1) Subnet Mask (255.255.255.0)
    Option: (3) Router
    Option: (6) Domain Name Server
    Option: (12) Host Name
    Option: (15) Domain Name
    Option: (28) Broadcast Address (255.255.255.255)
    Option: (255) End
```

Wireshark Packet Details

We observe that the UDP (User Datagram Protocol) protocol contains within it the DHCP (Dynamic Host Configuration Protocol) protocol. Under the DHCP segment, we can see that headers a DHCP protocol contains. The most notable is the Message. FOr the request message, it is set to Boot request and for the ack message, it is set to Boot reply. Under the ack message, we see that the your IP address is the virtual machine address and Client IP address is 0.0.0.0.

2 Implement ICMP traceroute using Scapy

- Because of the way Scapy accesses the network hardware, you'll need to run your script with superuser privilege (ie, using sudo). This is not busy-work: the intention is for you to implement something you know using a new tool that has much wider uses. We'll expand on our use of Scapy shortly

Here we can find documentation on using Scapy: https://scapy.readthedocs.io/en/latest/usage.html.

Here is the program we wrote in scapy to perform ICMP taceroute:

```

>>> ping  = IP(ttl=(1,20), dst = "72.14.176.147")/ICMP(id=1000) # here we
    set up an IP packet with ttl between 1 and 20 and a destination of
    72.14.176.147 inside an ICMP packet with id = 1000. The id number is
    random and the command didn't work without it.
>>> ans, unans = sr(ping) # here we send the ping packets using sr and
    store the results to answered and unanswered
Begin emission:
Finished sending 20 packets.
..******^C
Received 8 packets, got 6 answers, remaining 14 packets
```

```
>>> for snd,rcv in ans:
...     print(snd.ttl, rcv.src)
...
1 140.233.9.254
2 140.233.160.3
3 72.14.176.147
4 72.14.176.147
5 72.14.176.147
6 72.14.176.147
```

As we can see, we get the same jump points we did in the previous lab.

3 Basic communication

- Find another person to work with. Both using netcat, one set up a TCP listener and the other attempt to connect. Observe the packets flowing in Wireshark or tcpdump. Send data, watch the data fly past. Use input redirection on the sender and output redirection on the receiver to send a binary (ie, non-text) file.

We run the following command on the listening device:

```
nc -l -p 4000 localhost
```

Which tells net cat to listen (-l) on port 4000 for packets sent to localhost (this is the ip address of lo).

On the sending device, we run:

```
nc localhost 4000
```

which tells net cat to establish connecting with ip address localhost on port 4000.

On wireshark, we see the three-way handshake:

```
1 0.000000000 127.0.0.1 127.0.0.1 TCP 74  44162     4000 [SYN] Seq=0 Win
   =65495 Len=0 MSS=65495 SACK_PERM=1 TSval=2985491006 TSecr=0 WS=128
2 0.000014631 127.0.0.1 127.0.0.1 TCP 74  4000     44162 [SYN, ACK] Seq=0
   Ack=1 Win=65483 Len=0 MSS=65495 SACK_PERM=1 TSval=2985491006 TSecr
   =2985491006 WS=128
3 0.000027706 127.0.0.1 127.0.0.1 TCP 66  44162     4000 [ACK] Seq=1 Ack=1
    Win=65536 Len=0 TSval=2985491006 TSecr=2985491006
```

The first being SYN sent from 127.0.0.1 (local host) to local host. Then the SYN, ACK sent back. Finally an ACK.

When we send the text hello on the bash of the client, we get the followin on wireshark:

```
[majd ~]$ nc -l -p 4000 localhost
Hello
```

on the server side:

```
[@majd ~]$ nc localhost 4000
Hello
```

on wireshark after the handshake:

```
4 336.740426717 127.0.0.1 127.0.0.1 TCP 72  4000     44162 [PSH, ACK] Seq
   =1 Ack=1 Win=65536 Len=6 TSval=2985827747 TSecr=2985491006
5 336.740445552 127.0.0.1 127.0.0.1 TCP 66  44162     4000 [ACK] Seq=1 Ack
   =7 Win=65536 Len=0 TSval=2985827747 TSecr=2985827747
```

We see the client (here the client and server are the same (localhost) but it is the case when sending to another ip address) send a PSH, ACK packet with "hello" as data, then the server acknowledging the receive of the data by sending back an ACK to the client.

4 Less-basic communication

- With one party still using netcat, implement the other side using Scapy. Start by performing the TCP 3-way handshake according to the rules of the protocol (you can use the built-in Scapy functionality for this). Once you've got that working, start playing around: cause the contents of the headers to deviate in some small way, observe this deviation in Wireshark, observe the response sent by the target.

  Part of the problem with using scapy to inject traffic directly onto the network is that the kernel on your machine has no idea what to do with any responses, and might itself respond... inconveniently (ie, with a RST packet, which the remote machine will interpret to mean that no such connection exists). To get around that, you can instruct the Linux kernel to decline to process any packets received from a particular host:

  ```
  $ sudo iptables -A INPUT --source <ip address> -j DROP
  ```

  I think this is a better solution than one which suppressed all outgoing TCP packets with the RST flag set, as sometimes these are useful/necessary.

  The kernel will forget about this particular configuration upon reboot, or you can actively remove it with this command:

  ```
  $ sudo iptables -D INPUT --source <ip address> -j DROP
  ```

  To verify that this worked, run this command and make sure there are no rules listed under the INPUT chain:

  ```
  $ sudo iptables -nvL
  ```

Here I will use Pete's machine to report my findings. It will be updated later in case there was a difference when doing with my partner.

We first prepare a Sync request and send it. By defaults, if no flags are provided to TCP(), scapy will fill the flags with Sync flags:

```
>>> sync = IP(dst="72.14.176.147")/TCP(dport=4000)
>>> ack = IP(dst="72.14.176.147")/TCP(dport=4000)
>>> sr(sync)
```

On wireshark, we see:

```
1103  861.700734441 140.233.186.95  72.14.176.147 TCP 54  [TCP
   Retransmission] [TCP Port numbers reused] 20     4000 [SYN] Seq=0 Win
   =8192 Len=0
1104  861.751985926 72.14.176.147 140.233.186.95  TCP 60  4000     20 [RST
   , ACK] Seq=1 Ack=1 Win=0 Len=0
```

We sent a sync message and received RST, ACk. This is not what is expected. This probably happened becayse there is no process listening on port 4000 on Pete's machine. Using the same code on my partner's computer, we received a SYN, ACK message. Success!

We should respond by an ACK to the SYN, ACK message. We have to make sure that the ack field in the SYN, ACK message matches with the 1 + squence number we send in our SYN message, increment our squence number by 1 to send in the ACK message, and add 1 to the sequence number sent by the SYN,ACK message. We wrote a python code to implement the handshake. When tested with my partner, we were able to perform the handshake with no issue!

```
# hanshake.py

from scapy.all import *
import sys



def sync_packet(ip, port):
    sync = IP(dst = ip)/TCP(flags = 'S', dport=port)
    return sync

def send(packet):
    return sr1(packet)

def ack_packet(ip, port, my_seq, their_seq):
    ack = IP(dst = ip)/TCP(flags='A', dport = port, seq = my_seq + 1, ack
   = their_seq + 1)
    return ack

def acknowledge(old_packet, ans):
    ip = old_packet.dst
    port = int(old_packet.dport)
    seq = int(ans.seq)
    ack = int(ans.ack)
    my_seq = int(old_packet.seq)

    if ack == my_seq + 1:
        ack = ack_packet(ip, port, my_seq, seq)
```

```
28
29      return ack
30
31
32
33  ip = "127.0.0.1"#input("please enter ip destination address:")
34  port = 4000#int(input("please enter the port number"))
35
36  sync = sync_packet(ip, port)
37  ans = send(sync)
38
39  ack = acknowledge(sync, ans)
40  send(ack)
```