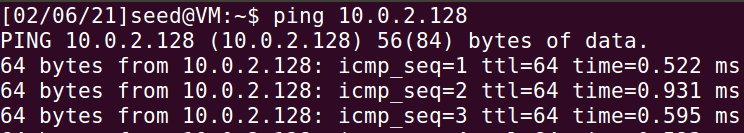
# 50.020 Network Security Lab 1

**Exercise 1: Packet Sniffing and Spoofing Lab**

**Task 1.1A**

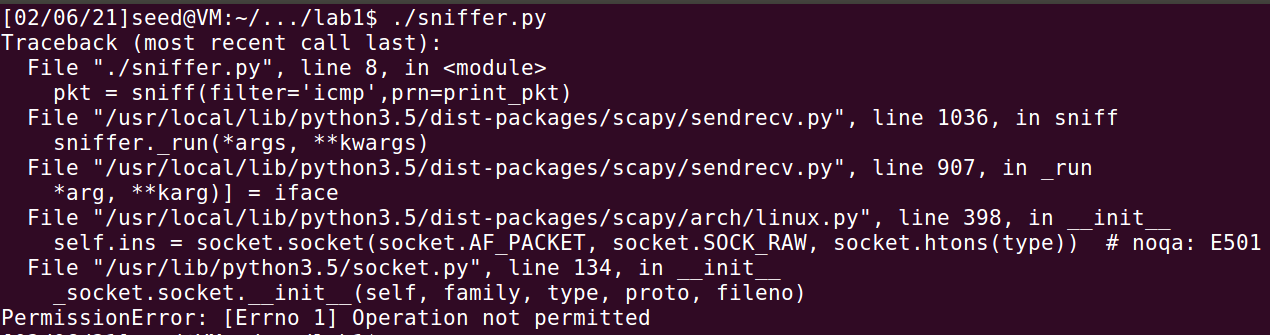
The program runs well with root privilege. Machine A with IP address 10.0.2.128 is set up with the running *sniffer.py* program and Machine B with IP address 10.0.2.129 is used to ping A’s IP address.







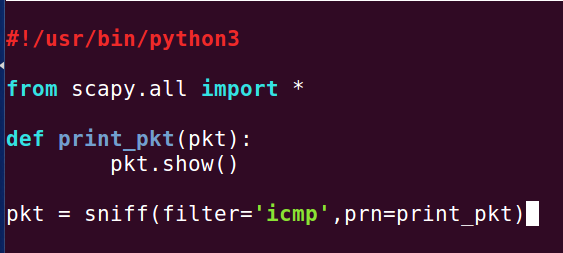
Without root privilege, the program doesn’t run because privilege is required for spoofing packets.



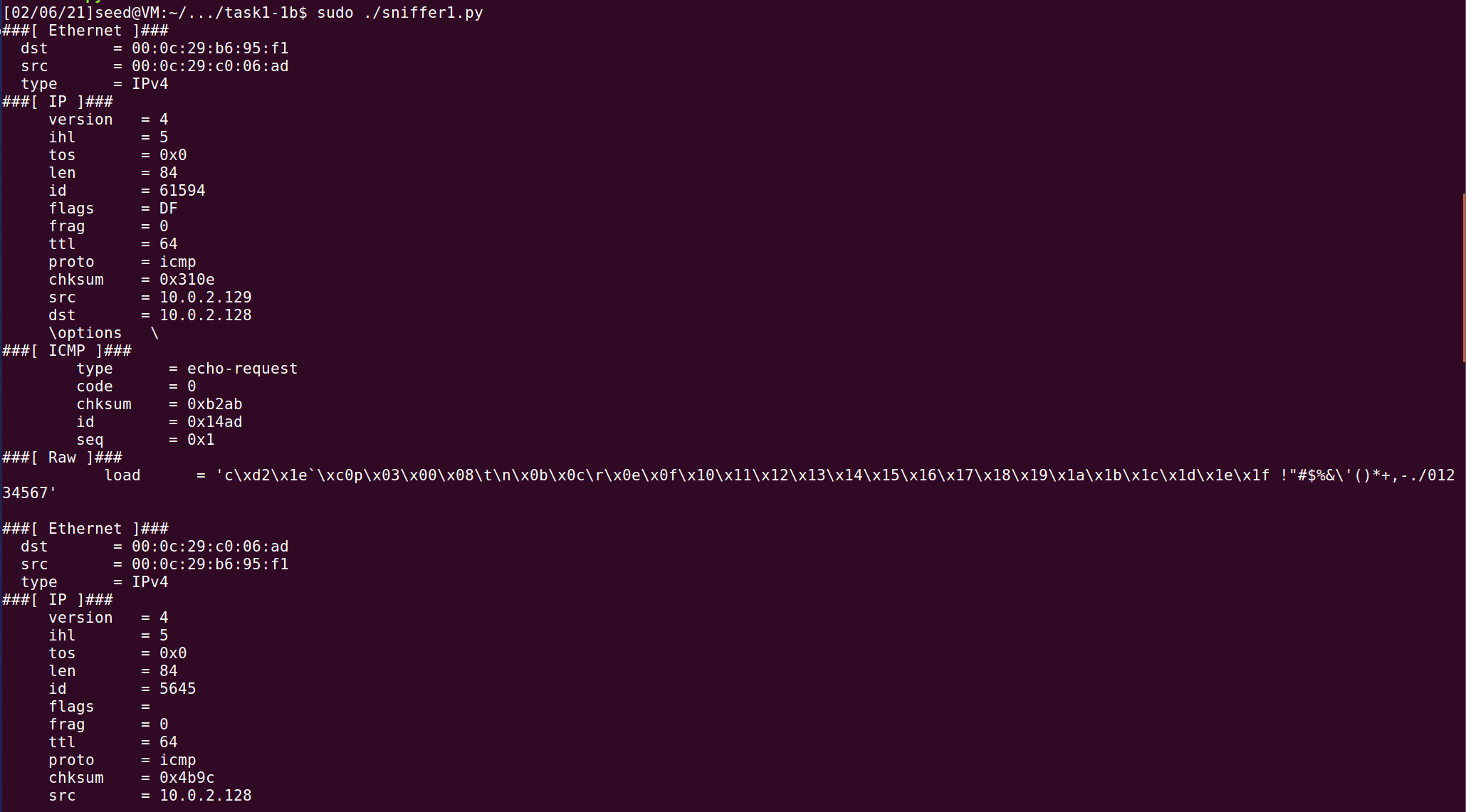
**Task 1.1B**

Capturing only the ICMP packet

The following code is used to filter only ICMP packets:

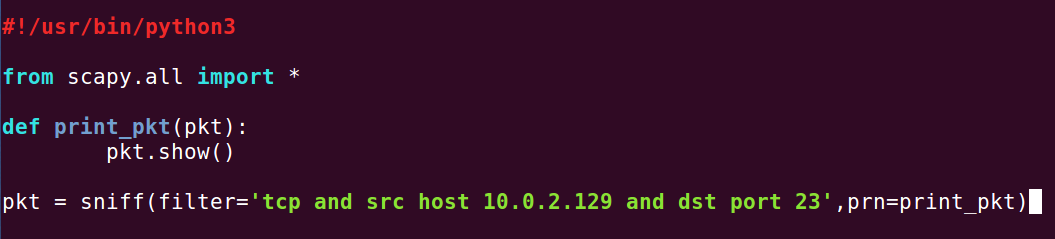


Using B to ping A like in task 1.1a, we get the IMCP packets when the sniffing code is run as shown:

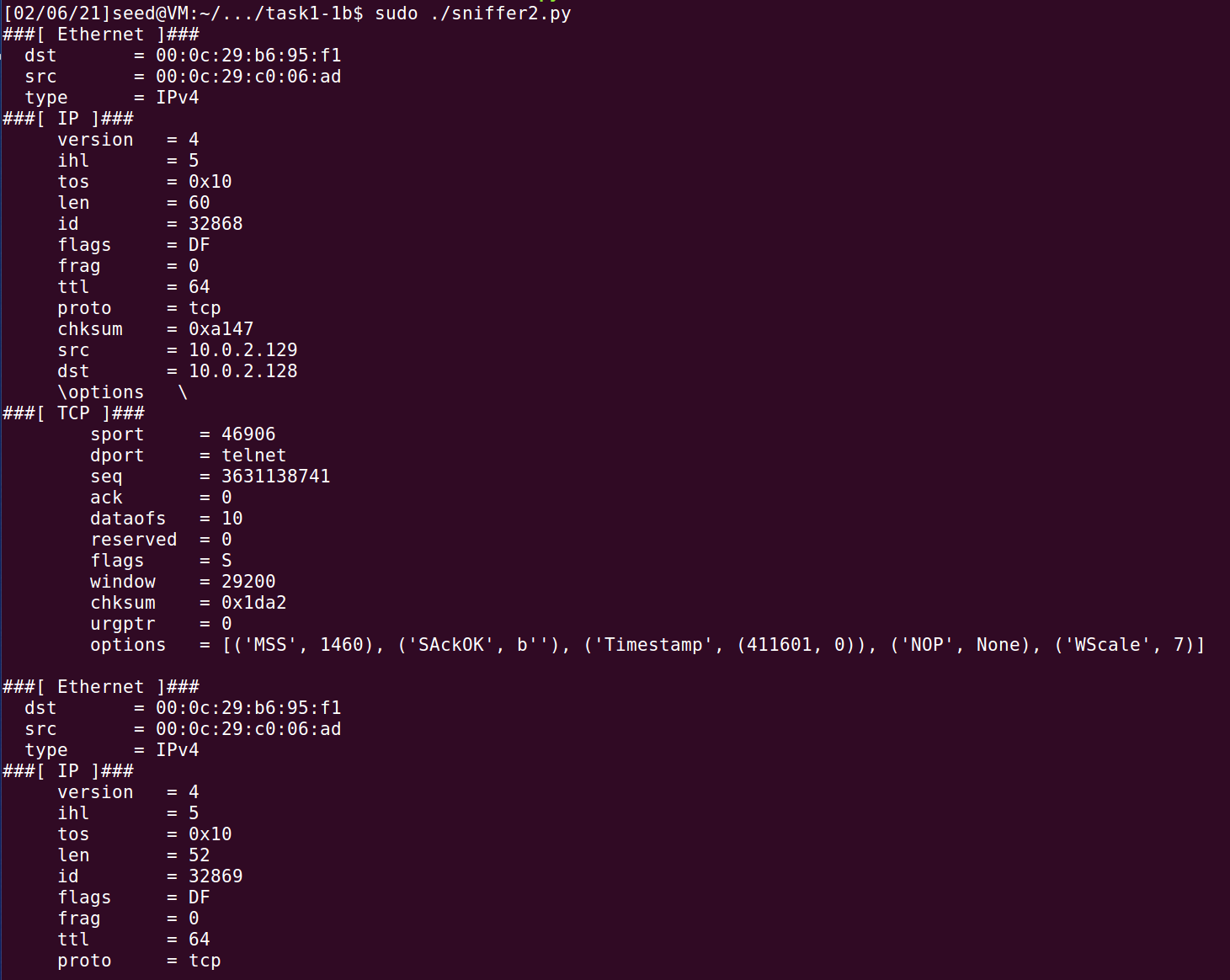


Capturing any TCP packet that comes from a particular IP and with a destination port number 23

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:



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:

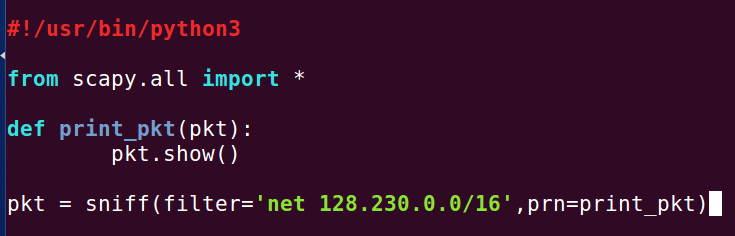


but does not capture ICMP packets that result from B pinging A.

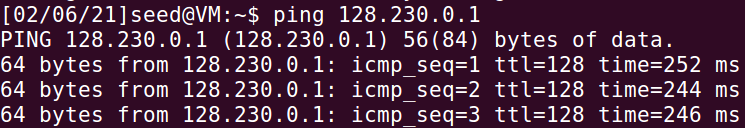
The sniffer program also does not capture TCP packets when another machine (Machine C, IP address 10.0.2.130) makes a telnet connection to Machine A.

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:

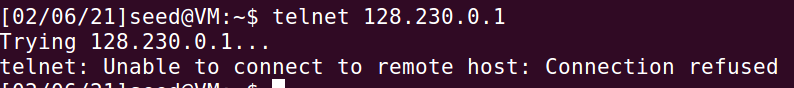


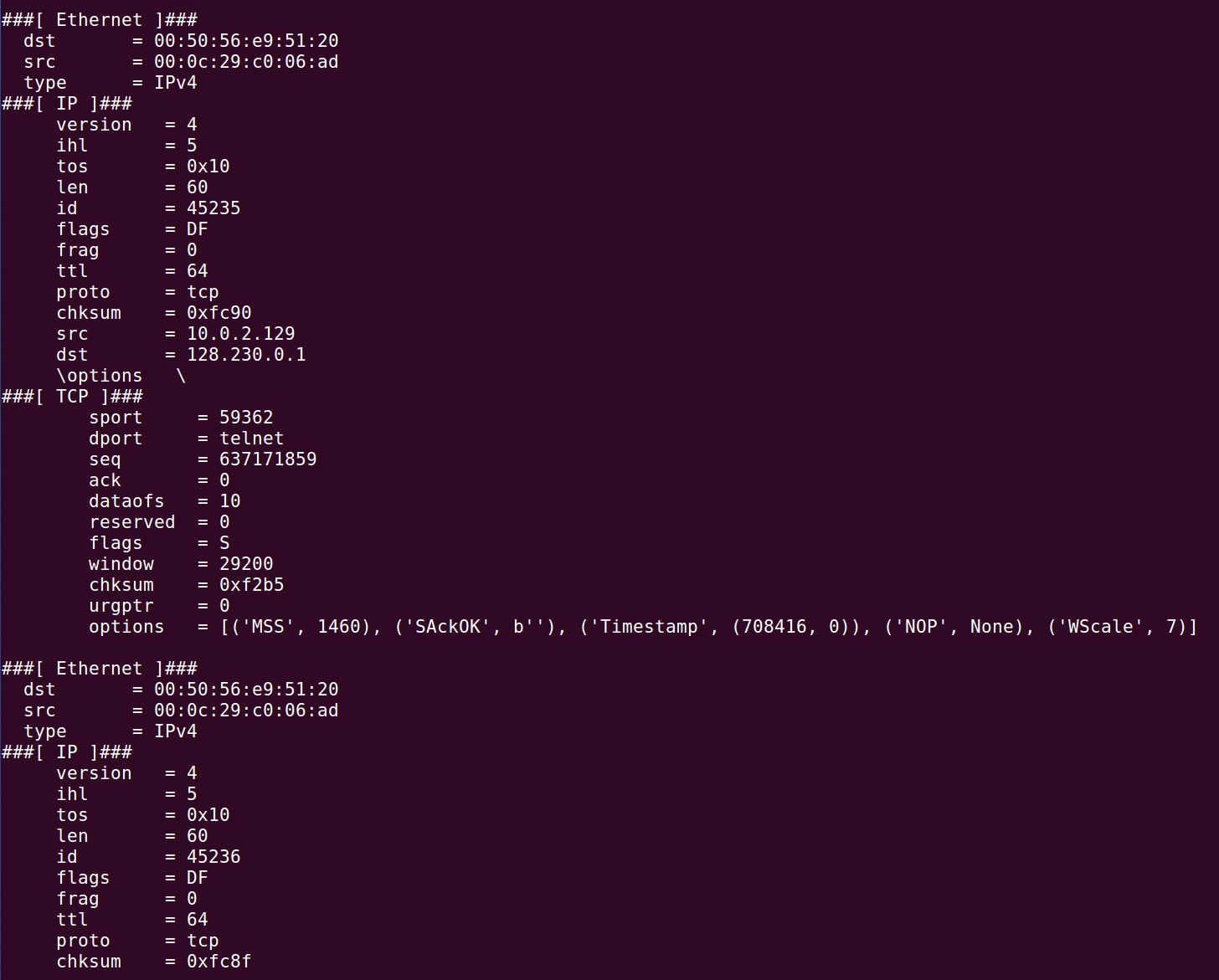
When the sniffer program is run on machine A, it only captures connections to IP addresses in the specified subnet. For example, when Machine B tries to ping 128.230.0.1, Machine A captures the ICMP packets:





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

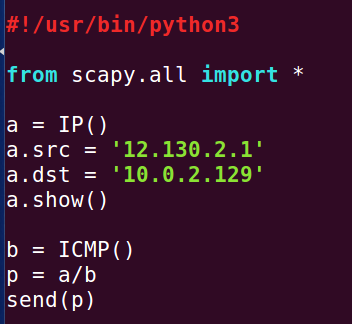




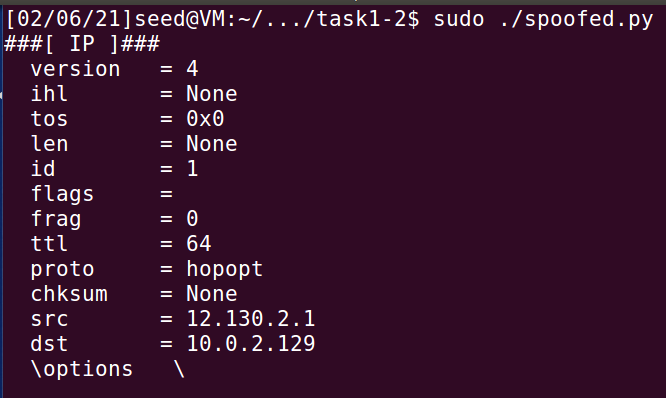
The sniffer program does not respond to any other packets from Machine A or B directed to any other IP address outside the subnet range.

**Task 1.2**

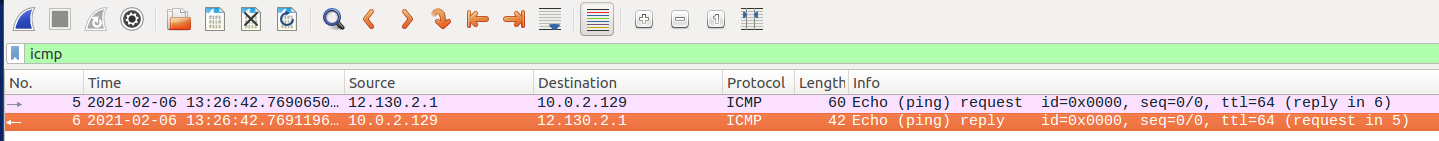
The following code is used to spoof an ICMP echo request packet with an arbitrary source IP address, in this case 12.130.2.1:



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):



Checking the packet capture using Wireshark on Machine B:

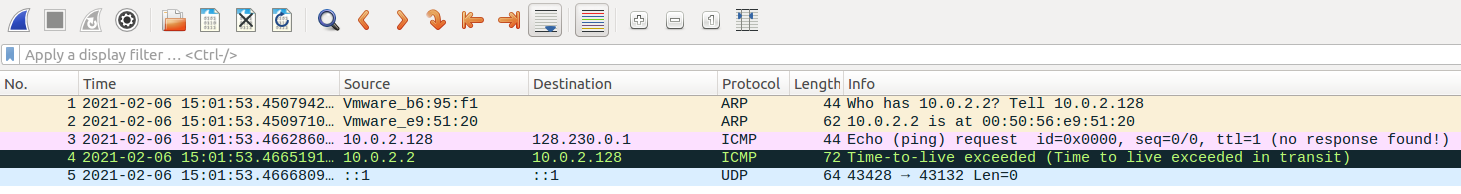


The request has been accepted by the receiver because the echo reply packet is sent to the spoofed IP address as shown in the screenshot, which is 12.130.2.1.

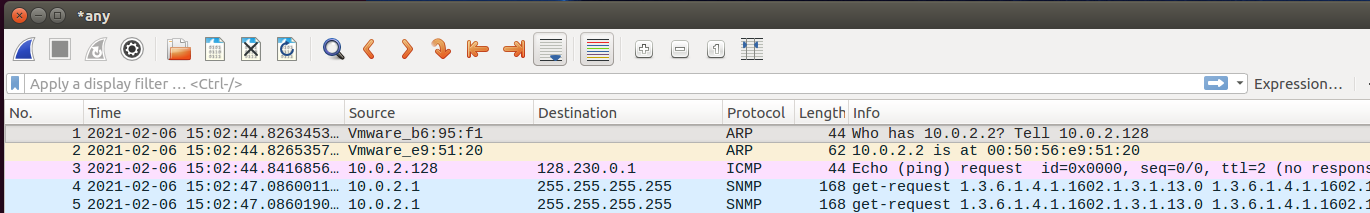
**Task 1.3**

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:



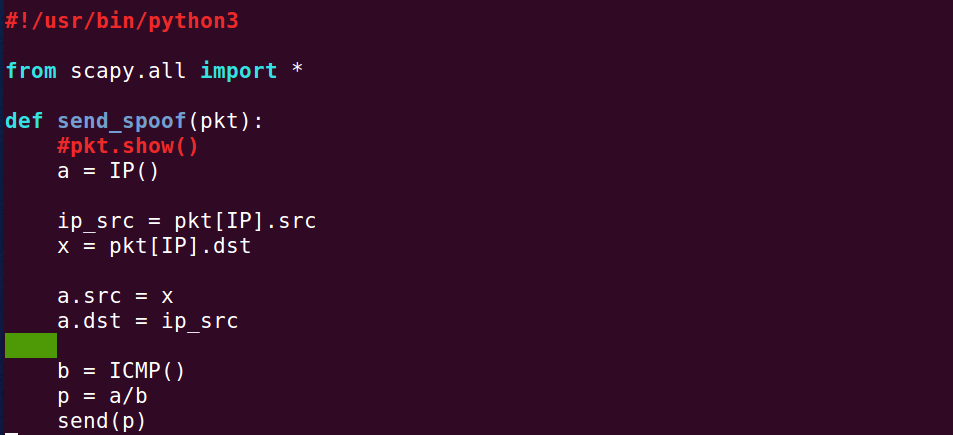
Our packet reaches the destination when the TTL field set to 2:



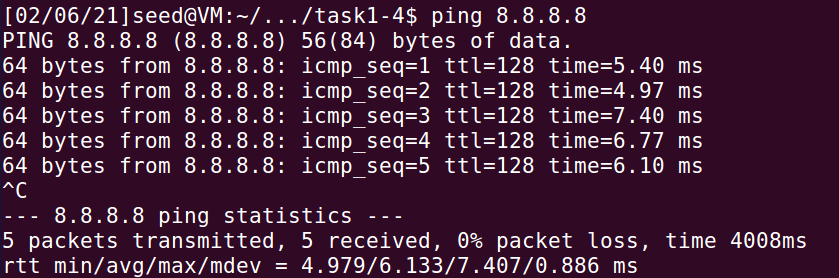
**Task 1.4**

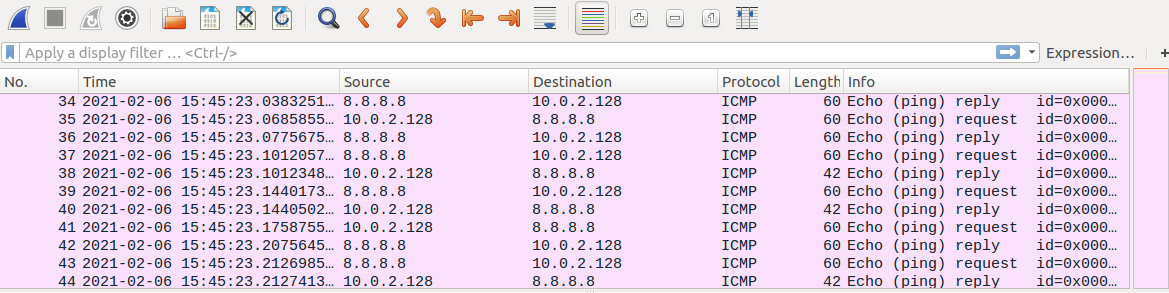
VM A has IP address 10.0.2.128 and VM B has IP address 10.0.2.129.

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:

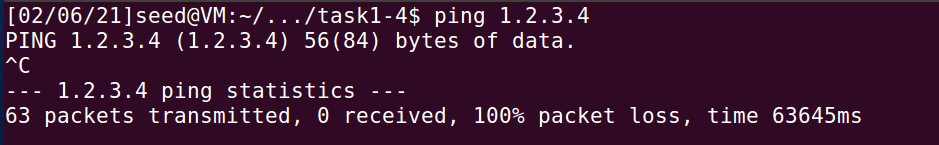


While the sniff-and-then-spoof program is running on VM B, VM A is used to ping a random IP address, which is 8.8.8.8 in this case. VM B sniffs its packets and returns spoofed packets, disguising as the host with IP 8.8.8.8 itself. The real host also responds with ICMP packets, as the host is actually 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.

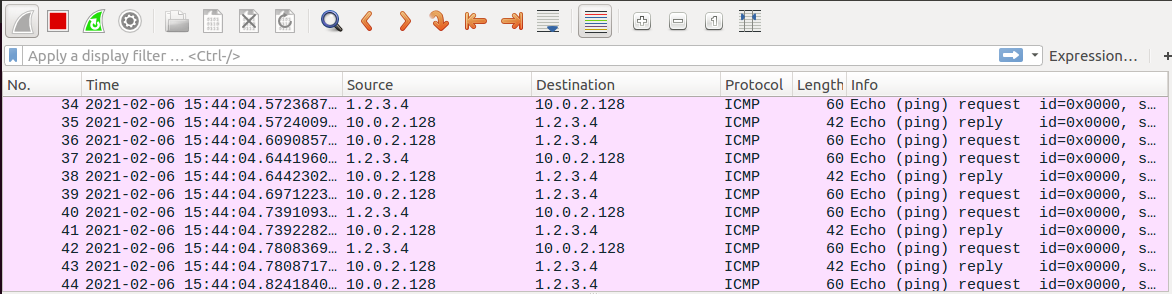




To show that VM A will receive an ICMP echo request regardless of whether X is actually alive, VM A is used to ping a host that does not actually accept the echo request ICMP packets, which is a host with IP address 1.2.3.4 in this case. This is shown as the ping statistics reflect 100% packet loss.



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.



**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.

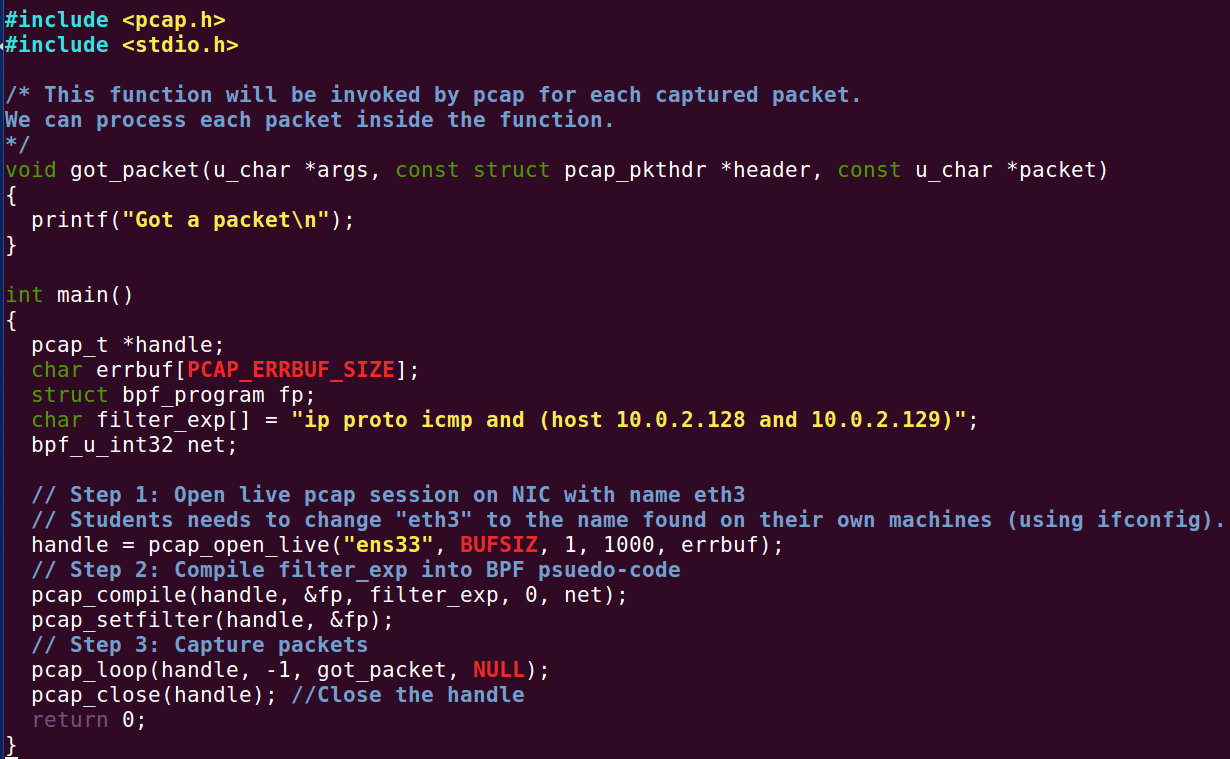
Q3: Demonstrate the difference when promiscuous mode is on and off in your sniffer program.

…(complete if there is time)

**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:



…(complete if there is more time)

**Exercise 2: ARP Cache Poisoning Attack Lab**

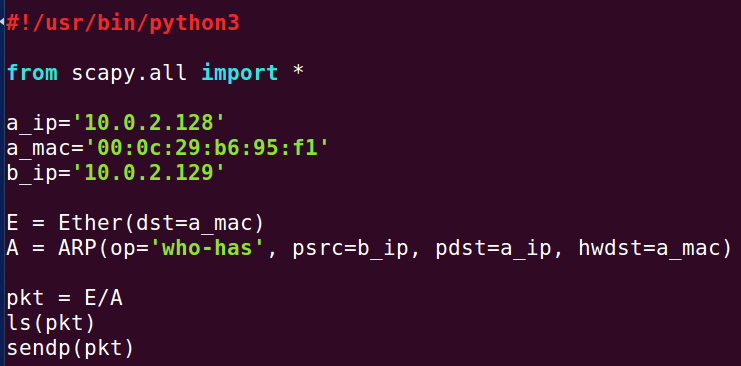
IP address of A: 10.0.2.128, MAC address of A: 00:0c:29:b6:95:f1

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

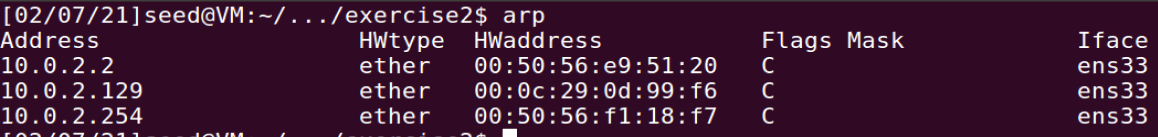
IP address of M: 10.0.2.130, MAC address of M: 00:0c:29:0d:99:f6

**Task 1A**

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



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

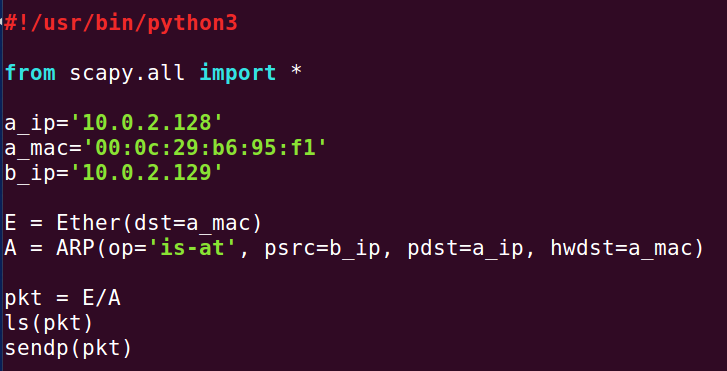


M’s MAC address (00:0c:29:0d:99:f6) is indeed mapped to B’s IP address (10.0.2.129) in A’s ARP cache, as spoofed.

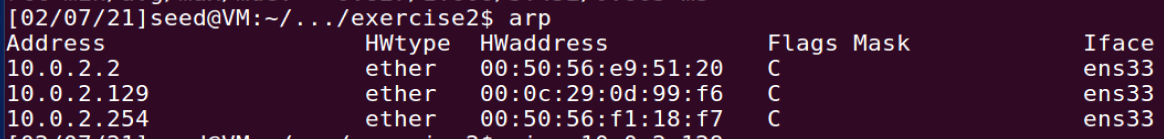
Note that the destination MAC address (A’s MAC address) has to be specified in the Ether() constructor as well, otherwise M will automatically send out an ARP broadcast message and list M’s own IP address as the source IP.

**Task 1B**

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

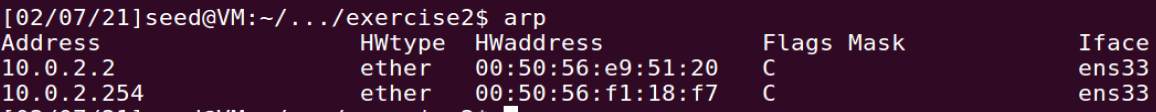


If the B’s IP address entry was previously in Host A’s ARP cache, Host A receives this packet and updates its ARP cache accordingly:



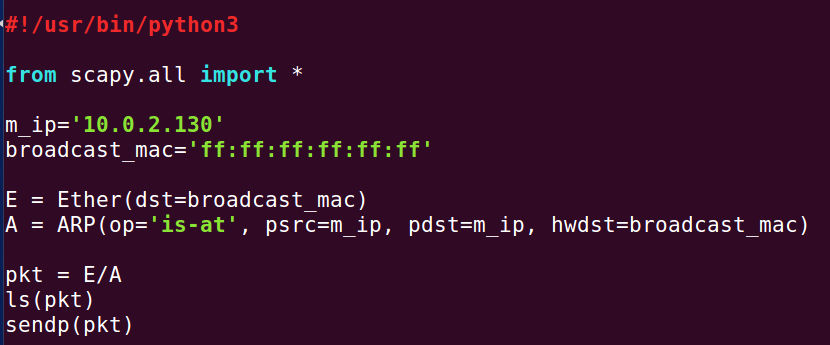
M’s MAC address (00:0c:29:0d:99:f6) is indeed mapped to B’s IP address (10.0.2.129) in A’s ARP cache, as spoofed.

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:

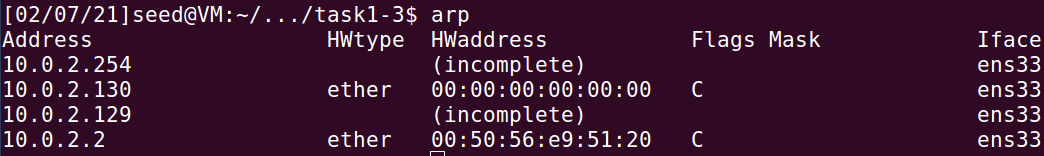


**Task 1C**

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

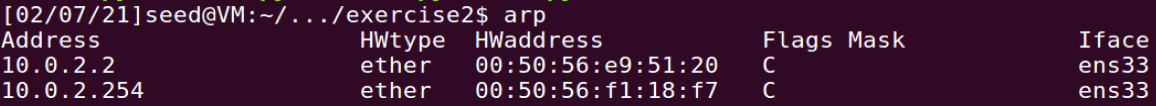


If the M’s IP address entry was previously in Host A’s ARP cache, Host A receives this packet and updates its ARP cache accordingly:



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.

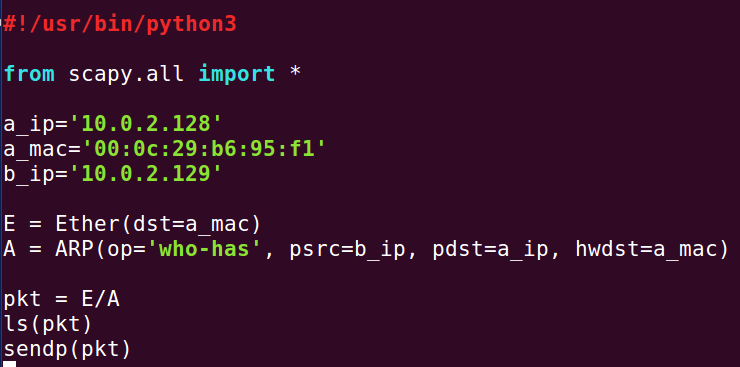
However, if M’s IP address entry is not in Host A’s ARP cache to begin with, Host A, upon receiving the packet, does NOT update its ARP cache:



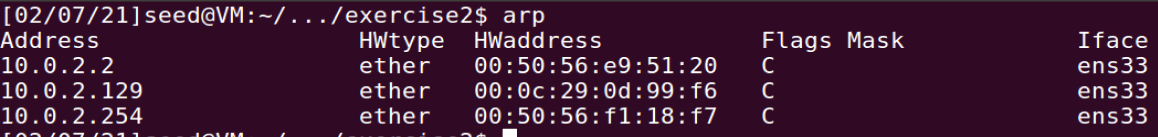
**Task 2**

Step 1: Launch the ARP cache poisoning attack

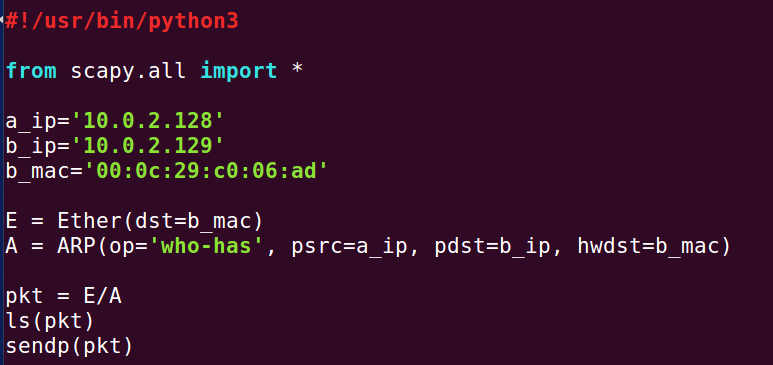
The following code is used to send a spoofed packet to A such that in A’s ARP cache, B’s IP address maps to M’s MAC address:



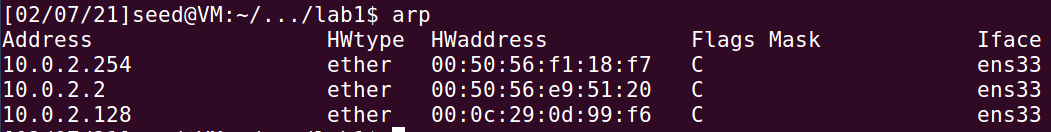
A’s ARP cache is updated accordingly:



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:

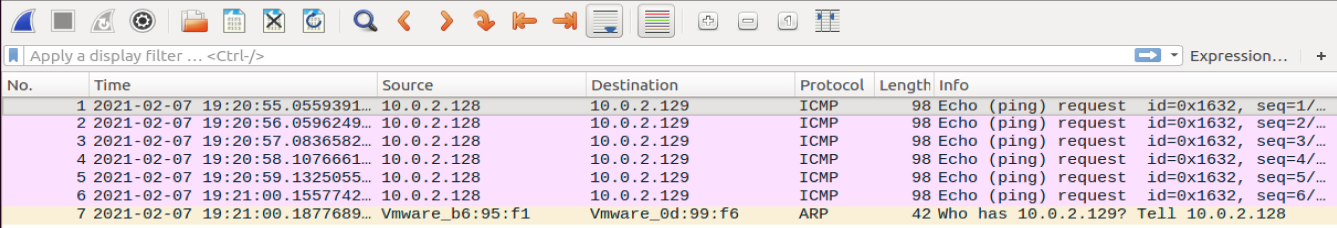
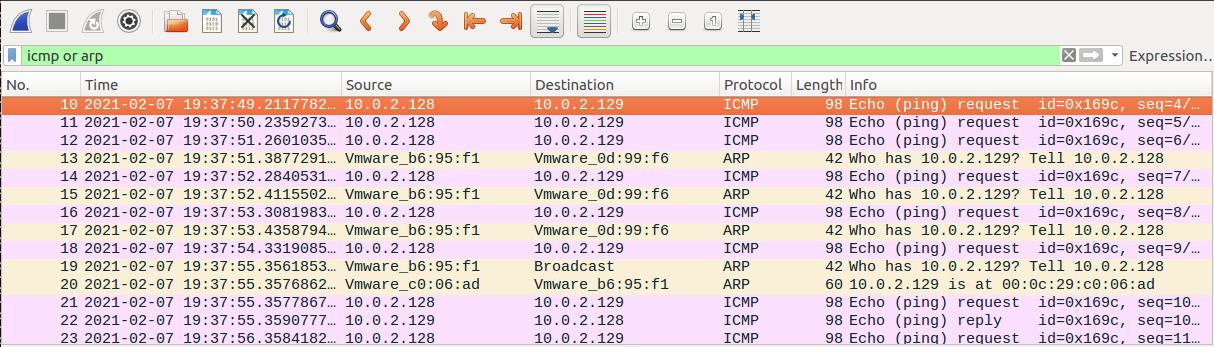


B’s ARP cache is updated accordingly:



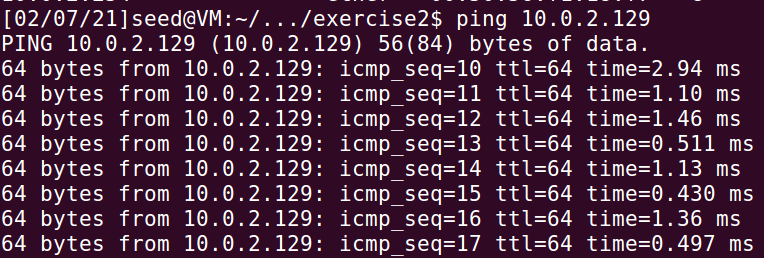
Step 2: Testing

Wireshark screenshot of Host A pinging Host B:



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.

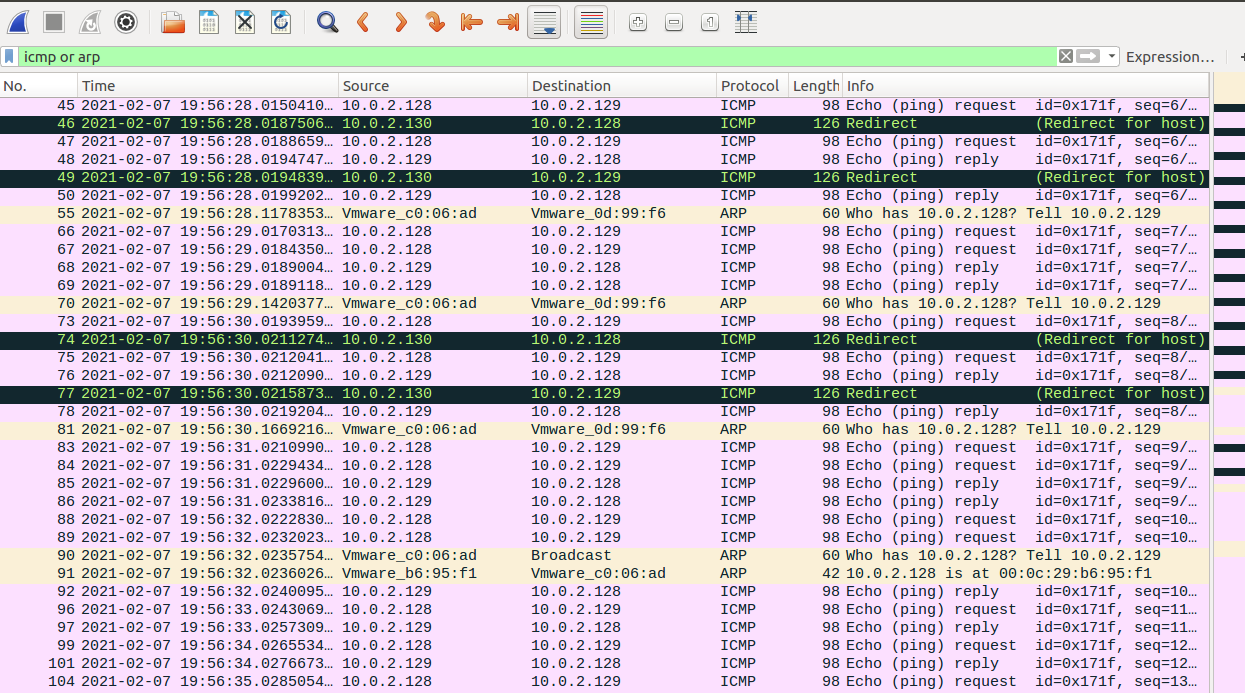
Host A eventually sends an ARP broadcast ARP request to resolve B’s IP address into its MAC address. Host B responds accordingly, so A’s ARP cache is updated to have B’s MAC address match B’s IP address. As this is resolved, the ping resumes its normal behaviour and echo request and reply packets are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



Host B pinging Host A would yield similar results.

Step 3: Turn on IP forwarding

Wireshark screenshot of Host A pinging Host B:

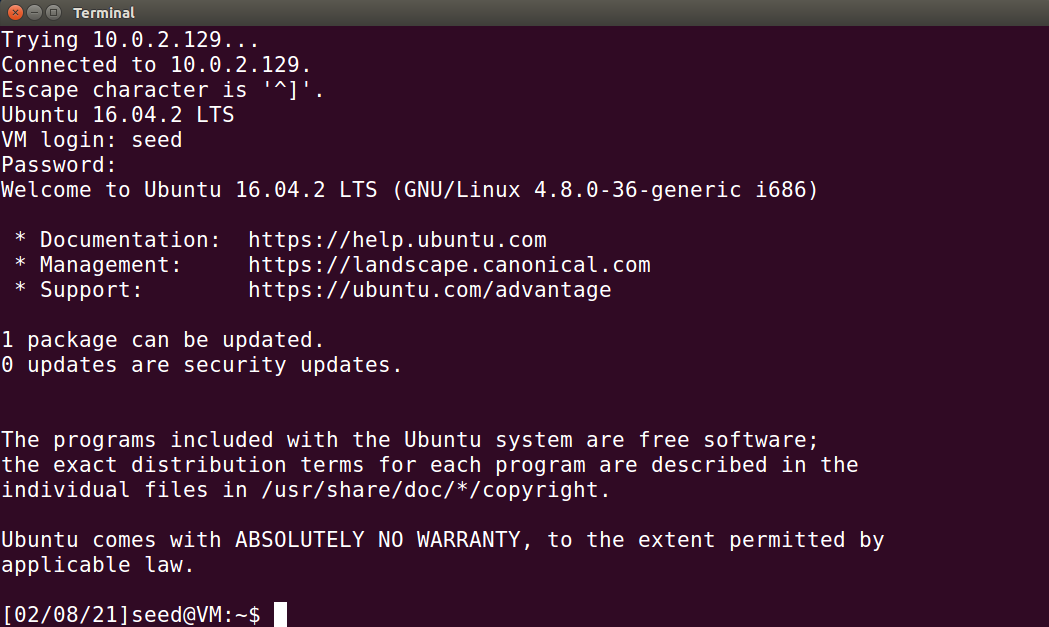


In this case, ICMP echo request packets sent from Host A to Host M (destination IP address is Host B’s IP, but destination MAC address is M’s MAC address as A’s ARP cache is poisoned) would be redirected by Host M to Host B (due to IP forwarding). Host B would then craft ICMP echo reply packets to Host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

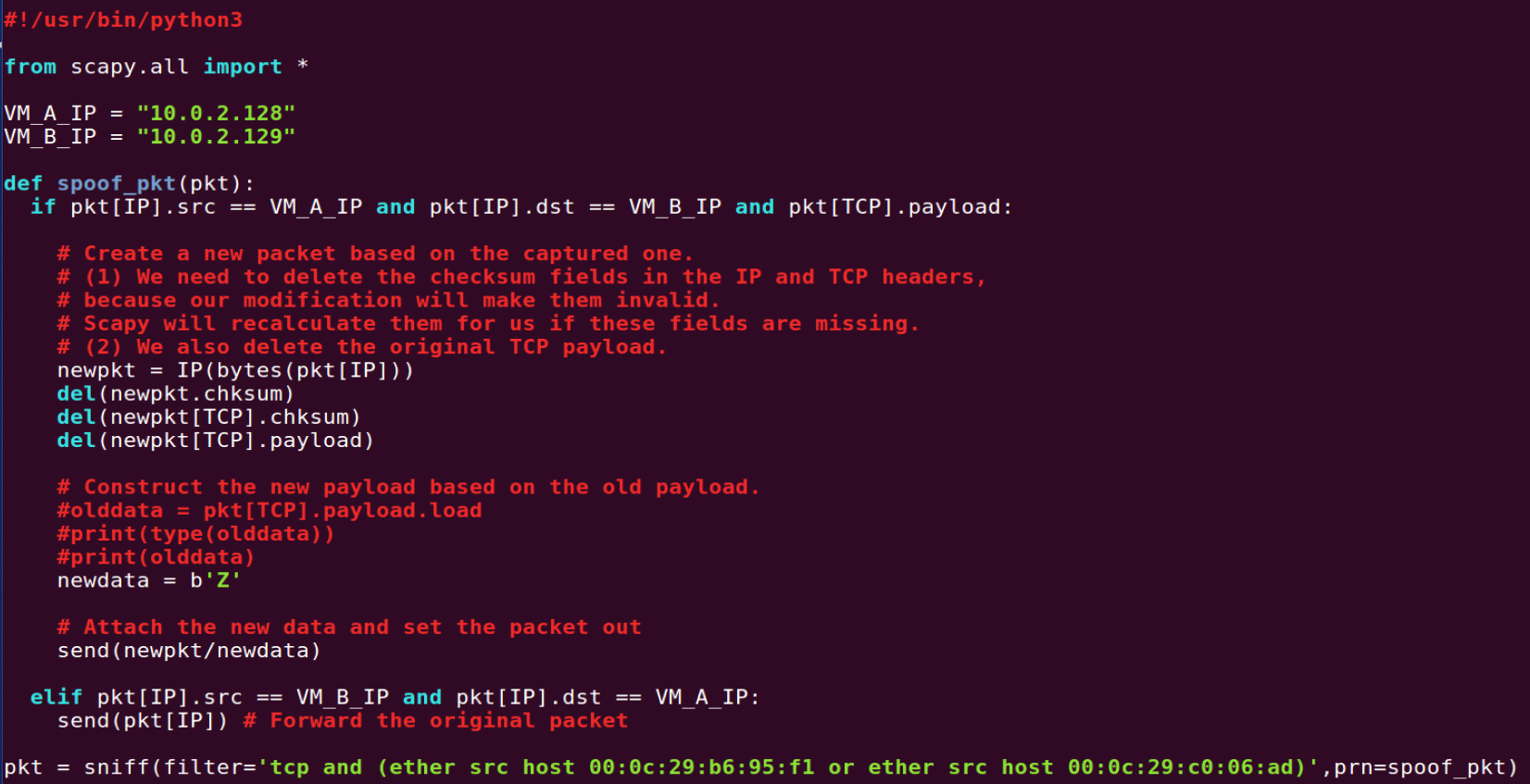
Host A eventually sends an ARP broadcast ARP request to resolve B’s IP address into its MAC address, and B responds accordingly. The ARP caches of A and B are both updated accordingly and the pinging happens directly between the two hosts. The redirection of packets no longer happen as the packets do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

Step 4: Launch the MITM attack

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



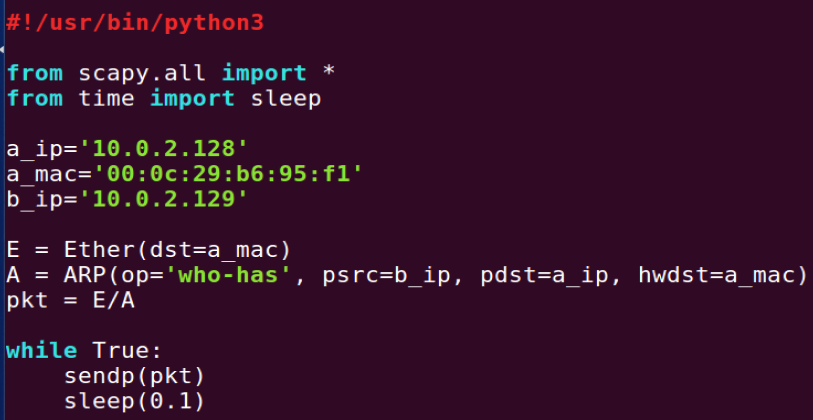
The following code is used to sniff and spoof the Telnet packets such that the character Z will be displayed no matter which character is entered in Host A:



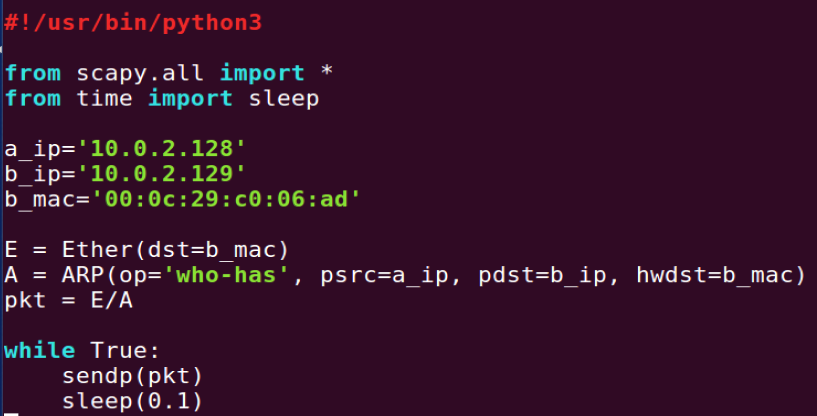
The filter is written such that the IP address will not confuse the program in reading packets sent by its own host (M). If the filter is not written properly, the packet forwarding results in duplicated packets and the packets will multiply and the forwarding of one packet will continue forever.

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:

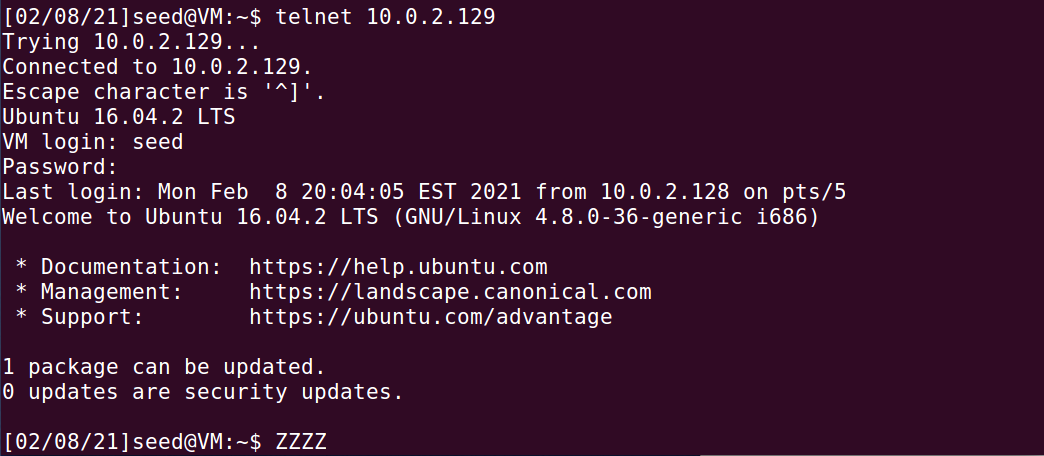
(Code executed to poison A’s cache)



(Code executed to poison B’s cache)

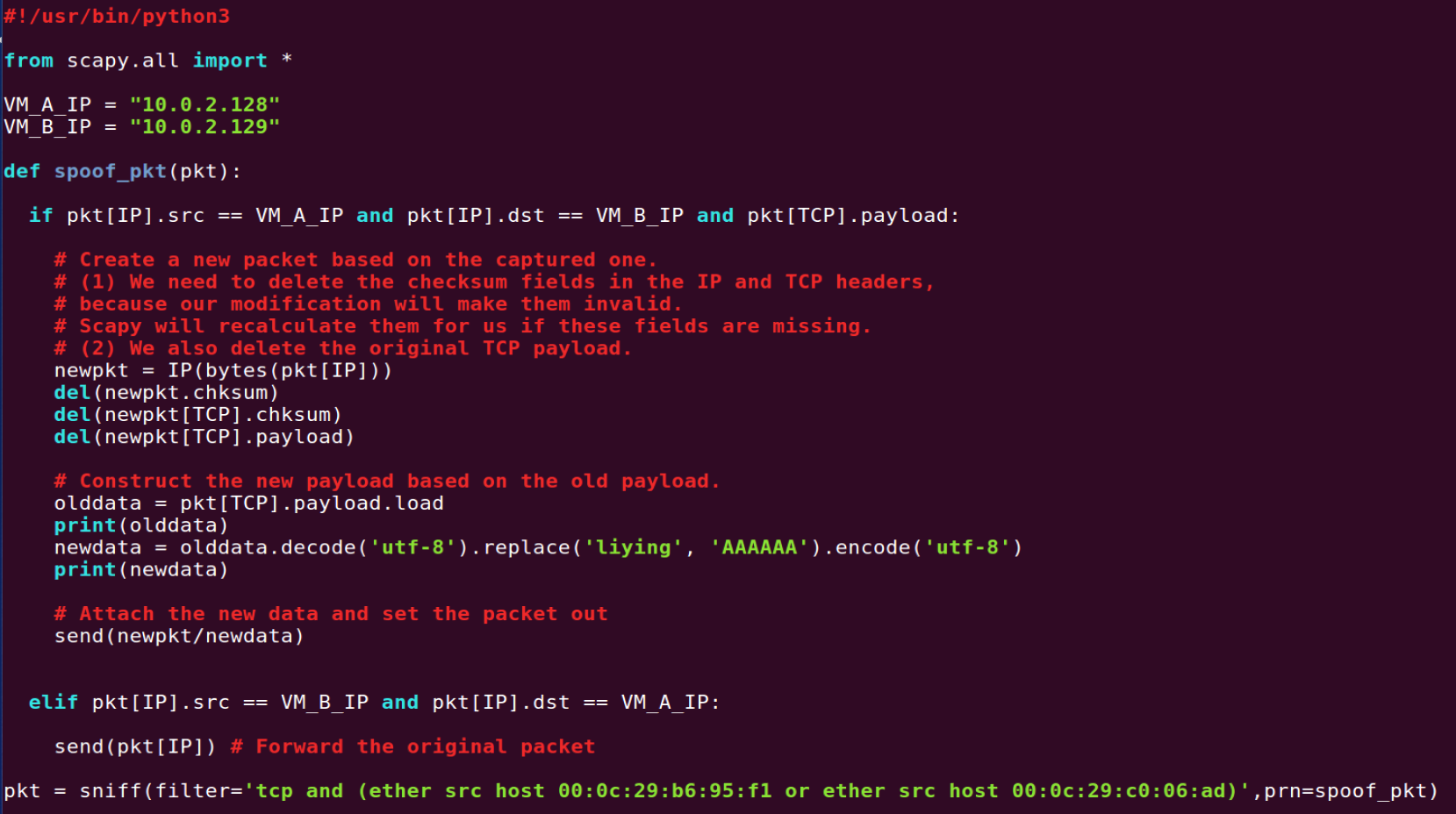


The newdata replaces the payload data with the character ‘Z’. The results of the successful spoofing is as shown:



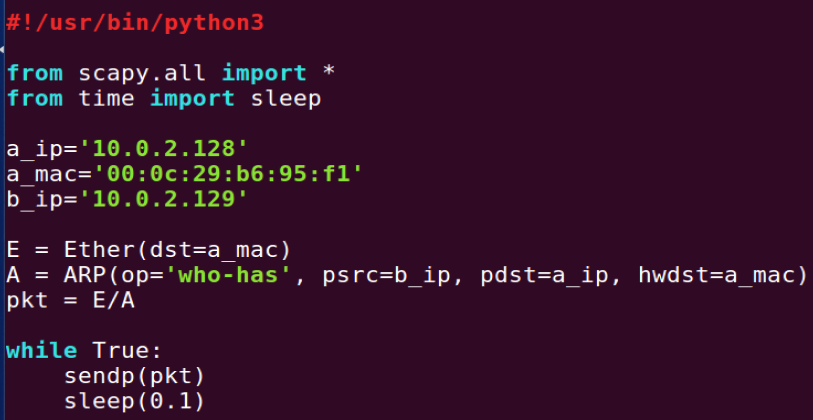
**Task 3**

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

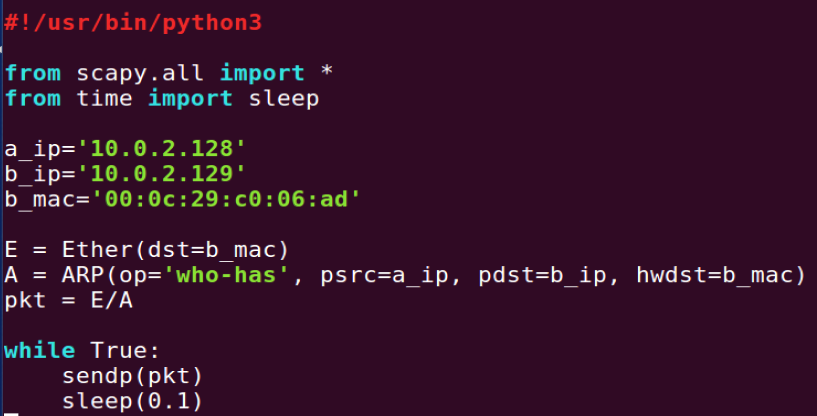


The ARP poisoning code is run separately and concurrently to ensure that the ARP caches of Host A and B remain spoofed. The ARP poisoning code is as shown:

(Code executed to poison A’s cache)

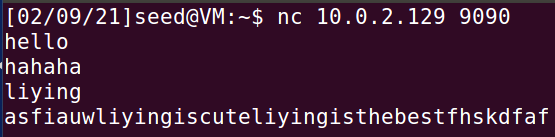


(Code executed to poison B’s cache)



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)



(Netcat display on Host B)

