“**Lab:11: Packet snifﬁng and spooﬁng.**

**Md Rony**

The sequence of the library calls that are essential for sniffer programs are as below:

Setting the device

Opening the device for sniffing

Filtering traffic

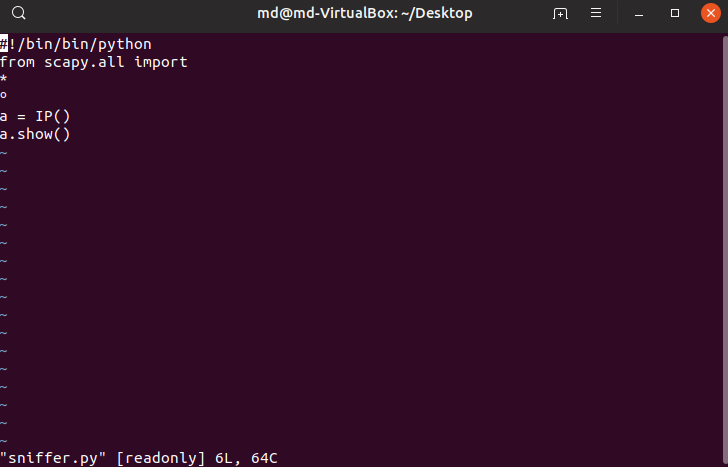
The actual sniffing

**Task 1: Writing Packet Sniffing Program**

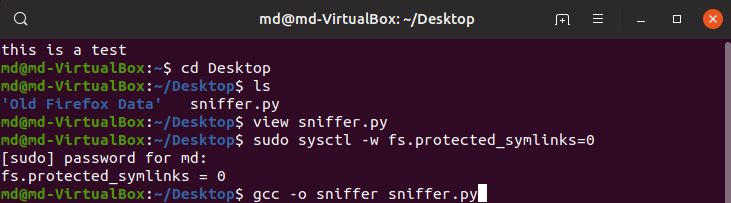


**Figure1**

**Task 1.a: Understanding Sniffex**

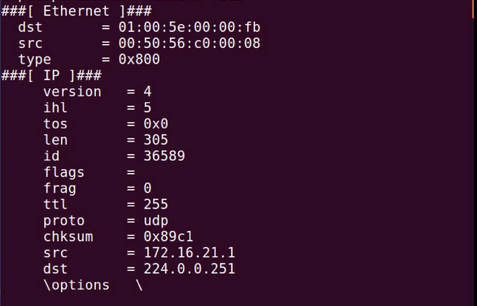
****

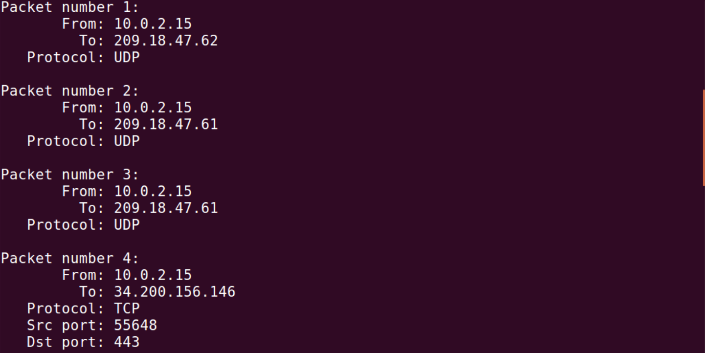
**figure 2**

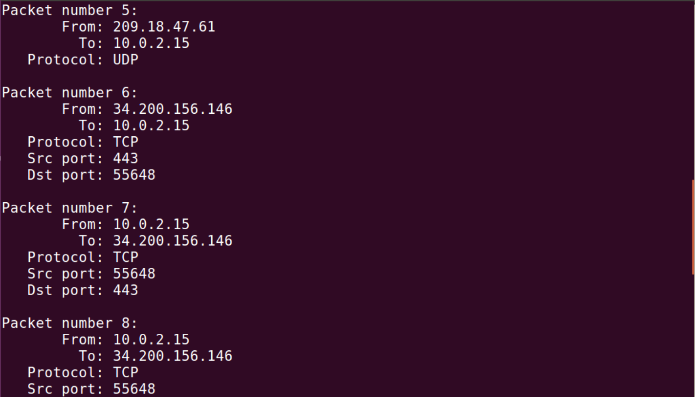
****

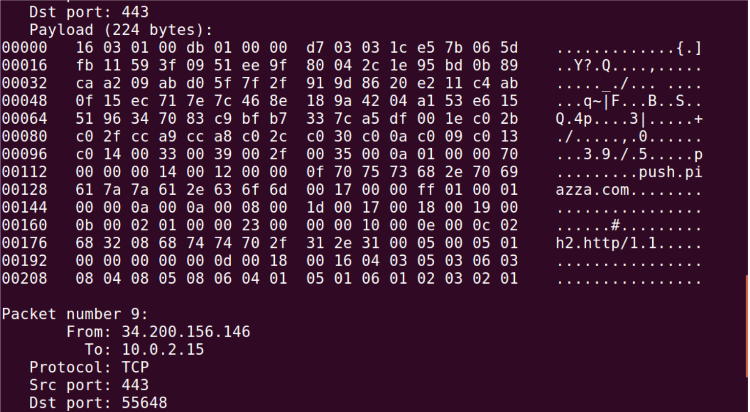
4.png

**Figure 3**

****

****

****

****

****

**Figure 4**

**Observation:** First, let’s try to run our sniffex code with root privilege. We successfully capture 10 packets after compilation.

**Problem 1**:

Here are the steps to the sequence of library calls essential for sniffer programs:

**1. Setting up Device**:

pcap sets the device on its own. If this fails, it saves the error message into errbuf.

pcap\_lookupdev(errbuf) can be used to find a device to sniff on.

**2. Opening the device for sniffing:**

pcap uses pcap\_open\_live() to open session on a device we will be sniffing on.

The format of the statement is as follows:

**pcap\_t \*pcap\_open\_live(char \*device, int snaplen, int promisc, int to\_ms, char \*ebuf)**

• char \*device: specifies the device we are sniffing on.

• snaplen: specifies max number of bytes to be captured by pcap.

• promisc: specifies if Promiscous mode is on or not.

• to\_ms: this value is non-zero as this is the read time out in milliseconds.

• char \*ebuf: stores error messages.

Note: Promiscous Mode is used to sniff all network traffic and not just the traffic to, from, or routed through a specific host.

**3. Filtering Traffic**:

We perform filtering using two functions in pcap library:

pcap\_compile() is used to compile the filter expression stored in a regular string.

pcap\_setfilter() is used to set the compiled filter to determine what the program sniffs.

Here’s the prototype for them:

**int pcap\_compile(pcap\_t \*p, struct bpf\_program \*fp, char \*str, int optimize, bpf\_u\_int32 netmask)**

• Pcap\_t \*p: specifies session handle.

• struct bpf\_program \*fp: specifies reference to the place we will store the compiled version of our filter.

• char \*str: specifies expression in a regular string format.

• int optimize: integer that decides if the expression should be "optimized" or not.

• bpf\_u\_int32 netmask: specifies the network mask of the network the filter applies to.

**int pcap\_setfilter(pcap\_t \*p, struct bpf\_program \*fp)**

• pcap\_t \*p: session handler.

• struct bpf\_program \*fp: specifies reference to the compiled version of the expression.

**4. Sniffing:**

**u\_char \*pcap\_next(pcap\_t \*p, struct pcap\_pkthdr \*h)** is used to capture a single packet at a time.

• pcap\_t \*p: session handler

• struct pcap\_pkthdr \*h: a pointer to a structure that holds general information about the packet

• The function returns a u\_char pointer to the packet that is described by this structure

**int pcap\_loop(pcap\_t \*p, int cnt, pcap\_handler callback, u\_char \*user)** is used to enter a loop that waits for n number of packets to be sniffed before being done.

• pcap\_t \*p: session handle

• int cnt: specifies how many packets it should sniff for before returning. Negative value means it should sniff until an error occurs.

• pcap\_handler callback: the name of the callback function

• u\_char \*user: useful in some applications, NULL for many situations.

**5. Close the Sniffing Session:**

pcap\_close() is used to close the sniffing session

4.png

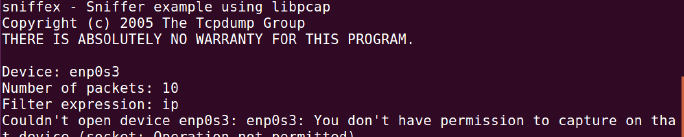
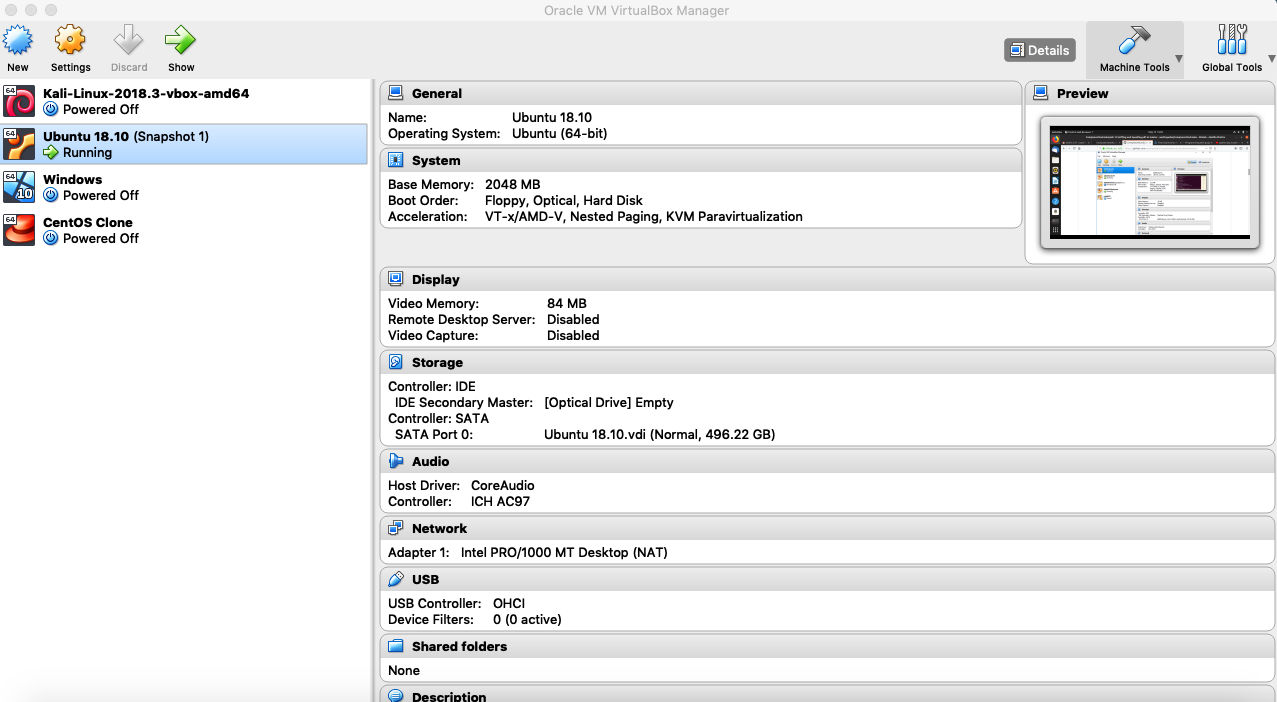


Figure 5

**Observation:** When sniffex is run without root privileges, it says that we don’t have permission to capture on that device.

**Explanation:** Pcap needs root permissions to run sniffex because it has to access the network interface card. Network interface card is the physical device that accepts the packets into the system and only root can access it. The pcap\_lookupdev() in the sniffex program fails as it is looking for the interface to sniff on and this requires root access. But since root privilege is not given, it fails.

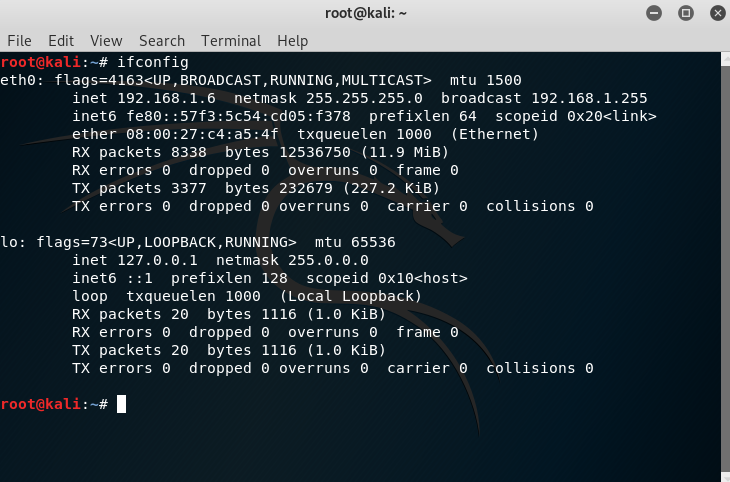
**Problem 3:**

****

**Figure 6**



**Figure 7**



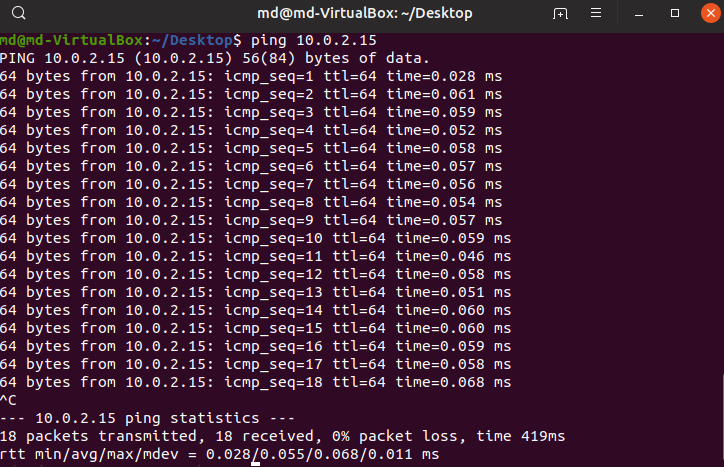
**Figure 8**

**Observation:** For this task we setup 2 machines. We change the networks of the machines from NatNetwork to Host only Adapter so that they work under the same network.

IP Addresses: For convenience,

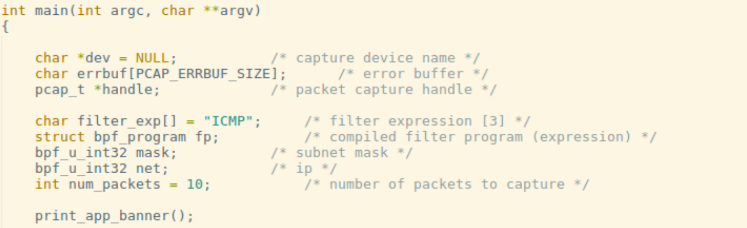
**Ubuntu (M1): 10.0.2.15.**

KaliLinux **(M2): 192.168.1.6**



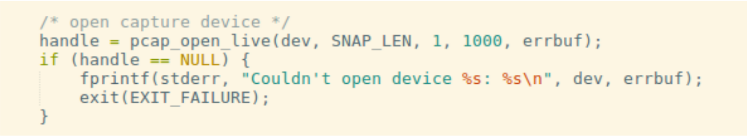
**Figure 9**

**Observation:** We are verifying if ping works and M3 is used to ping M1 (192.168.65.101) and we find that this is successful.



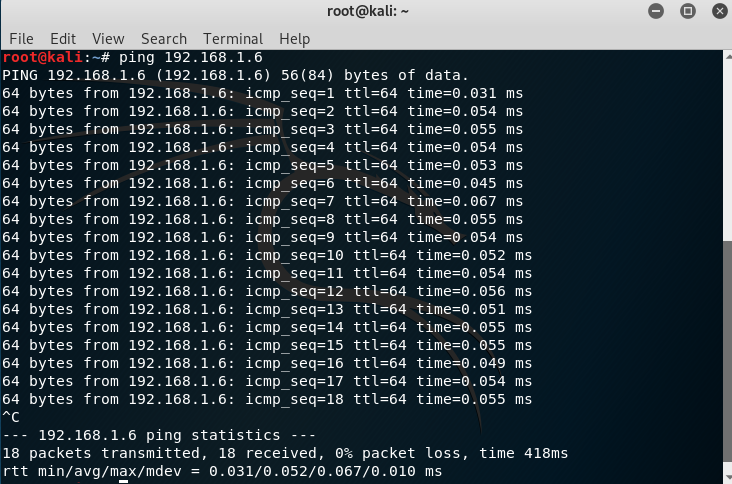
**Figure 10**

**Observation:** We set our filter to capture ICMP packets.



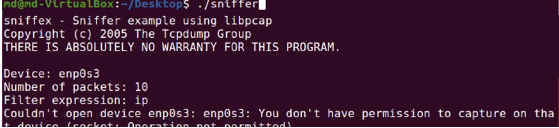
**Figure 11**

**Observation:** We turn on Promiscous mode by setting 3rd argument of pcap\_open\_live as 1.

****

**Figure 12**

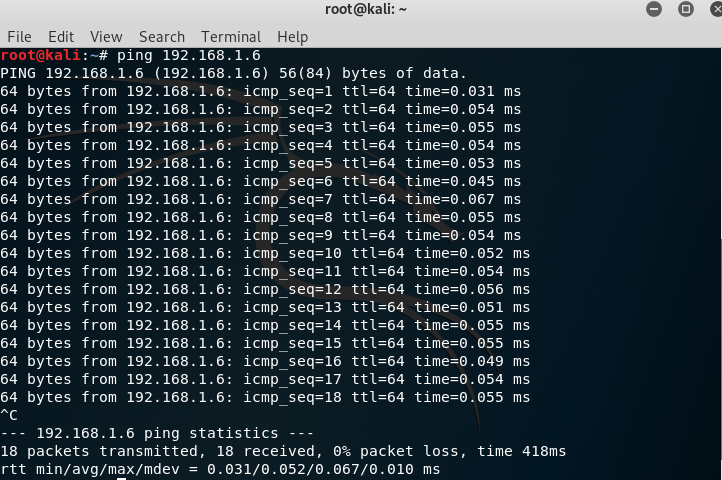
**Observation:** We create a ping from M3 to M2 now.



**Figure 13**

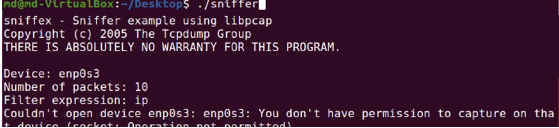
**Observation:** When promiscuous mode is turned on, the user sitting on M1 can observe this connection by running the sniffer program.

**Explanation:** Promiscuous mode bit is set in the pcap\_open\_live() function. The 3rd bit parameter is set to 1, indicating that promiscuous mode is on. When promiscuous mode is on, sniffer program can capture all the packets in the same network regardless of the destination IP.



**Figure 14**

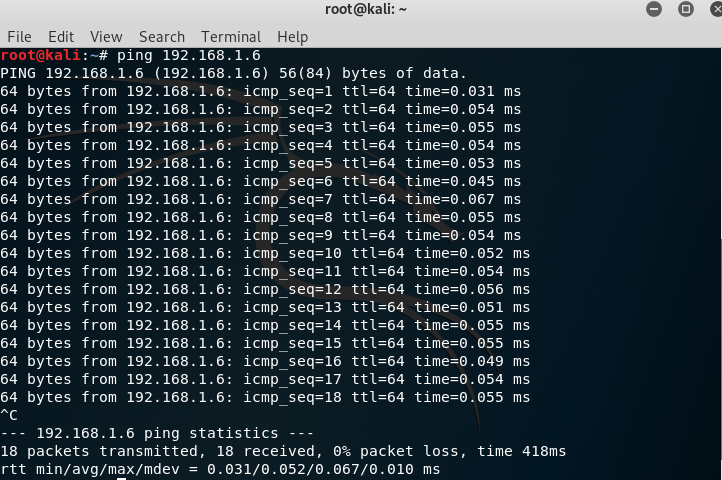
**Observation:** Now, we turn off the promiscuous mode and perform the same ping operation.



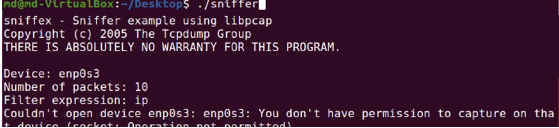
**Figure 15**

**Observation:** We observe that M1 cannot capture packets of communication between M3 and

M2.



**Figure 16**



**Figure 17**

**Observation:** M3 pings M1 when Promiscous mode is turned off. We are able to sniff this.

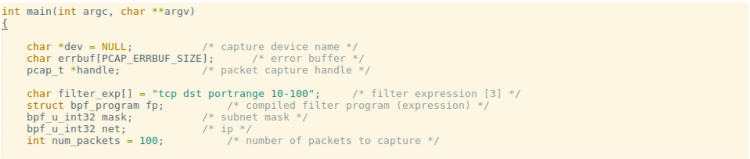
**Explanation**: Promiscuous mode bit is set in the pcap\_open\_live() function. The 3rd bit parameter is set to 0, indicating that promiscuous mode is off. When promiscuous mode is off, sniffer program cannot capture all the packets in the same network, it can only capture packets whose destination IP is the IP of the sniffer’s system.

**Task 1.b: Writing Filters**

• **Capturing ICMP Packets**

Refer to Figure 10, 11, 12, 13 in task 1.a for this task.

• **Capture the TCP packets that have a destination port range from to port 10 – 100**



**Figure 18**

**Observation:** We set our filter expression to TCP packets that have a destination port range 10-100.

**Observation:** The above screenshots depicts that TCP packets are captured that have a destination port number in the range of 10 to 100. The filter expression is shown in the code. When the user establishes a telnet connection, the destination port is 23, which falls in the range, so the sniffer captures those packets.

**Explanation:** Filters are used to capture specific traffic. In the above case we capture TCP packets whose destination port number is between 10 and 100. To apply filter, we first need to create a rule set to filter the traffic, then we need to compile the rule set because the filter has to be understood by pcap. We then need to apply the filter using pcap\_setfilter(). This makes pcap only receive packets based on the filter applied.

**Task 2: Spoofing**

**Task 2.a: Write a spoofing program**

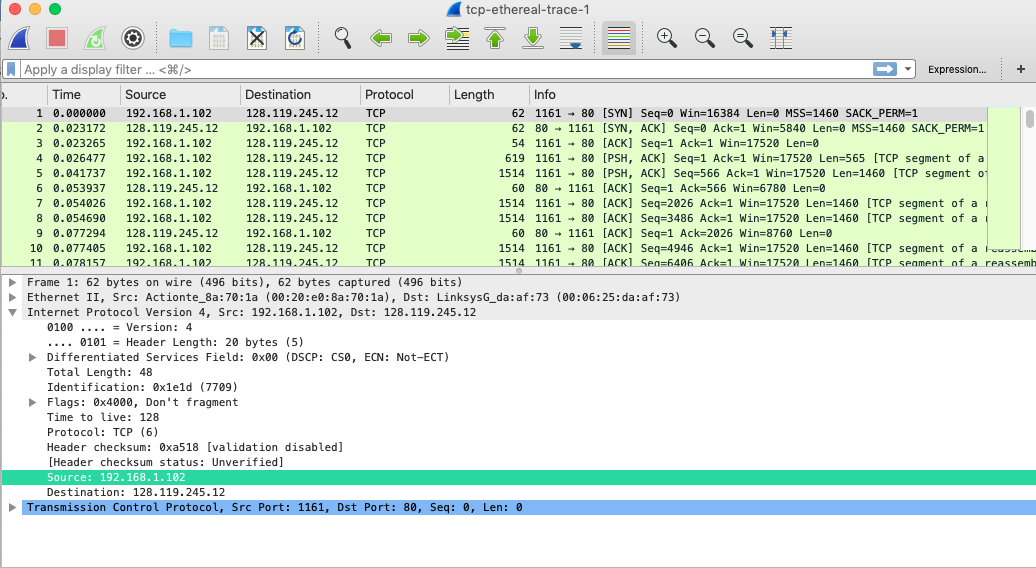
****

**Figure 24**

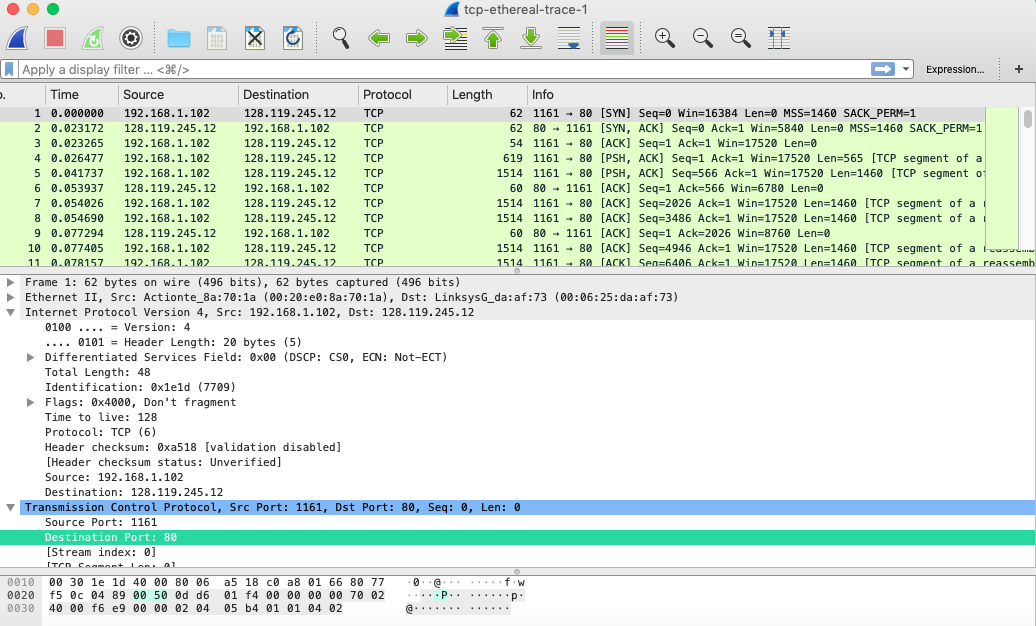
**Observation:** The above screenshot shows our spoofing program.

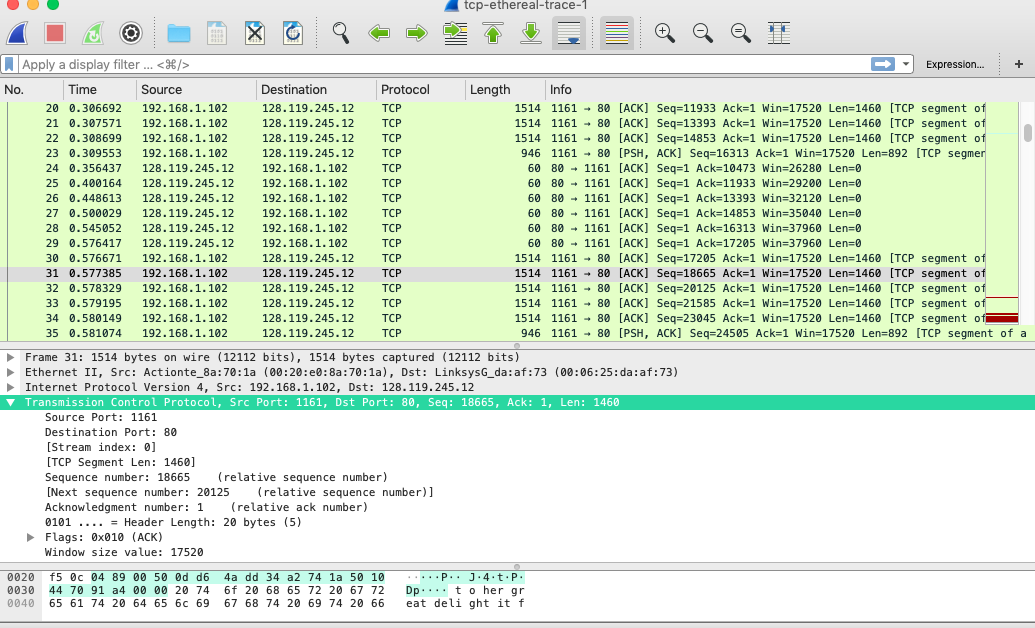
**Observation:** The above screenshot compiles our spoof.c program

Client IP Address: 192.168.1.102, And TCP Port: 1161.

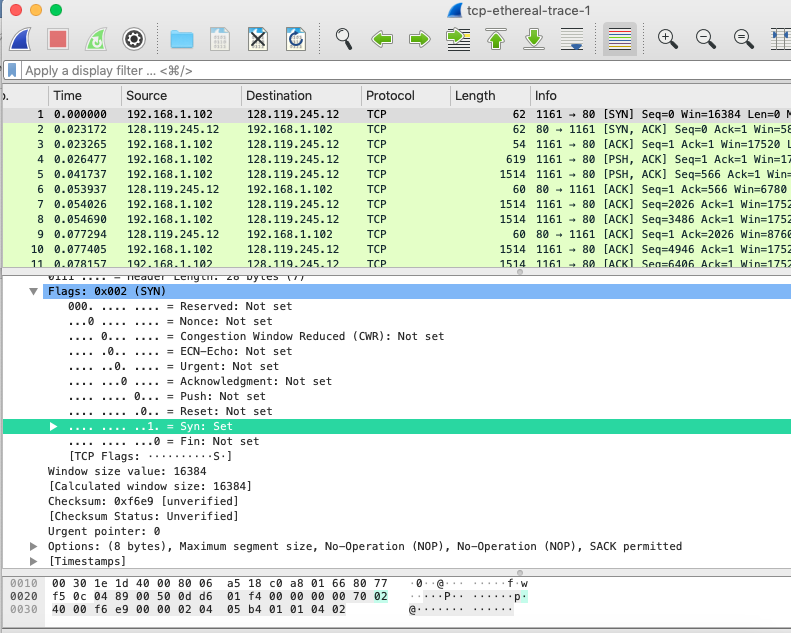


gaia.cs.umass.edu IP address is 128.119.245.12. TCP Port number:80.

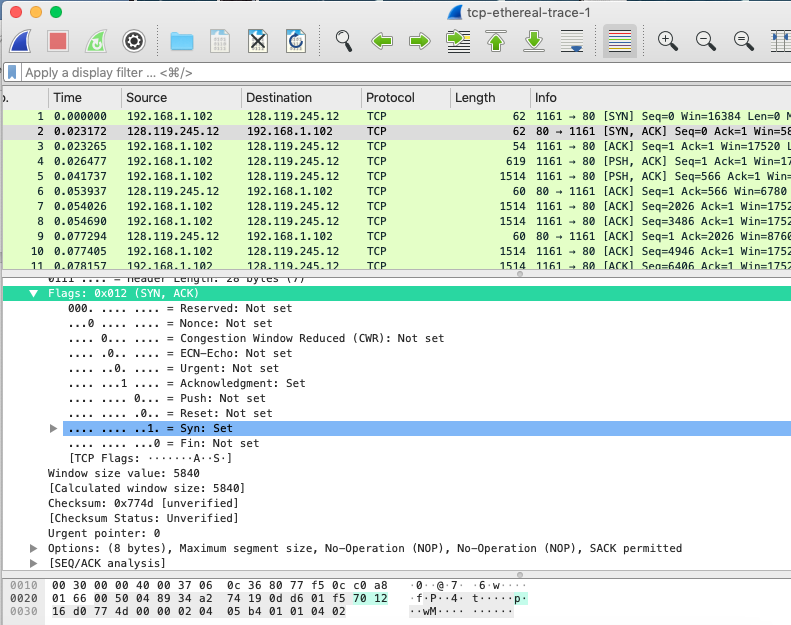
 Client IP Address: 192.168.1.102, And TCP Port: 1161.

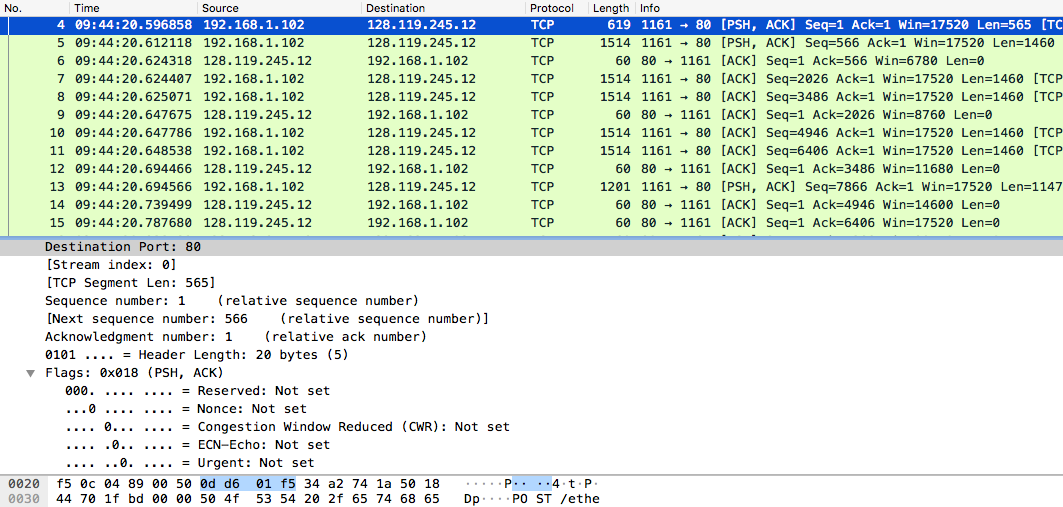


the sequence number of the TCP SYN segment that is used to initiate the TCP is 0. the segment that identifies the segment as a SYN segment is 1.

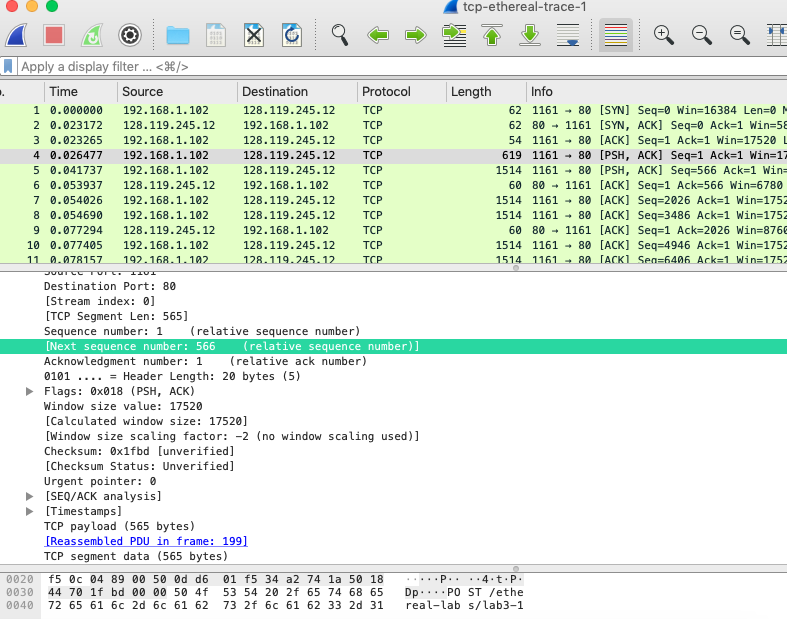


Sequence number of the SYNACK segment from gaia.cs.umass.edu to the client computer in reply to the SYN has the value of 0 in this trace. The value of the ACK field in the SYNACK segment is 1. The value of the ACK field in the SYNACK segment is determined by gaia.cs.umass.edu by adding 1 to the initial sequence number of SYN segment from the client computer (i.e. the sequence number of the SYN segment initiated by the client computer is 0. The SYN flag and Acknowledgement flag in the segment are set to 1 and they indicate that this segment is a SYNACK segment.

4 segment is the TCP segment containing the HTTP POST command. The sequence number of this segment has the value of 1.



Segment 4, 5, 7, 8, 10, and 11 in this trace respectively. The ACKs of segments 1 to 6 are No. 6, 9, 12, 14, 15, and 16 in this trace.



Segment 1 sequence number: 1



Segment 2 sequence number: 566



Segment 3 sequence number: 2026



Segment 4 sequence number: 3486



Segment 5 sequence number: 3486

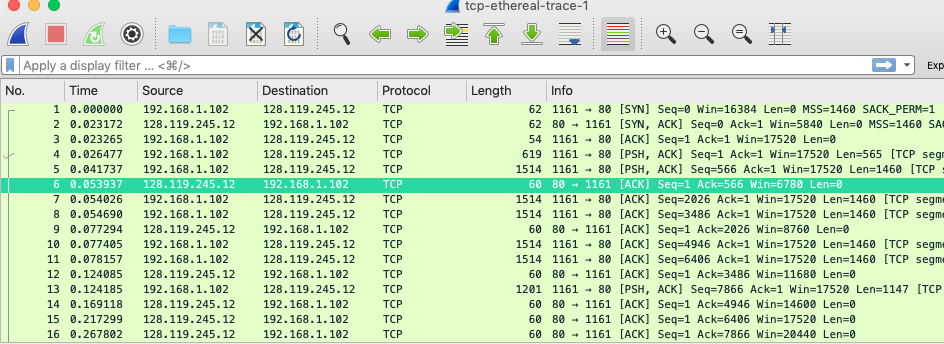
Screen%20Shot%202019-03-18%20at%201.55.40%20PM.png

Segment 6 sequence: 6406



|  |  |  |  |
| --- | --- | --- | --- |
|  | Sent time | ACK received time | RTT(seconds) |
| Segment 1 | 0.026477 | 0.053937 | 0.02746 |
| Segment 2 | 0.041737 | 0.077294 | 0.035557 |
| Segment 3 | 0.054026 | 0.124085 | 0.070059 |
| Segment 4 | 0.054690 | 0.169118 | 0.11443 |
| Segment 5 | 0.077405 | 0.217299 | 0.13989 |
| Segment 6 | 0.078157 | 0.267802 | 0.18964 |
|  |  |  |  |

ACK received time: The ACKs of segments 1 to 6 are No. 6, 9, 12, 14, 15, and 16 in this trace.



Estimated RTT = 0.875 \* Estimated RTT + 0.125 \* Sample RTT

Estimated RTT after the receipt of the ACK of segment 1: Estimated RTT = RTT for Segment 1 = 0.02746 second

Estimated RTT after the receipt of the ACK of segment 2:  
Estimated RTT = 0.875 \* 0.02746 + 0.125 \* 0.035557 = 0.0285

Estimated RTT after the receipt of the ACK of segment 3:  
Estimated RTT = 0.875 \* 0.0285 + 0.125 \* 0.070059 = 0.0337

Estimated RTT after the receipt of the ACK of segment 4:  
Estimated RTT = 0.875 \* 0.0337+ 0.125 \* 0.11443 = 0.0438

Estimated RTT after the receipt of the ACK of segment 5:  
Estimated RTT = 0.875 \* 0.0438 + 0.125 \* 0.13989 = 0.0558

Estimated RTT after the receipt of the ACK of segment 6:  
Estimated RTT = 0.875 \* 0.0558 + 0.125 \* 0.18964 = 0.0725

8. Length is actually the difference between the sequence number.

Segment 1 sequence number: 1

segment 1: 566 – 1 = 565

Segment 2 sequence number: 566



segment 2: 2026 – 566 = 1460

Segment 3 sequence number: 2026



segment 3: 3486 – 2026 = 1460

Segment 4 sequence number: 3486



segment 4: 4946 -3486 = 1460

Segment 5 sequence number: 3486

Screen%20Shot%202019-03-18%20at%201.55.40%20PM.png

segment 5: 6406 – 4946 = 1460

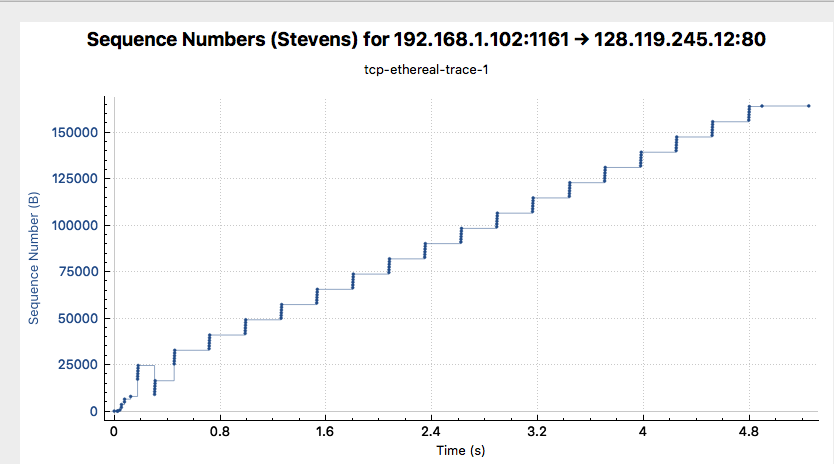
Segment 6 sequence: 6406



segment 6: 7866 – 6406 = 1460

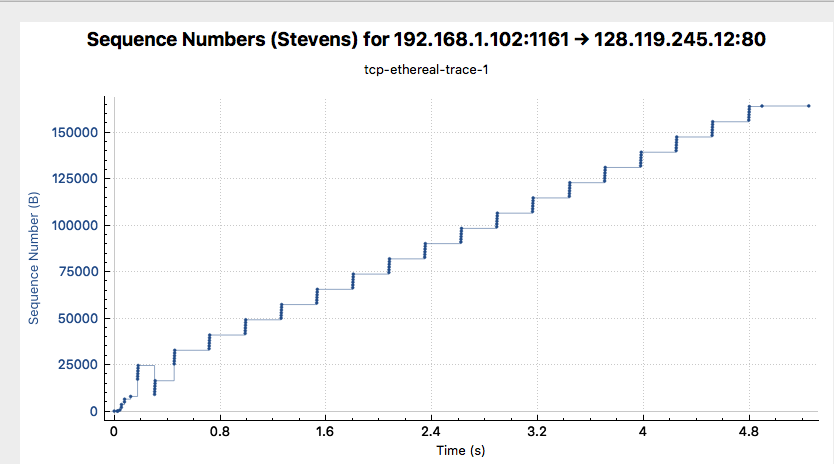
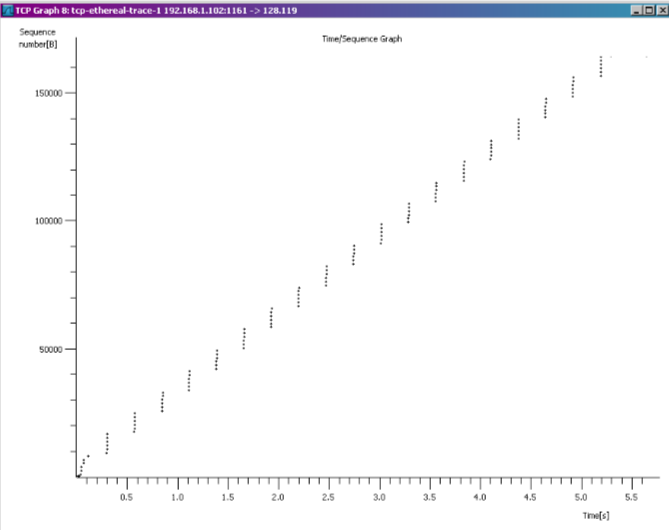
The minimum amount of buffer space (receiver window) advertised at gaia.cs.umass.edu for the entire trace is 5840 bytes, which shows in the first acknowledgement from the server. This receiver window grows steadily until a maximum receiver buffer size of 62780 bytes. The sender is never throttled due to lacking of receiver buffer space by inspecting this trace.



Sequence number is increasing along with the time so there will be no retransmitted segments.

|  |  |  |
| --- | --- | --- |
|  | ACK sequence number | ACK data |
| ACK 1 | 566 | 566 |
| ACK 2 | 2026 | 1460 |
| ACK 3 | 3486 | 1460 |
| ACK 4 | 4946 | 1460 |
| ACK 5 | 6406 | 1460 |
| ACK 6 | 7866 | 1460 |
| ACK 7 | 9013 | 1147 |
| ACK 8 | 10473 | 1460 |
| ACK 9 | 11933 | 1460 |
| ACK 11 | 13393 | 1460 |
| ACK 10 | 14853 | 1460 |
| ACK 12 | 16313 | 1460 |

The computation of TCP throughput largely depends on the selection of averaging time period. As a common throughput computation, in this question, we select the average time period as the whole connection time. Then, the average throughput for this TCP connection is computed as the ratio between the total amount data and the total transmission time. The total amount data transmitted can be computed by the difference between the sequence number of the first TCP segment (i.e. 1 byte for No. 4 segment) and the acknowledged sequence number of the last ACK (164091 bytes for No. 202 segment). Therefore, the total data are 164091 - 1 = 164090 bytes. The whole transmission time is the difference of the time instant of the first TCP segment (i.e., 0.026477 second for No.4 segment) and the time instant of the last ACK (i.e., 5.455830 second for No. 202 segment). Therefore, the total transmission time is 5.455830 - 0.026477 = 5.4294 seconds. Hence, the throughput for the TCP connection is computed as 164090/5.4294 = 30.222 KByte/sec.

congestion takes place at the bottom of each of the line where the data has been sent. congestion avoidance takes place on the gap between the data of time.

Reference :

1. <https://www.studocu.com/en-us/document/george-mason-university/introduction-network-security/practical/final-submission-for-cyse-330/3514022/view>”