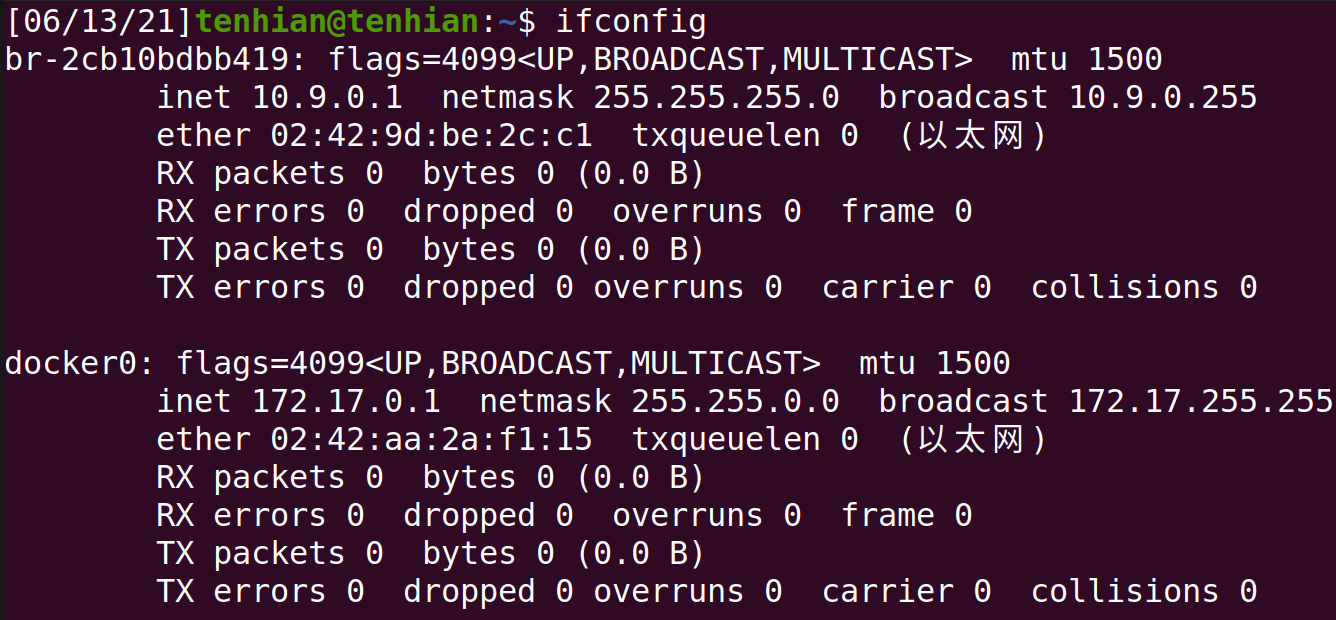
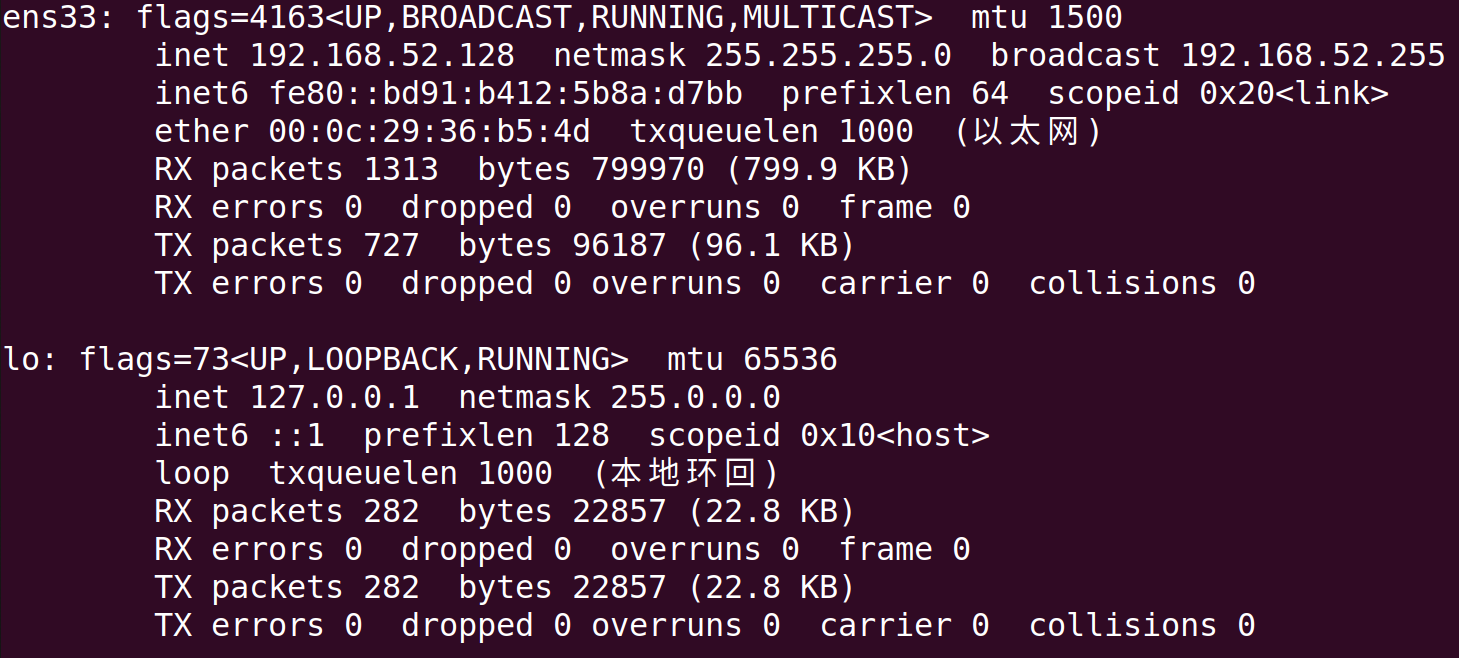
# 实验一流程remake

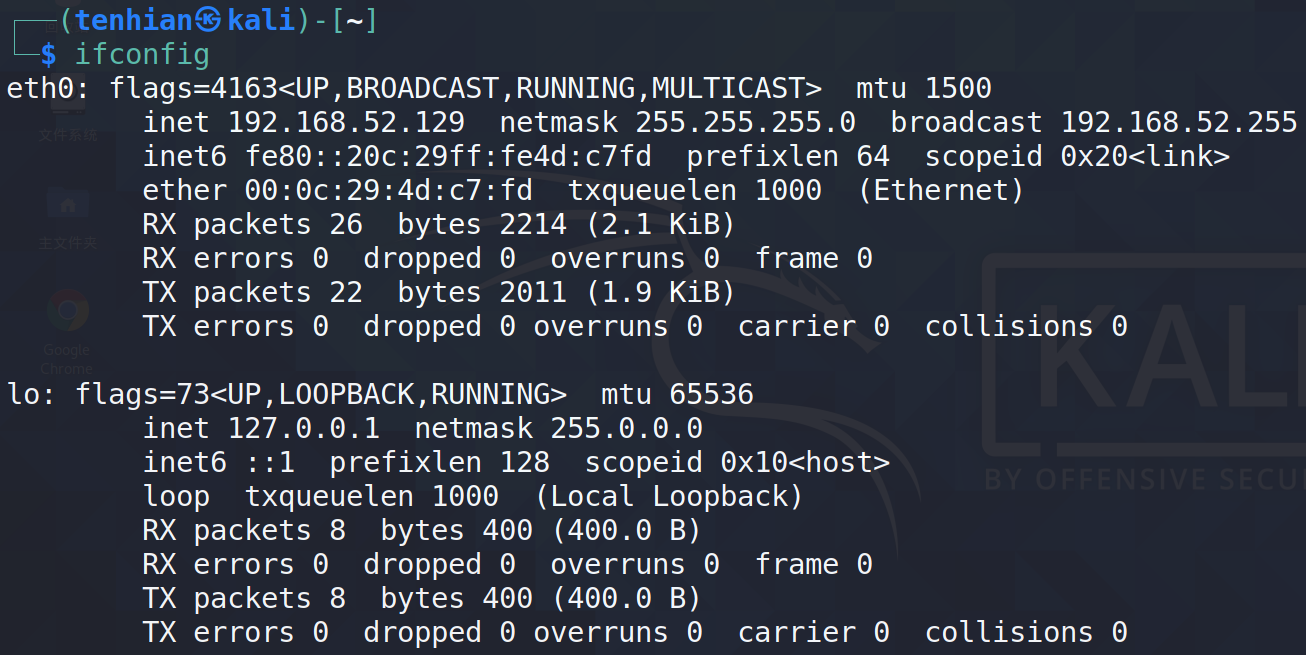
produced by TenHian

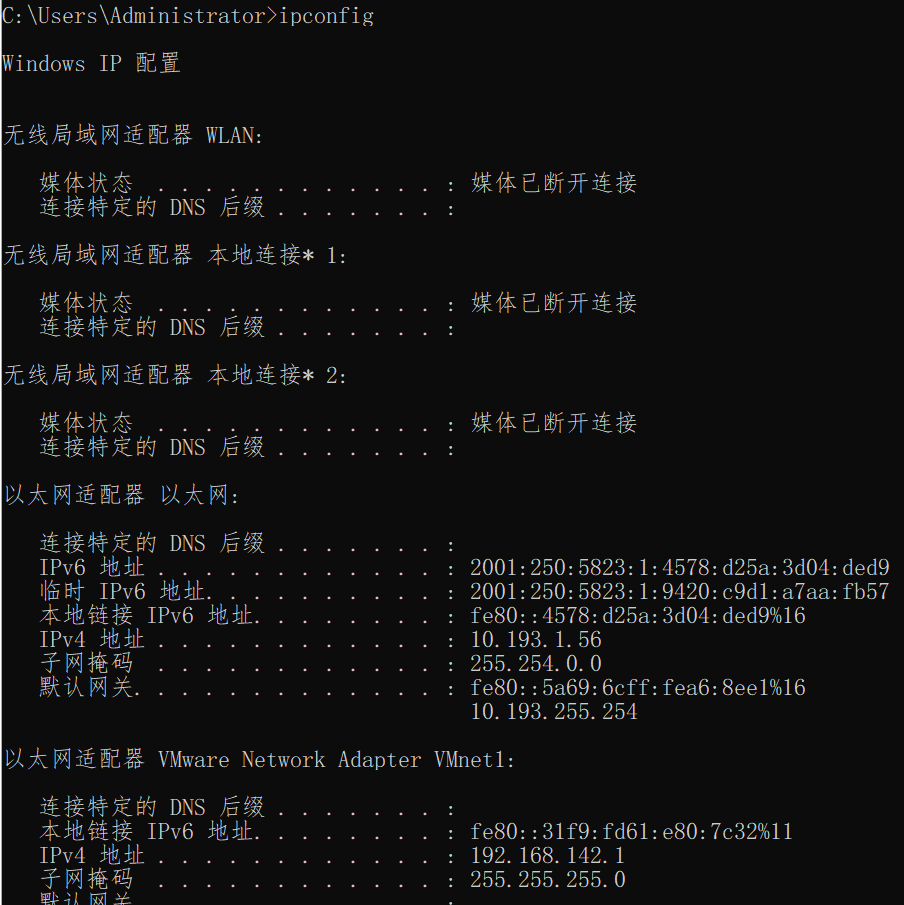
ifconfig/ipconfig命令的讲解:  
 ifconfig命令和ipconfig命令的功能是相同的，区别是ifconfig是linux命令，ipconfig是win系统命令，其作用都是查看本机网络信息  
  
我们在Ubuntu上运行ifconfig命令





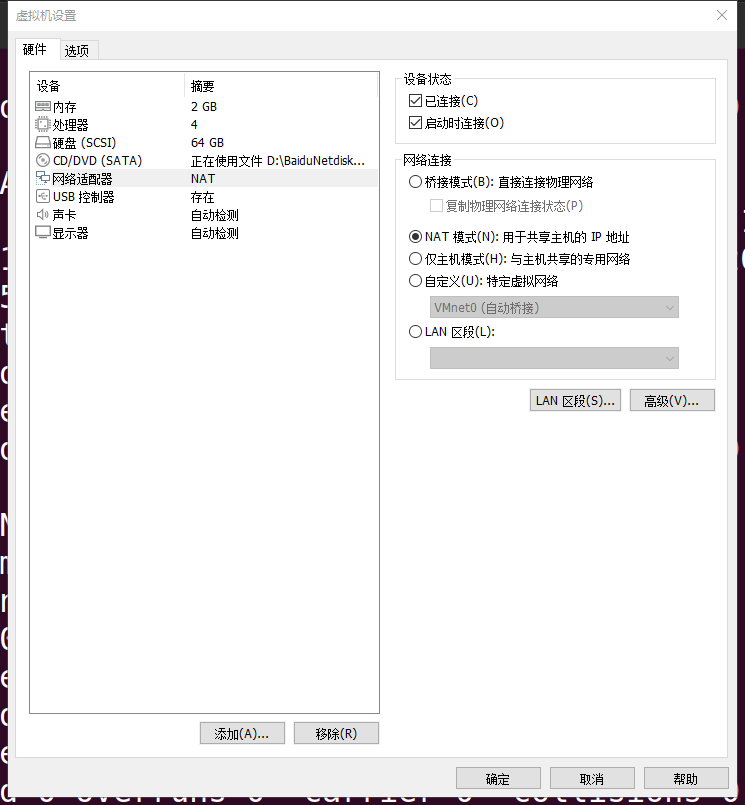
我们可以看到我这台Ubuntu上有三张网卡   
br-2cb10bdbb419 docker0 ens33  
和一个本地环回  
lo   
以 ens33为例，我们介绍一下命令返回结果的含义  
ens33: flags=4163<UP,BROADCAST,RUNNING,MULTICAST> mtu 1500  
网卡名称(iface):ens33 状态代码:flags=4163  
<UP,BROADCAST,RUNNING,MULTICAST>  
<启用中，广播中，运行中，混杂模式>其中RUNNING表示这一块是该主机正在运行的网卡，该网卡的IP地址就是本机IP地址 mtu 1500 最大传输单元1500字节  
inet 192.168.52.128 netmask 255.255.255.0 broadcast 192.168.52.255  
IP地址:192.168.52.128 掩码:255.255.255.0 广播地址:192.168.52.255  
inet6 fe80::bd91:b412:5b8a:d7bb prefixlen 64 scopeid 0x20⁢<link>  
IPv6地址: fe80::bd91:b412:5b8a:d7bb  
ether 00:0c:29:36:b5:4d txqueuelen 1000 (以太网)   
网卡Mac:00:0c:29:36:b5:4d 连接方式为以太网  
往下的数据包统计不做讲解  
  
在我的Kali和win上分别运行ifconfig和ipconfig命令



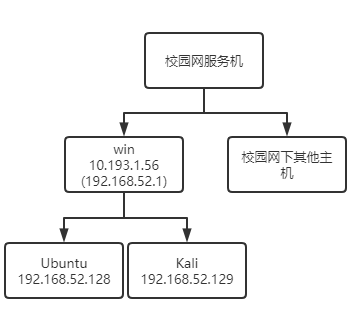


则各个机器IP为  
Ubuntu:192.168.52.128 网卡为ens33  
Kali:192.168.52.129 网卡为eth0  
win:10.193.1.56

win为实机，Ubuntu和Kali为虚拟机，虚拟机与实机的网络连接方式为NAT模式



下图为我这三台计算机的组网示意图



## Lab Task Set 1: Using Scapy to Sniff and Spoof Packets

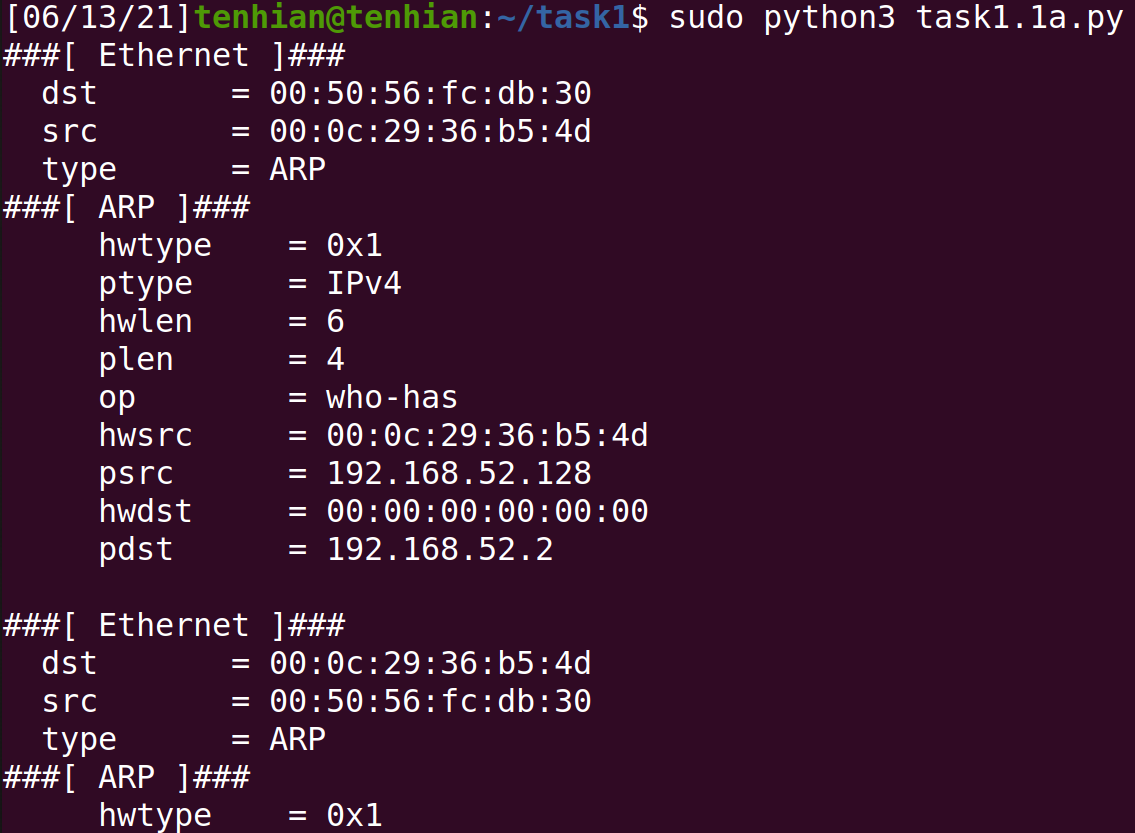
### Task 1.1: Sniffing Packets

#### Task 1.1A

写出下面程序，调用print pkt()函数打印抓取的包信息

#!/usr/bin/env python3  
  
from scapy.all import \*  
def print\_pkt(pkt):  
 pkt.show()   
  
pkt = sniff(iface='ens33', prn=print\_pkt)  
  
print\_pkt(pkt)

#!/usr/bin/env python3指定Python3解释器  
pkt = sniff(iface='ens33', prn=print\_pkt)  
嗅探设备ens33的数据包，赋值给pkt并交由函数 print\_pkt打印  
  
运行代码



关闭终端，Task1.1A完成

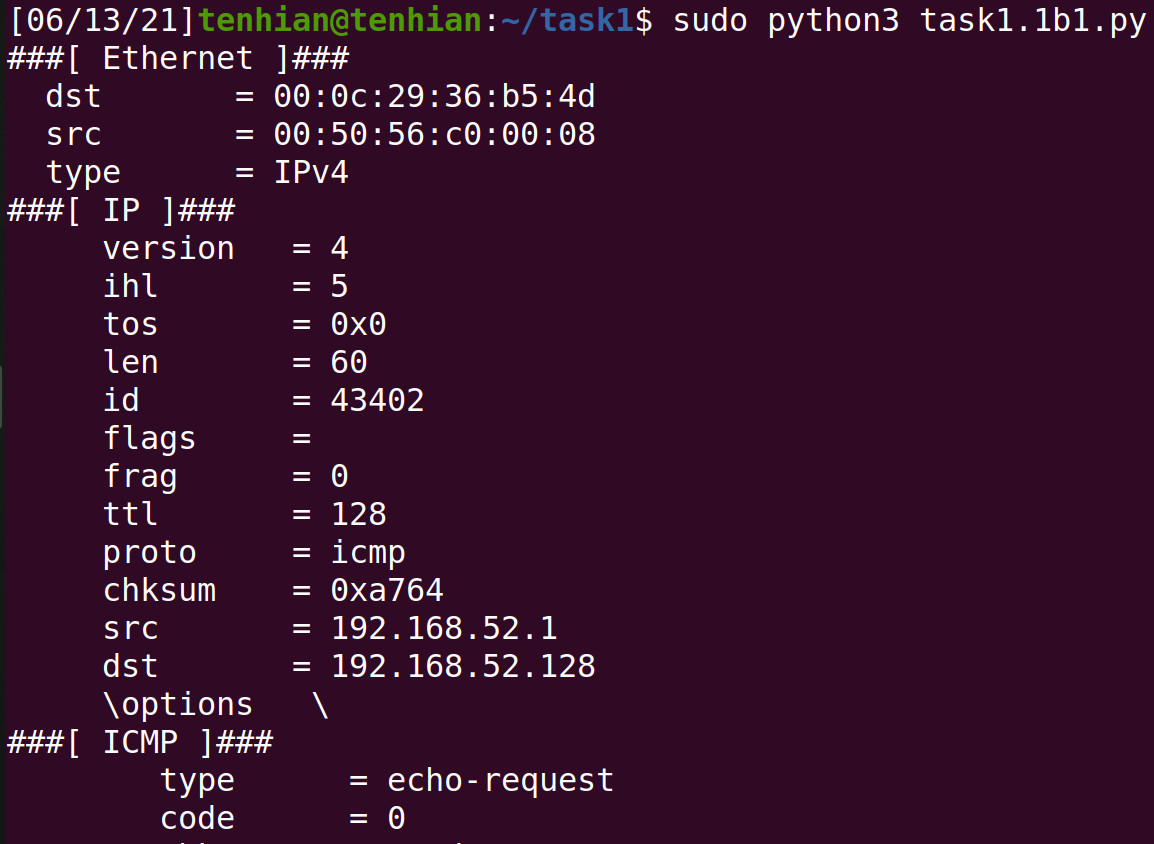
#### Task 1.1B

**Capture only the ICMP packet**

只抓取ICMP类型的数据包  
代码:

#!/usr/bin/env python3  
  
from scapy.all import \*  
def print\_pkt(pkt):  
 pkt.show()   
  
pkt = sniff(iface='ens33', filter='icmp', prn=print\_pkt)  
  
print\_pkt(pkt)

运行:



关闭终端，Task1.1B-1完成

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

只抓取 来自特定IP 23号端口 TCP类型 的数据包  
代码:

#!/usr/bin/env python3  
  
from scapy.all import \*  
def print\_pkt(pkt):  
 pkt.show()   
  
pkt = sniff(iface='ens33', filter='tcp && src 192.168.52.129 && port 23', prn=print\_pkt)  
  
print\_pkt(pkt)

这个程序要抓取来自 192.168.52.129(Kali) 通过23端口 发来的TCP包，23端口是远程登陆指令 telnet 的端口,所以我们要在Ubuntu和Kali上配置Telnet客户端和服务器

在Ubuntu和Kali上分别执行下述过程

安装telnet客户端:

sudo apt-get install telnet

安装telnet服务端:

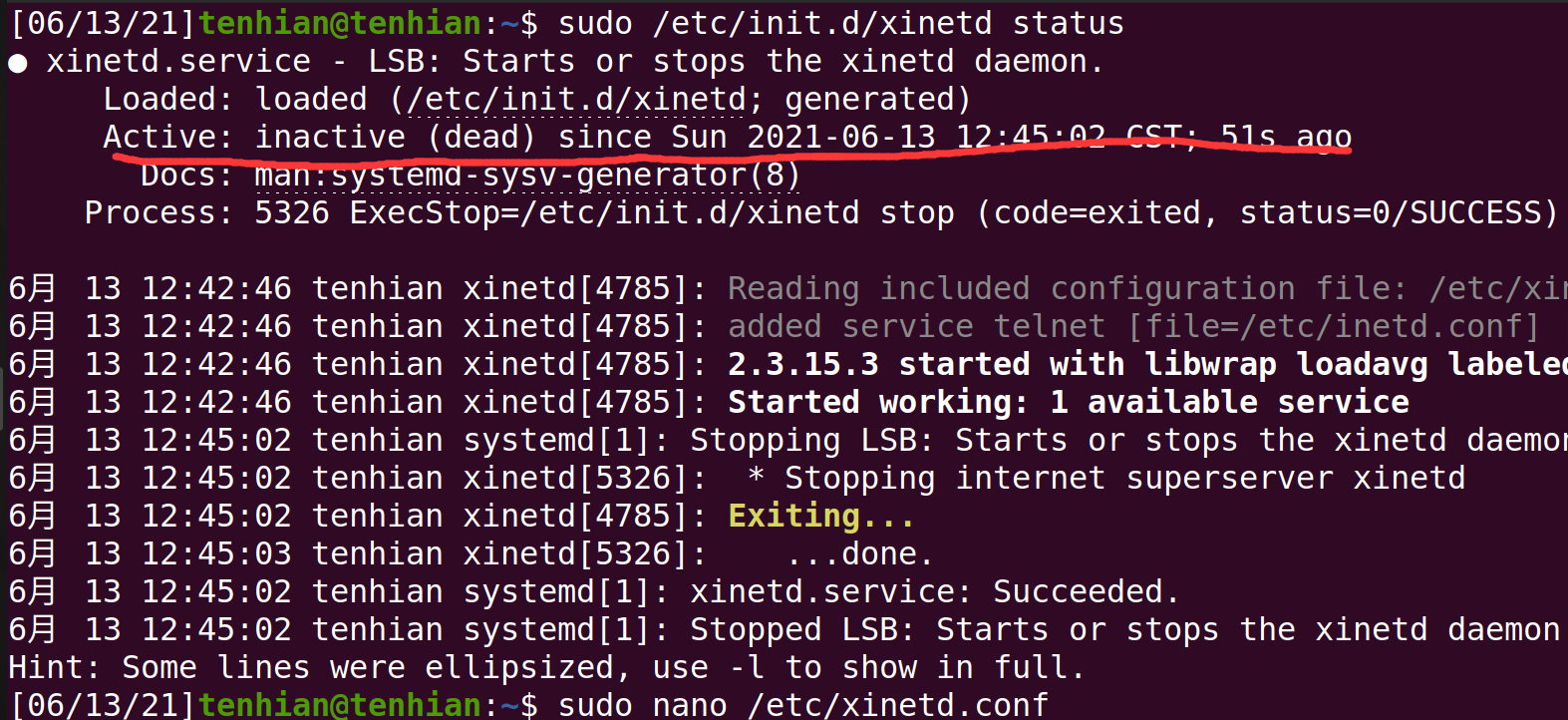
sudo apt-get install telnetd

安装xinetd

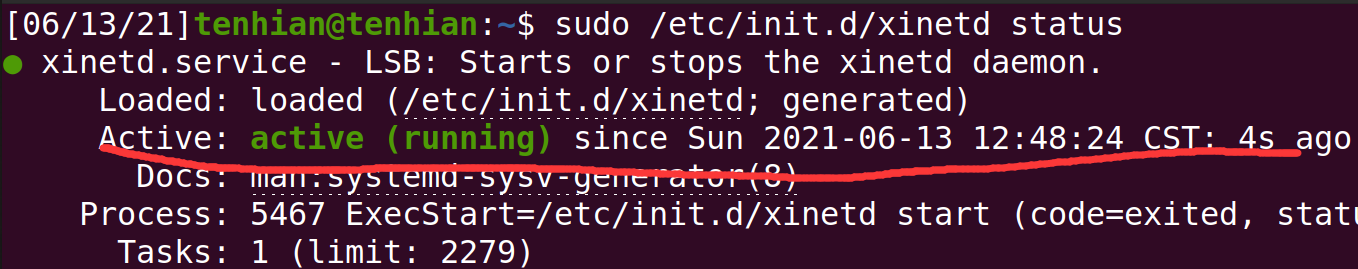
sudo apt-get install xinetd

查看xinetd状态

sudo /etc/init.d/xinetd status



上图为服务关闭状态



上图为服务开启状态  
有以下命令

sudo /etc/init.d/xinetd status //查看服务状态  
sudo /etc/init.d/xinetd start //开启服务  
sudo /etc/init.d/xinetd stop //关闭服务  
sudo /etc/init.d/xinetd restart //重启服务

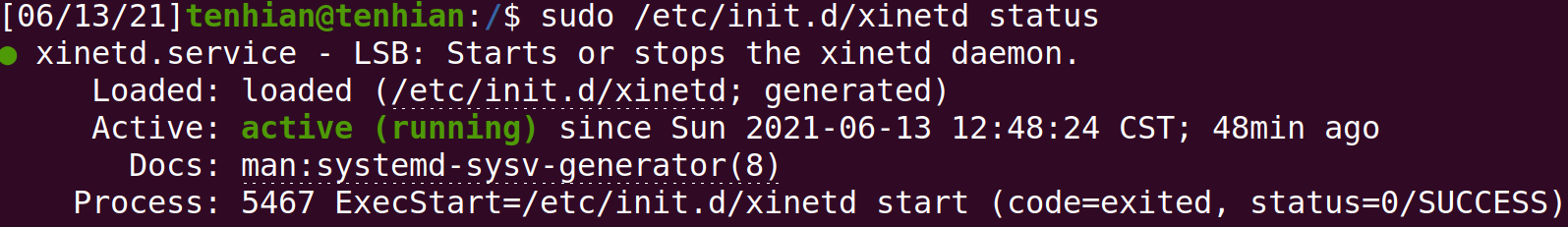
若服务开启状态则执行命令关闭服务  
修改配置文件

sudo nano /etc/xinetd.conf //个人比较喜欢用nano这个文本编辑器，也可用vim打开，如  
sudo vim /etc/xinetd.conf

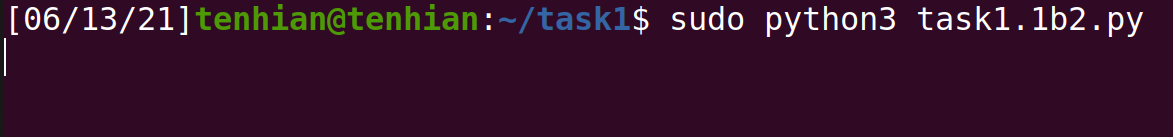
修改为:

defaults  
{  
# Please note that you need a log\_type line to be able to use log\_on\_success  
# and log\_on\_failure. The default is the following :  
# log\_type = SYSLOG daemon info  
instances = 60  
log\_type = SYSLOG authpriv  
log\_on\_success = HOST PID  
log\_on\_failure = HOST  
cps = 25 30  
}

返回命令行，在Ubuntu启动服务，并查看服务状态是否为启用

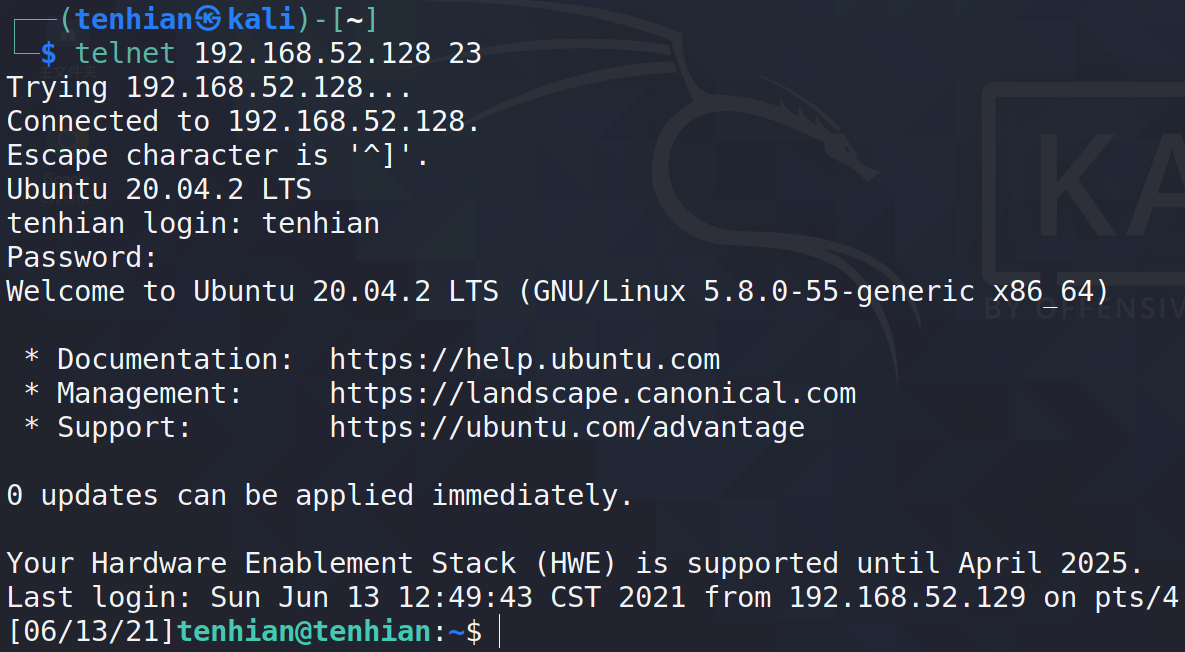


启用后运行代码

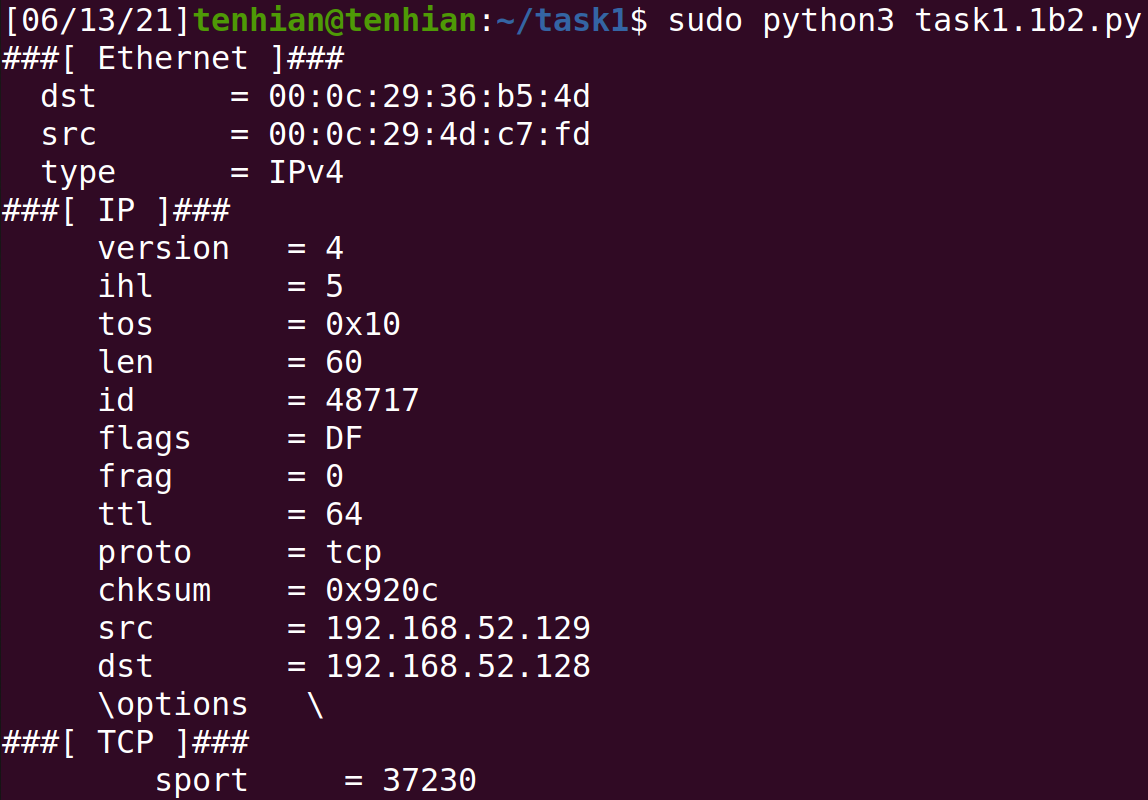


然后在Kali中执行命令，通过23号端口远程登陆 192.168.52.128(Ubuntu)

telnet 192.168.52.128 23



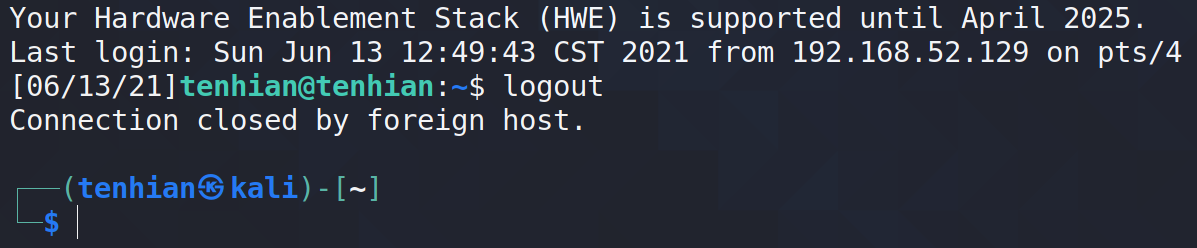
输入Ubuntu的用户名密码登陆，这时程序已抓到包



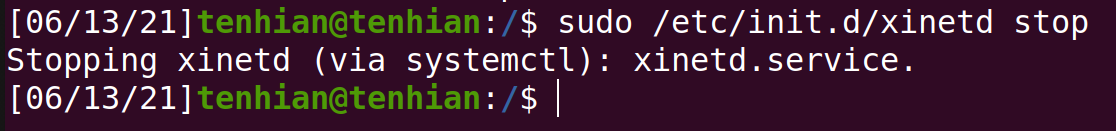
在Kali中执行命令

logout

退出登录，断开连接



在Ubuntu中关闭服务



关闭运行抓包代码的终端，程序停止运行，Task1.1B-2完成

**Capture packets comes from or to go to a particular subnet. You can pick any subnet, such as 128.230.0.0/16; you should not pick the subnet that your VM is attached to**

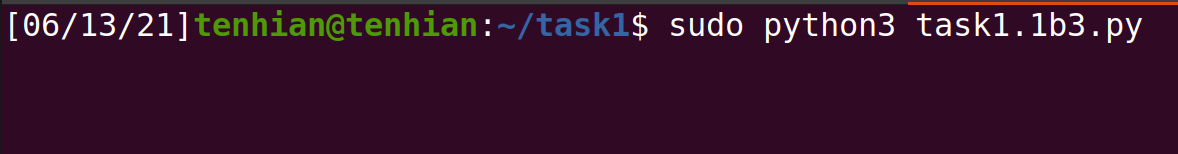
捕获来自或去往特定子网的数据包。可以挑任意子网，如 128.230.0.0/16；不能选择包括当前虚拟机的子网

由于win 192.168.52.1 Ubuntu 192.168.52.128 Kali 192.168.52.129 处于win 10.193.1.56分配的子网中，不包括当前虚拟机(192.168.52.128)的子网，可以指定成分配Ubuntu(192.168.52.128)的win(10.193.1.56)所在层级的子网

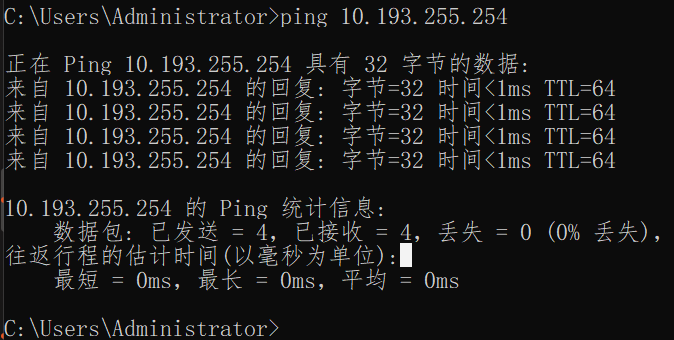
#!/usr/bin/env python3  
  
from scapy.all import \*  
def print\_pkt(pkt):  
 pkt.show()   
  
pkt = sniff(iface='ens33', filter='net 10.193.255.0/24', prn=print\_pkt)  
  
print\_pkt(pkt)

filter='net 10.193.255.0/24'意义为子网范围从10.193.255.1到10.193.255.254

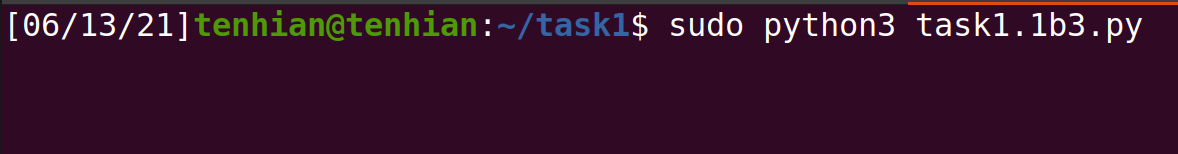
运行程序



在win中 ping 10.193.255.254



还是什么都没抓到



只有数据包流经选的网卡，才能捕捉数据包，包括 监听设备所在层级网络中计算机之间的数据交互(Ubuntu与Kali的数据交互) 和 监听设备所在层级网络中计算机与外网设备之间的交互(访问百度)

关闭终端，Task1.1B-3完成

### Task 1.2: Spoofing ICMP Packets

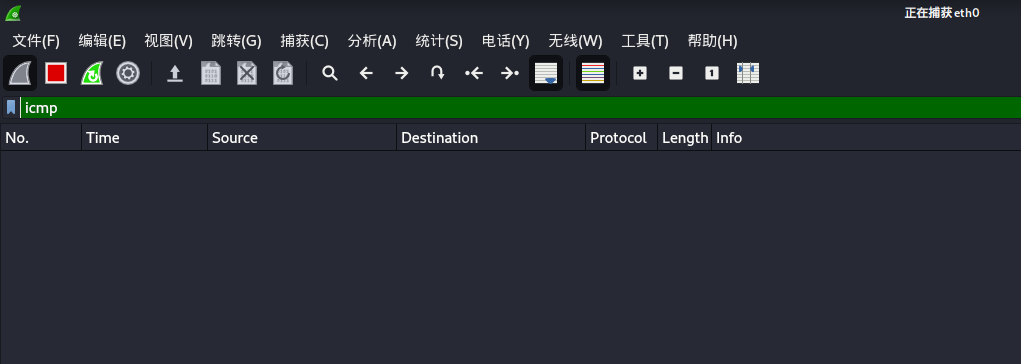
编写程序向某一IP发送一个ICMP-echo-request

我们在使用ping指令时，会用源IP向目的IP发送4个ICMP数据包，这种由源发出发向目的的 ICMP 包称为 ICMP-echo-request ，目的IP收到源IP发送的 ICMP-echo-request 后会向源回复相应的 ICMP-echo-reply 若源收到回复，则为ping通

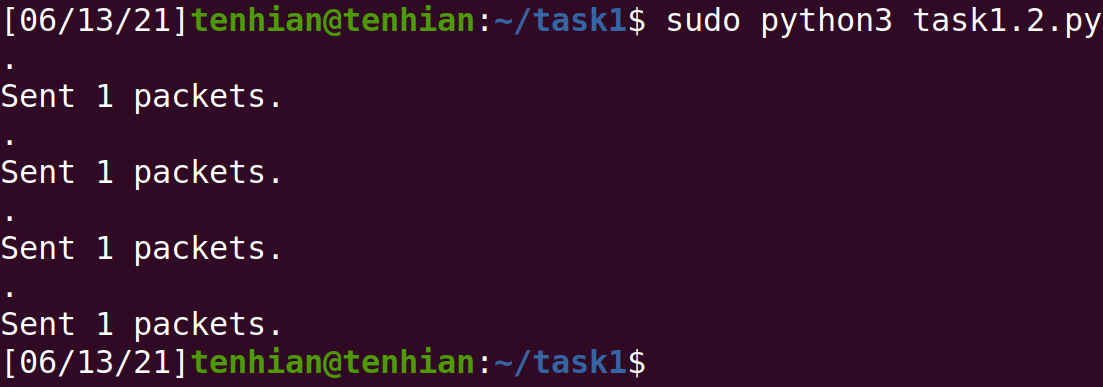
我们向Kali (192.168.52.129) 发送4个 ICMP-echo-request ，代码:

#!/usr/bin/env python3  
  
from scapy.all import \*  
  
for i in range(0,4): #循环4次  
 a=IP() #定义一个IP头  
 a.dst='192.168.52.129' #设定目的IP  
 b=ICMP() #定义一个ICMP头  
 c=a/b #将两个头组合，形成一个ICMP包  
 send(c) #发送包

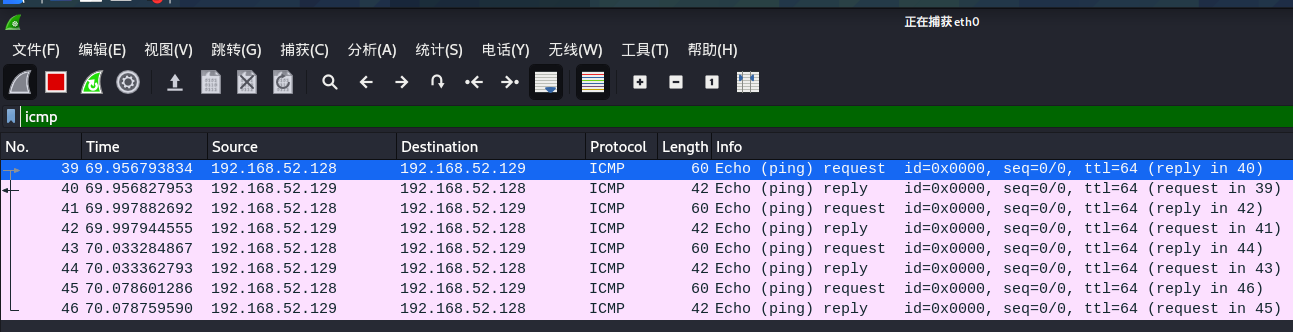
在Kali上打开wireshark，监听网卡eth0，过滤条件为ICMP，开始嗅探



运行写的程序



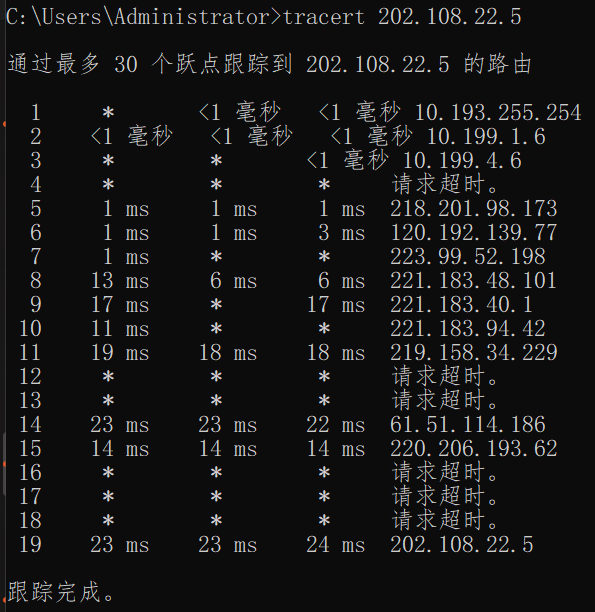
查看wireshark的捕获



ICMP-echo-request与ICMP-echo-reply成对出现，共4对  
关闭终端，task1.2完成

### Task 1.3: Traceroute

有关traceroute和tracert的解释:  
这两个命令的功能是相同的，只不过是一个linux命令一个win命令，用来追踪源IP访问目的IP需要多少次路由(经过多少个跃点 or 经过多少次转发)，用于测两台主机在网路上的距离

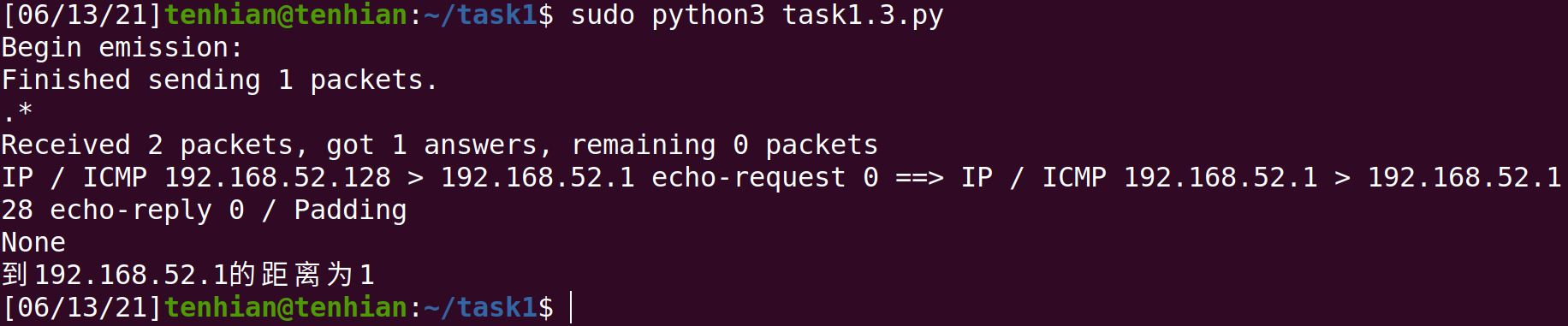


访问百度需要这些次转发

我们将使用scapy库，来模拟traceroute命令的功能，具体思路是，将发送的IP数据包的生存周期(ttl)初始设为1，然后再加1级的增加，直到访问成功或失败，IP数据包就会在发送通路上所有转发IP停留一次  
代码：

#!/usr/bin/env python3  
  
from scapy.all import \*  
  
final=0  
ttl=1  
a=IP()  
a.dst='192.168.52.1' #目的IP  
b=ICMP()  
  
while final==False:  
 a.ttl=ttl  
 ans,unas=sr(a/b)  
 print(ans.summary())  
  
 if ans.res[0][1].type==0:  
 final=True  
 else:  
 ttl+=1  
  
print("到%s的距离为%d"%(a.dst,ttl))

运行代码



task1.3完成

### Task 1.4: Sniffing and-then Spoofing

嗅探所有的ICMP包，不论是发往任何IP，都回复源IP一个对应的ICMP-echo-reply

下面是一些常用的网络协议包解析  
我们写出以下代码，来随意的捕获一些流量包，以ICMP包为例

#!/usr/bin/env python3  
  
from scapy.all import \*  
def print\_pkt(pkt):  
 pkt.show()   
  
pkt = sniff(iface='ens33', prn=print\_pkt)  
  
print\_pkt(pkt)

首先，网络数据包由两部分组成 头部和报文   
头部可以有多个，来记录一些数据包首要信息：“从哪里来？要送到哪去？数据包类型？等等”  
报文记录着这个数据包要传送的信息，以ICMP包为例，我们捕获到的ICMP包的格式是这样的

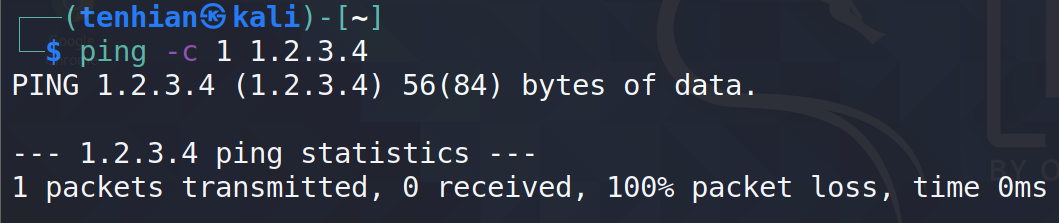
###[ Ethernet ]### //以太网首部，MAC头，共14字节  
 dst = 00:50:56:c0:00:08 //目的MAC地址  
 src = 00:0c:29:36:b5:4d //源MAC地址  
 type = IPv4 //以太网类型  
###[ IP ]### //IP头，20字节  
 version = 4 //版本 4字节  
 ihl = 5 //首部长度  
 tos = 0x0 //服务类型  
 len = 84 //总长  
 id = 20312 //标识  
 flags = DF //标志位3字节  
 frag = 0 //段偏移量  
 ttl = 64 //存活时间  
 proto = icmp //协议  
 chksum = 0x17f //校验和  
 src = 192.168.52.128 //源IP地址  
 dst = 192.168.52.1 //目的IP地址  
 \options \  
###[ ICMP ]### //ICMP头  
 type = echo-request //消息类型  
 code = 0 //代码  
 chksum = 0x323d //校验和  
 id = 0x1 //身份标识  
 seq = 0x1 //序列号  
 unused = ''  
###[ Raw ]### //报文  
 load = '\x1f\\xfd\\xc9`\x00\x00\x00\x00\x1d\\x90\x00\x00\x00\x00\x00\x00\x10\x11\x12\x13\x14\x15\x16\x17\x18\x19\x1a\x1b\x1c\x1d\x1e\x1f !"#$%&\'()\*+,-./01234567'

可见一个ICMP包包含 Ether头 IP头 ICMP头 和 报文   
其中着重讲一下校验和 chksum 这是导致数据包被判断为正确数据包和错误数据包的依据  
IP头校验和  
在发送IP包时，将chksum置0，然后将IP报头按16比特分成多个单元，如包头长度不是*16*比特的倍数，则用0比特填充到16比特的倍数；  
对各个单元采用反码加法运算(即高位溢出位会加到低位,通常的补码运算是直接丢掉溢出的高位),将得到的和的反码填入校验和字段。  
IP包被接收时，将IP包头按16比特分成多个单元，如包头长度不是16比特的倍数，则用0比特填充到16比特的倍数；  
对各个单元采用反码加法运算，检查得到的和是否符合是全1\*有的实现可能对得到的和会取反码，然后判断最终值是不是全0)；  
如果是全1则进行下步处理,否则意味着包已变化从而丢弃之。需要强调的是反码和是采用高位溢出加到低位的，如3比特的反码和运算：100b+101b=010b(因为100b+101b=1001b,高位溢出1，其应该加到低位，即001b+1b(高位溢出位)=010b)。  
  
在Python的scapy库中，能通过简单的对头部初始化来置校验和

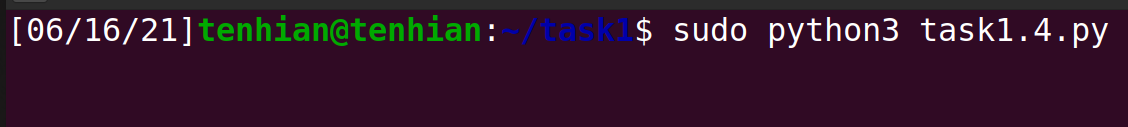
我们嗅探Kali(192.168.52.129)的所有的ICMP包，不论是发往任何IP，都回复源IP一个对应的ICMP-echo-reply，这就要求将截获的ICMP包，源MAC地址与目的MAC地址互换，源IP地址与目的IP地址互换，再计算新的IP包校验和和ICMP包校验和，更改ICMP包类型为echo-reply，最后发回源机器  
于是写出以下代码:

#!/usr/bin/python3  
from scapy.all import \*  
import json  
  
  
def spoof(pkt):  
 pkt.show()  
  
 a=Ether()  
 a.src=pkt[Ether].dst  
 a.dst=pkt[Ether].src  
 a.type=pkt[Ether].type  
  
 b=IP()  
 b.version=pkt[IP].version  
 b.ihl=pkt[IP].ihl  
 b.tos=pkt[IP].tos  
 b.len=pkt[IP].len  
 b.id=pkt[IP].id  
 b.ttl=128  
 # b.chksum=0  
 b.src=pkt[IP].dst #scr是源，dst是目的IP，将其互换，就变成了reply  
 b.dst=pkt[IP].src #伪装成dst  
   
 c=ICMP()  
 c.type="echo-reply" #将类型换成echo reply  
 c.code=0  
 c.chksum=0  
 c.id=pkt[ICMP].id  
 c.seq=pkt[ICMP].seq  
  
 d=pkt[Raw].load  
  
 p=a/b/c/d  
 #p.show()  
  
 s=Ether(p)  
  
 s.show()  
  
 sendp(s)  
  
pkt=sniff(iface='ens33',filter='icmp[icmptype]==icmp-echo && host 192.168.52.129',prn=spoof)#嗅探icmp request  
sniff\_spoof(pkt)

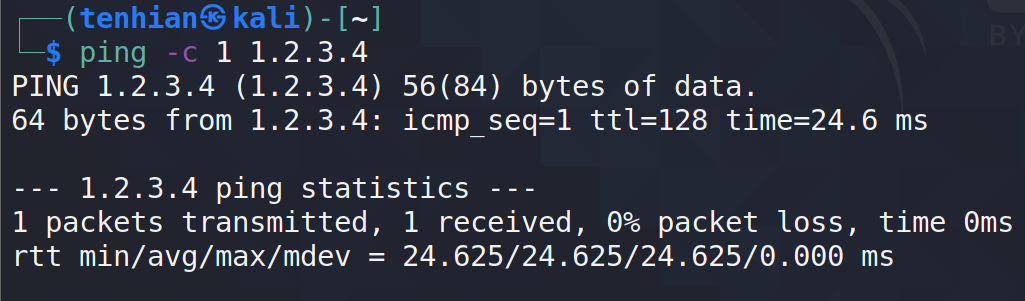
在Kali上执行 ping -c 1 1.2.3.4由于1.2.3.4这个IP是不存在的，所以ping不通



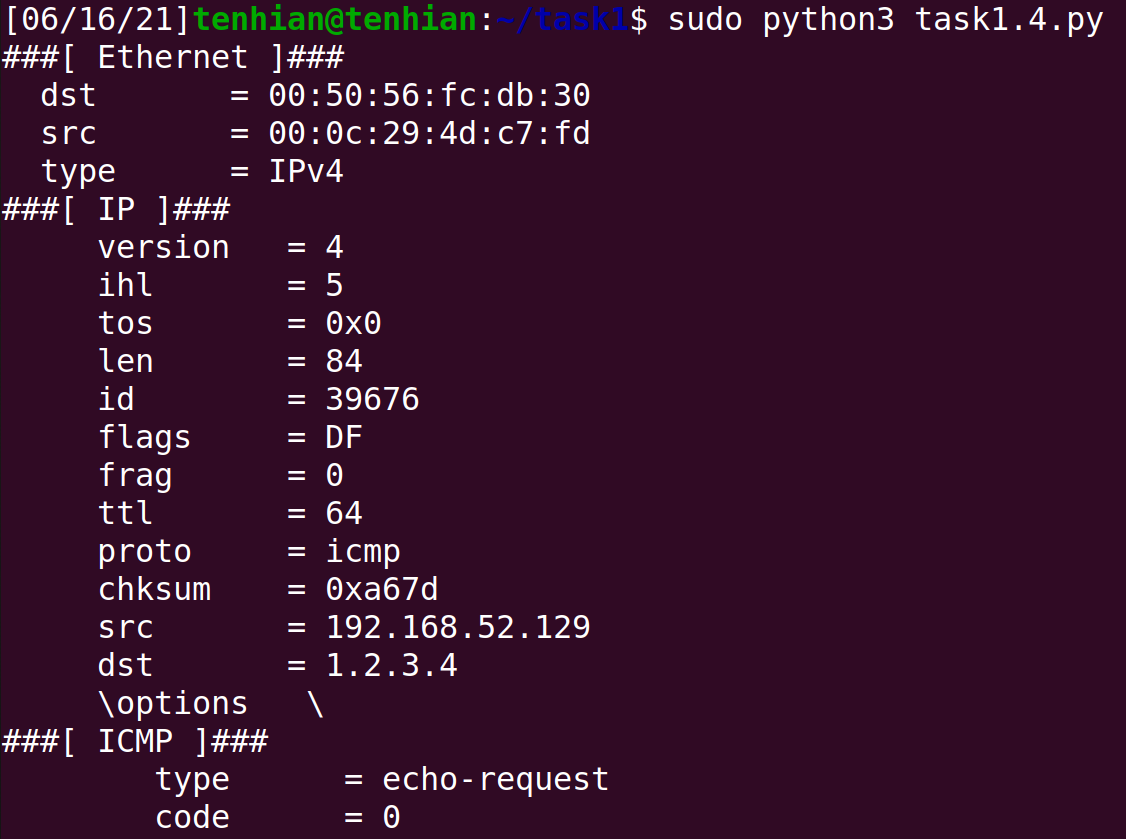
在Ubuntu上运行程序

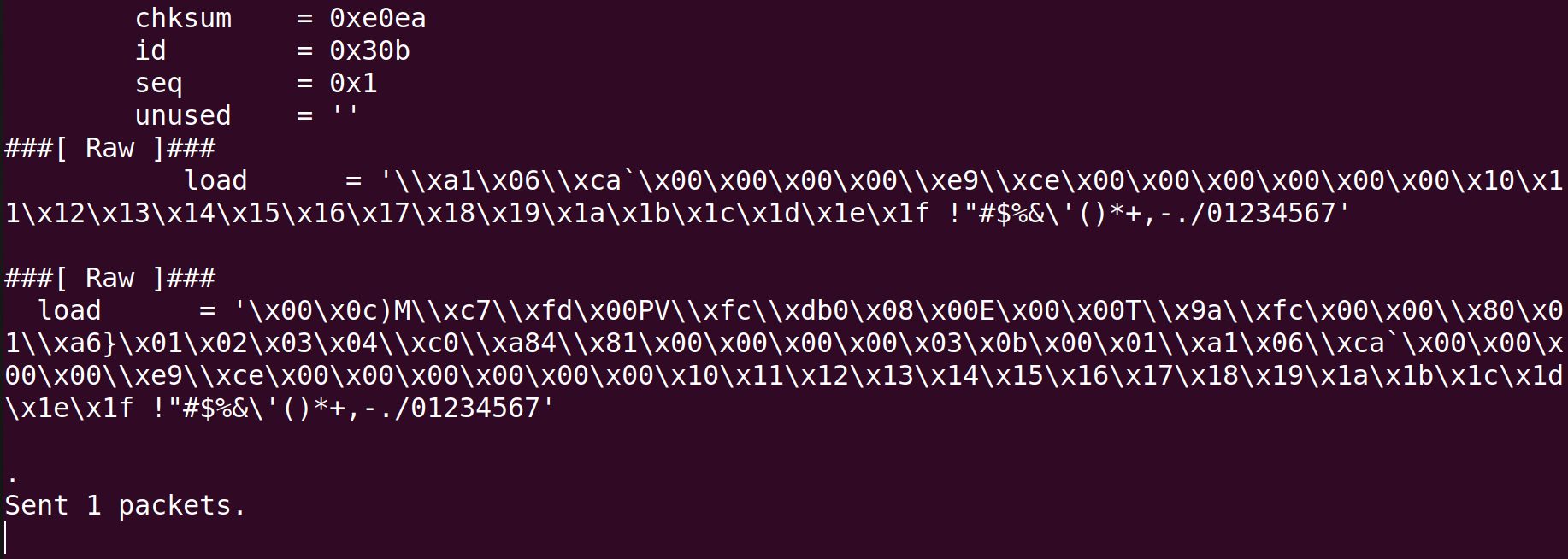


然后再在Kali上ping



现在就能ping通了，Ubuntu的程序上也显示了捕获和发送的包的内容





这种操作我们称为ICMP欺骗，task1.4完成

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

Task 1.1: Sniffing Packets