PROJECT IP-FIREWALL

using Python Language

Prepared by:

JAMAI OMAR

ELKHAYATI YASSER

BEROUISSAT MOHAMMED

SUMMARY

- Introduction.....3
- IP-FIREWALL methods.....4
- Part I : reading wireshark captures.....5
- Part II: live packet sniffer.....7
- Conclusion.....19

INTRODUCTION

The main purpose of a firewall is to separate a secure area from a less secure area and to control communications between the two. Firewalls can perform a variety of other functions, but are chiefly responsible for controlling inbound and outbound communications on anything from a single machine to an entire network.

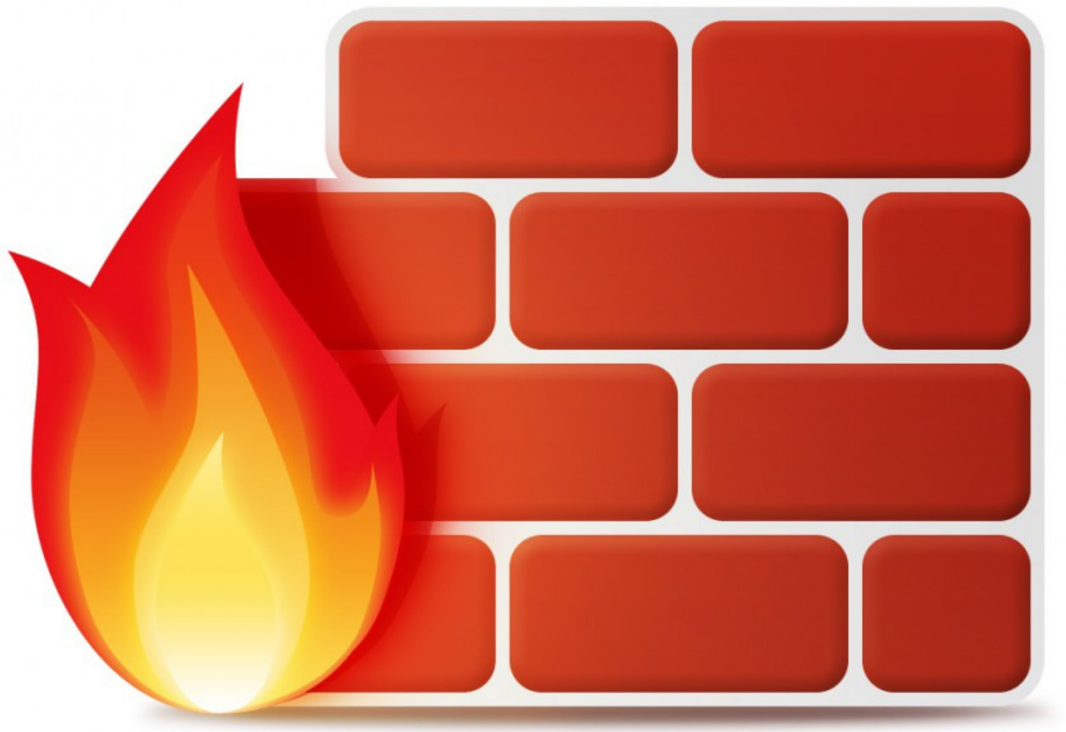
Packet-filtering firewalls operate at the network layer (Layer 3) of the OSI model. Packet-filtering firewalls make processing decisions based on network addresses, ports, or protocols.

Packet-filtering firewalls are very fast because there is not much logic going behind the decisions they make. They do not do any internal inspection of the traffic. They also do not store any state information. You have to manually open ports for all traffic that will flow through the firewall.

Packet-filtering firewalls are considered not to be very secure. This is because they will forward any traffic that is flowing on an approved port. So there could be malicious traffic being sent, but as long as it’s on an acceptable port, it will not be blocked.

IP-FIREWALL METHODS

how the firewall deals with packets?



Packet-filtering firewalls provide a way to filter IP addresses by either of two basic methods:

1.Allowing access to known IP addresses

2.Denying access to IP addresses and ports

By allowing access to known IP addresses, for example, you could allow access only to recognized, established IP addresses, or, you could deny access to all unknown or unrecognized IP addresses.

According to a report by CERT, it is most beneficial to utilize packet filtering techniques to permit only approved and known network traffic to the utmost degree possible. The use of packet filtering can be a very cost-effective means to add traffic control to an already existing router infrastructure.

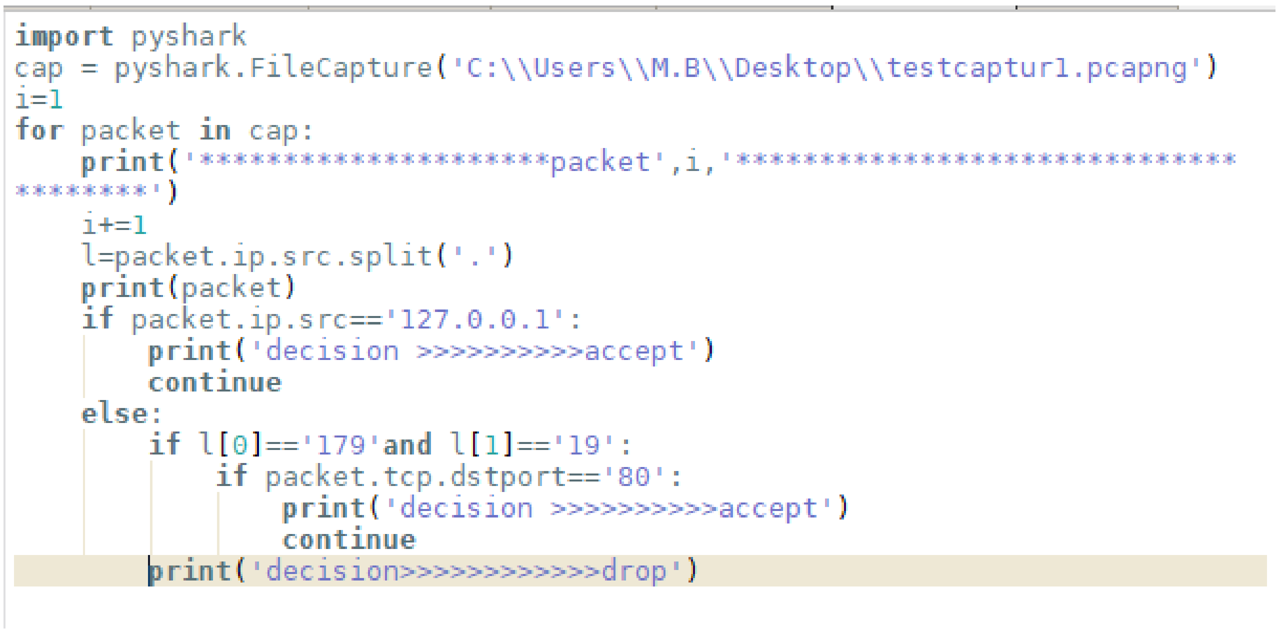
PART I: READING WIRESHARK CAPTURES

Wireshark is a free and open-source packet analyzer. It is used for network troubleshooting, analysis, software and communications protocol development, and education.

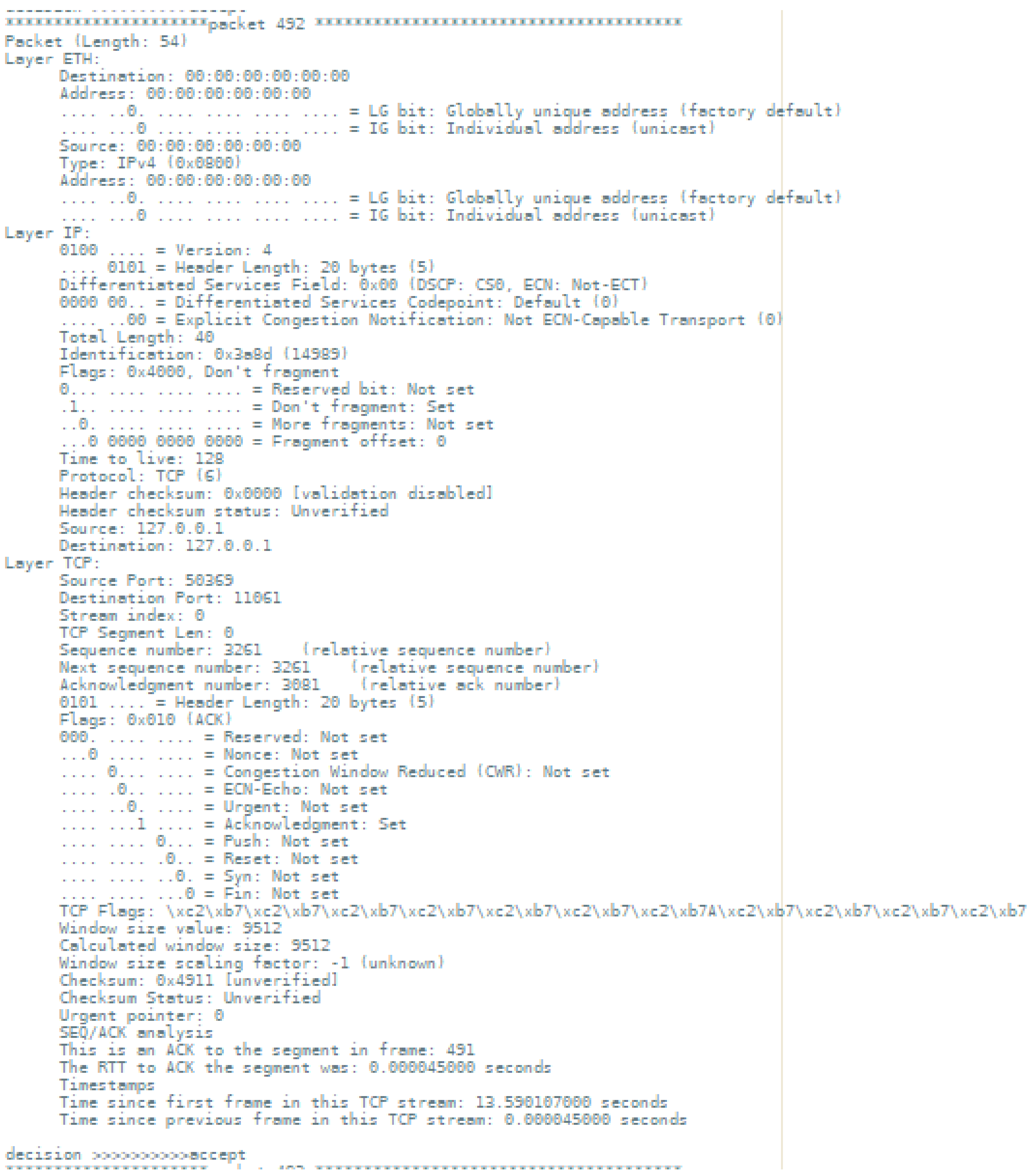
Wireshark lets the user put network interface controllers into promiscuous mode (if supported by the network interface controller), so they can see all the traffic visible on that interface and make captures of it.Thus making a very suitable starting point for our project.

We'll be using python's library Pyshark. Pyshark is a Python wrapper for tshark allowing python packet parsing using wireshark dissectors and extracting all the necessary information from the packets.

Below is an example of a code on reading a capture, filtering packets based on ip source address and making decision.



```
import pyshark
cap = pyshark.FileCapture('C:\\Users\\M.B\\Desktop\\testcapturl.pcapng')
i=1
for packet in cap:
    print('**********************packet',i,'*******************************
********')
    i+=1
    l=packet.ip.src.split('.')
    print(packet)
    if packet.ip.src=='127.0.0.1':
        print('decision >>>>>>>>>>accept')
        continue
    else:
        if l[0]=='179'and l[1]=='19':
            if packet.tcp.dstport=='80':
                print('decision >>>>>>>>>>accept')
                continue
        print('decision>>>>>>>>>>>>drop')
```



```
*********************packet 492 **************************************
Packet (Length: 54)
Layer ETH:
      Destination: 00:00:00:00:00:00
      Address: 00:00:00:00:00:00
      .... ..0. .... .... .... .... = LG bit: Globally unique address (factory default)
      .... ...0 .... .... .... .... = IG bit: Individual address (unicast)
      Source: 00:00:00:00:00:00
      Type: IPv4 (0x0800)
      Address: 00:00:00:00:00:00
      .... ..0. .... .... .... .... = LG bit: Globally unique address (factory default)
      .... ...0 .... .... .... .... = IG bit: Individual address (unicast)
Layer IP:
      0100 .... = Version: 4
      .... 0101 = Header Length: 20 bytes (5)
      Differentiated Services Field: 0x00 (DSCP: CS0, ECN: Not-ECT)
      0000 00.. = Differentiated Services Codepoint: Default (0)
      .... ..00 = Explicit Congestion Notification: Not ECN-Capable Transport (0)
      Total Length: 40
      Identification: 0x3a8d (14989)
      Flags: 0x4000, Don't fragment
      0... .... .... .... = Reserved bit: Not set
      .1.. .... .... .... = Don't fragment: Set
      ..0. .... .... .... = More fragments: Not set
      ...0 0000 0000 0000 = Fragment offset: 0
      Time to live: 128
      Protocol: TCP (6)
      Header checksum: 0x0000 [validation disabled]
      Header checksum status: Unverified
      Source: 127.0.0.1
      Destination: 127.0.0.1
Layer TCP:
      Source Port: 50369
      Destination Port: 11061
      Stream index: 0
      TCP Segment Len: 0
      Sequence number: 3261    (relative sequence number)
      Next sequence number: 3261    (relative sequence number)
      Acknowledgment number: 3081    (relative ack number)
      0101 .... = Header Length: 20 bytes (5)
      Flags: 0x010 (ACK)
      000. .... .... = Reserved: Not set
      ...0 .... .... = Nonce: Not set
      .... 0... .... = Congestion Window Reduced (CWR): Not set
      .... .0.. .... = ECN-Echo: Not set
      .... ..0. .... = Urgent: Not set
      .... ...1 .... = Acknowledgment: Set
      .... .... 0... = Push: Not set
      .... .... .0.. = Reset: Not set
      .... .... ..0. = Syn: Not set
      .... .... ...0 = Fin: Not set
      TCP Flags: \xc2\xb7\xc2\xb7\xc2\xb7\xc2\xb7\xc2\xb7\xc2\xb7\xc2\xb7A\xc2\xb7\xc2\xb7\xc2\xb7\xc2\xb7
      Window size value: 9512
      Calculated window size: 9512
      Window size scaling factor: -1 (unknown)
      Checksum: 0x4911 [unverified]
      Checksum Status: Unverified
      Urgent pointer: 0
      SEQ/ACK analysis
      This is an ACK to the segment in frame: 491
      The RTT to ACK the segment was: 0.000045000 seconds
      Timestamps
      Time since first frame in this TCP stream: 13.590107000 seconds
      Time since previous frame in this TCP stream: 0.000045000 seconds

decision >>>>>>>>>>accept
```

NOTE in this example , we followed two rules:

-Allowing local packets.

-allowing packets from the network '179.19.0.0' destined to tcp port 80

PART II: LIVE PACKET SNIFFER

- *What is Packet Sniffer?*

Sniffers are the special programs and tools that can capture network traffic packets from the network and then parse/ analyze them for various purposes. actually, sniffing tools have the ability to capture flowing data packets from networks. Data packets like TCP, UDP, ICMP etc. and after capturing these packets, sniffer also provides the facilities to extract these data packets and represent these packets in easy to understand interface. Well, There are many types of sniffers are available in the internet

- *How This Programs Going to Work?*

Actually, This Programs works on a Very Clear Concept. Every Client Interact With a Server Through Sending and receiving various types of data Packets like TCP, UDP etc. so, our program is going to capture all those data packets from our local computer network and then analyze and represent those packets in easy to understandable ways. In Simple Words, every networking service and networking program works on sending and receiving packet concept so what we need? is just to capture all traveling packets from our network.

- *How To Capture Packets?*

Of course, for this job we are going to use socket module. basically, socket module is the main player in our game because in python programming language socket module provides us the facility to play with network concept. so here for capturing packets, we are going to use socket.socket module.

For sniffing with socket module in python we have to create a socket.socket class object with special configuration. In simple words, we have to configure socket.socket class object to capture low-level packets from the network so that it can capture packet from low-level networks and provides us output without doing any type of changes in capture packets.

there is a small difference in Python socket module codes based on operating systems. Because Windows kernel works in a different way compared to Linux kernel

But how the programme can know if he is running on linux or windows ?
Thanks to the library os of python we have os.name wish tells us eactly what kind of operating system we are using

```
# if operating system is windows
if os.name == "nt":
    s = socket.socket(socket.AF_INET,socket.SOCK_RAW,socket.IPPROTO_IP)
    s.bind(("wlan0",0))
    s.setsockopt(socket.IPPROTO_IP,socket.IP_HDRINCL,1)
    s.ioctl(socket.SIO_RCVALL,socket.RCVALL_ON)

# if operating system is linux
else:
    s=socket.socket(socket.PF_PACKET, socket.SOCK_RAW, socket.ntohs(0x0800))
```

let's create a simplest packet sniffer script in python.
SimpleSniffer.py

```
import socket
import os
s=0
def windows():
    global s
    s = socket.socket(socket.AF_INET,socket.SOCK_RAW,socket.IPPROTO_IP)
    s.bind(("wlan0",0))
    s.setsockopt(socket.IPPROTO_IP,socket.IP_HDRINCL,1)
    s.ioctl(socket.SIO_RCVALL,socket.RCVALL_ON)
def linux():
    global s
    s = socket.socket(socket.AF_INET, socket.SOCK_RAW, socket.IPPROTO_TCP)
if os.name=="posix":
    linux()
else: windows()

while True:
   print s.recvfrom(65565)
```

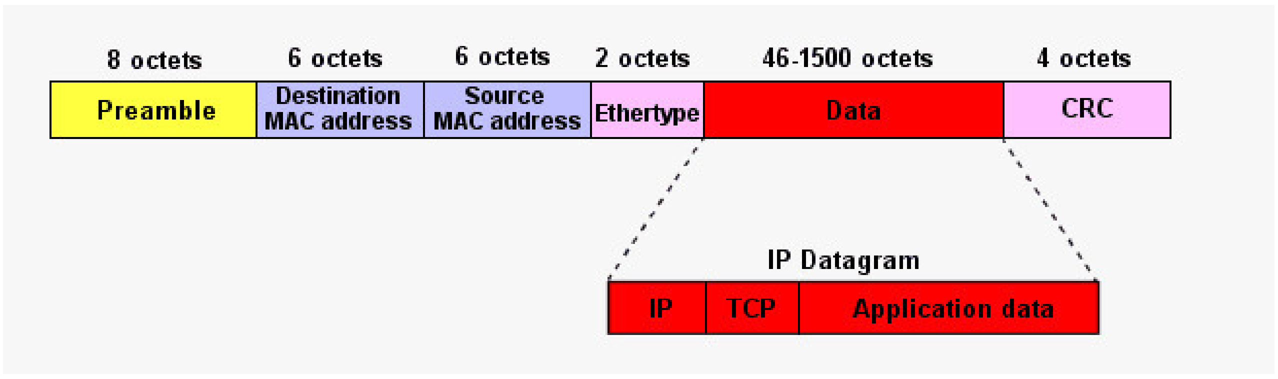
- ***Above Codes Explanation.***

We import the modules socket and os we have 2 fuctions depending on our operating system one of them would be working and finaly an infinite loop to capture and print.

- ***How To Parse/Extract Captured Packets?***

Actually, There Are Various Types Of Data Formats Are Available In Networking. But Here, I Am only going to describe few Important And Most Usable Data Formats. In Order To Understand These Data formats, Let's Take A Look At Data Structure Diagrams.

Ethernet Frame Format



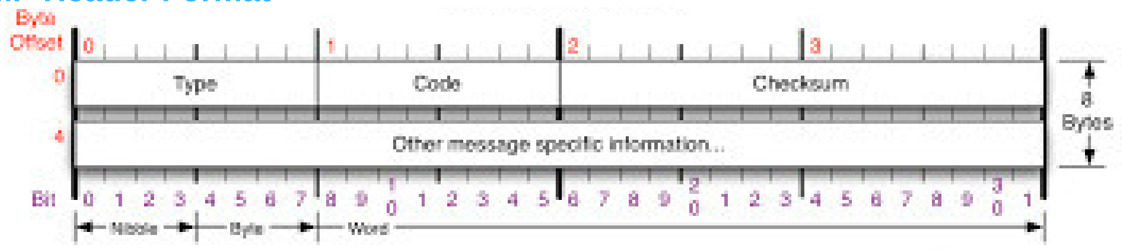
As you can see in Ethernet Frame Format Diagram There are more than 3 fields to extract but here, for this project we are only going to extract only 3 fields, Source Mac Address, Destination Mac Address and Ethernal Protocol Type.

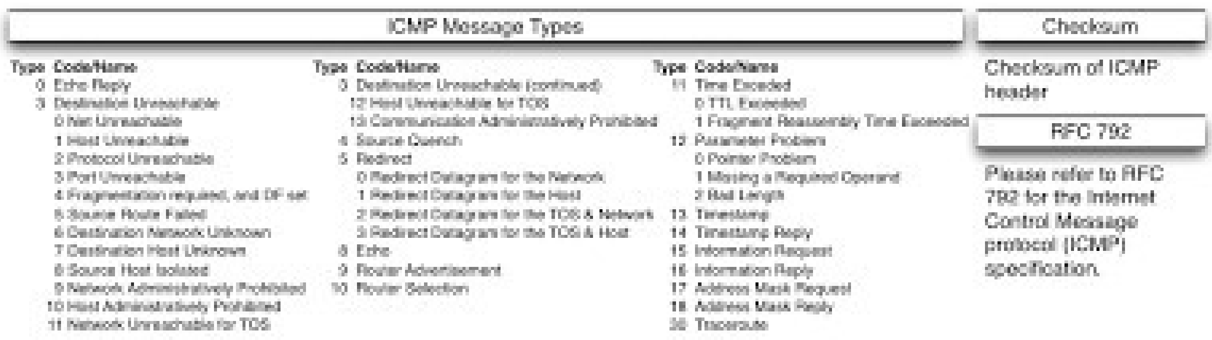
To Extract Source Address, Destination Address, and Ethernet Type Address, We have to use struct module which can unpack network packets for us. Basically, To Extract Data From Network Packets we have to pass an argument that going to represent field types, we want to extract in struct.unpack function.

Let's create afunctions to extract data from a frame and represent it in simplest form.

```
# Ethernet Header
   def eth_header(self, data):
       storeobj=data
       storeobj=struct.unpack("!6s6sH",storeobj)
       destination_mac=binascii.hexlify(storeobj[0])
       source_mac=binascii.hexlify(storeobj[1])
       eth_protocol=storeobj[2]
       data={"Destination Mac":destination_mac,
       "Source Mac":source_mac,
       "Protocol":eth_protocol}
       return data
```

ICMP Header Format





ICMP Message Types

Type	Code/Name
0	Echo Reply
3	Destination Unreachable
	0 Net Unreachable
	1 Host Unreachable
	2 Protocol Unreachable
	3 Port Unreachable
	4 Fragmentation required, and DF set
	5 Source Route Failed
	6 Destination Network Unknown
	7 Destination Host Unknown
	8 Source Host Isolated
	9 Network Administratively Prohibited
	10 Host Administratively Prohibited
	11 Network Unreachable for TOS
3	Destination Unreachable (continued)
	12 Host Unreachable for TOS
	13 Communication Administratively Prohibited
4	Source Quench
5	Redirect
	0 Redirect Datagram for the Network
	1 Redirect Datagram for the Host
	2 Redirect Datagram for the TOS & Network
	3 Redirect Datagram for the TOS & Host
8	Echo
9	Router Advertisement
10	Router Selection
11	Time Exceeded
	0 TTL Exceeded
	1 Fragment Reassembly Time Exceeded
12	Parameter Problem
	0 Pointer Problem
	1 Missing a Required Operand
	2 Bad Length
13	Timestamp
14	Timestamp Reply
15	Information Request
16	Information Reply
17	Address Mask Request
18	Address Mask Reply
30	Traceroute

Checksum

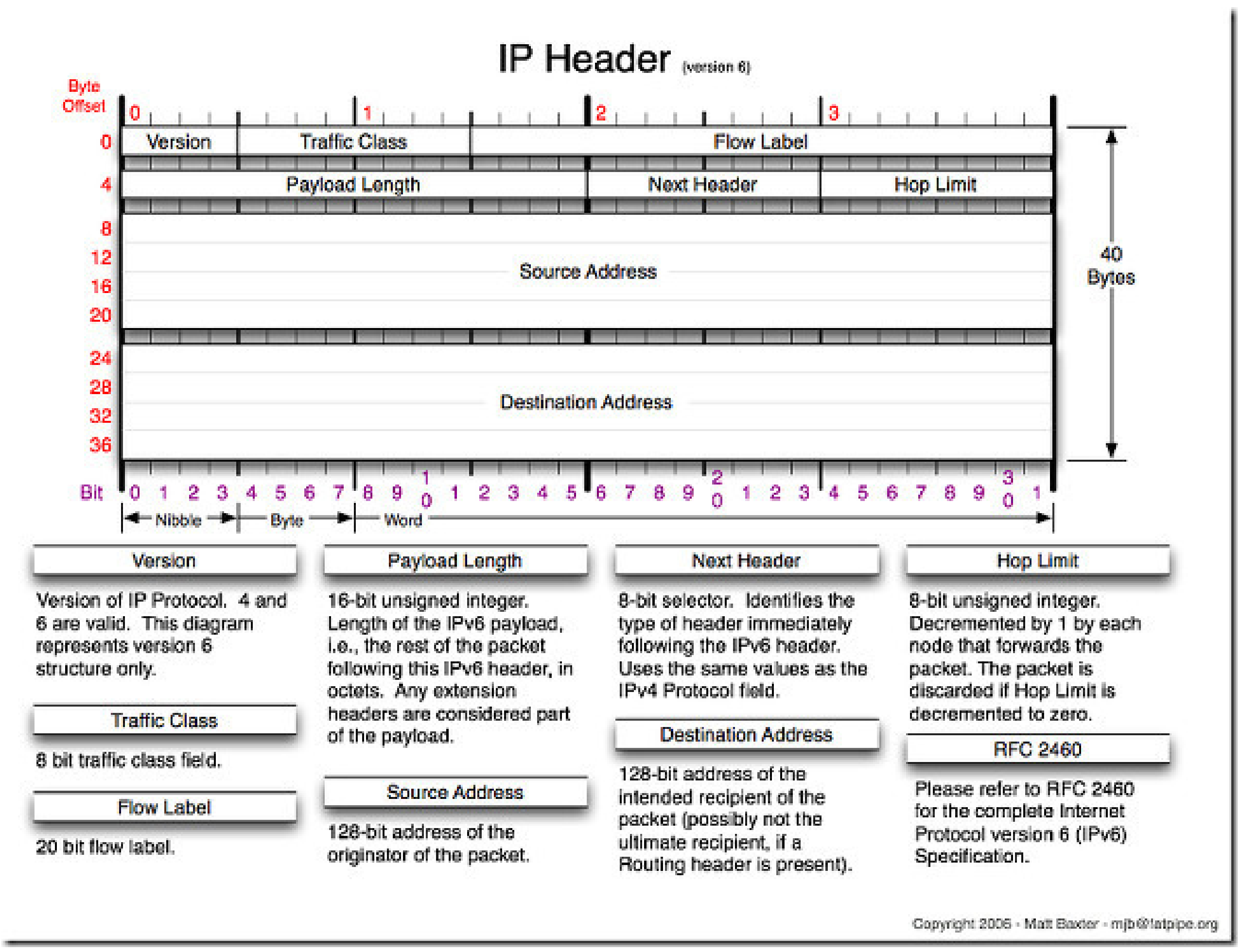
Checksum of ICMP header

RFC 792

Please refer to RFC 792 for the Internet Control Message protocol (ICMP) specification.

```
# ICMP HEADER Extraction
    def icmp_header(self, data):
        icmph=struct.unpack('!BBH', data)
        icmp_type = icmph[0]
        code = icmph[1]
        checksum = icmph[2]
        data={'ICMP Type':icmp_type,
        "Code":code,
        "CheckSum":checksum}
        return data
```

IP Header Format



Version

Version of IP Protocol. 4 and 6 are valid. This diagram represents version 6 structure only.

Traffic Class

8 bit traffic class field.

Flow Label

20 bit flow label.

Payload Length

16-bit unsigned integer. Length of the IPv6 payload, i.e., the rest of the packet following this IPv6 header, in octets. Any extension headers are considered part of the payload.

Source Address

128-bit address of the originator of the packet.

Next Header

8-bit selector. Identifies the type of header immediately following the IPv6 header. Uses the same values as the IPv4 Protocol field.

Destination Address

128-bit address of the intended recipient of the packet (possibly not the ultimate recipient, if a Routing header is present).

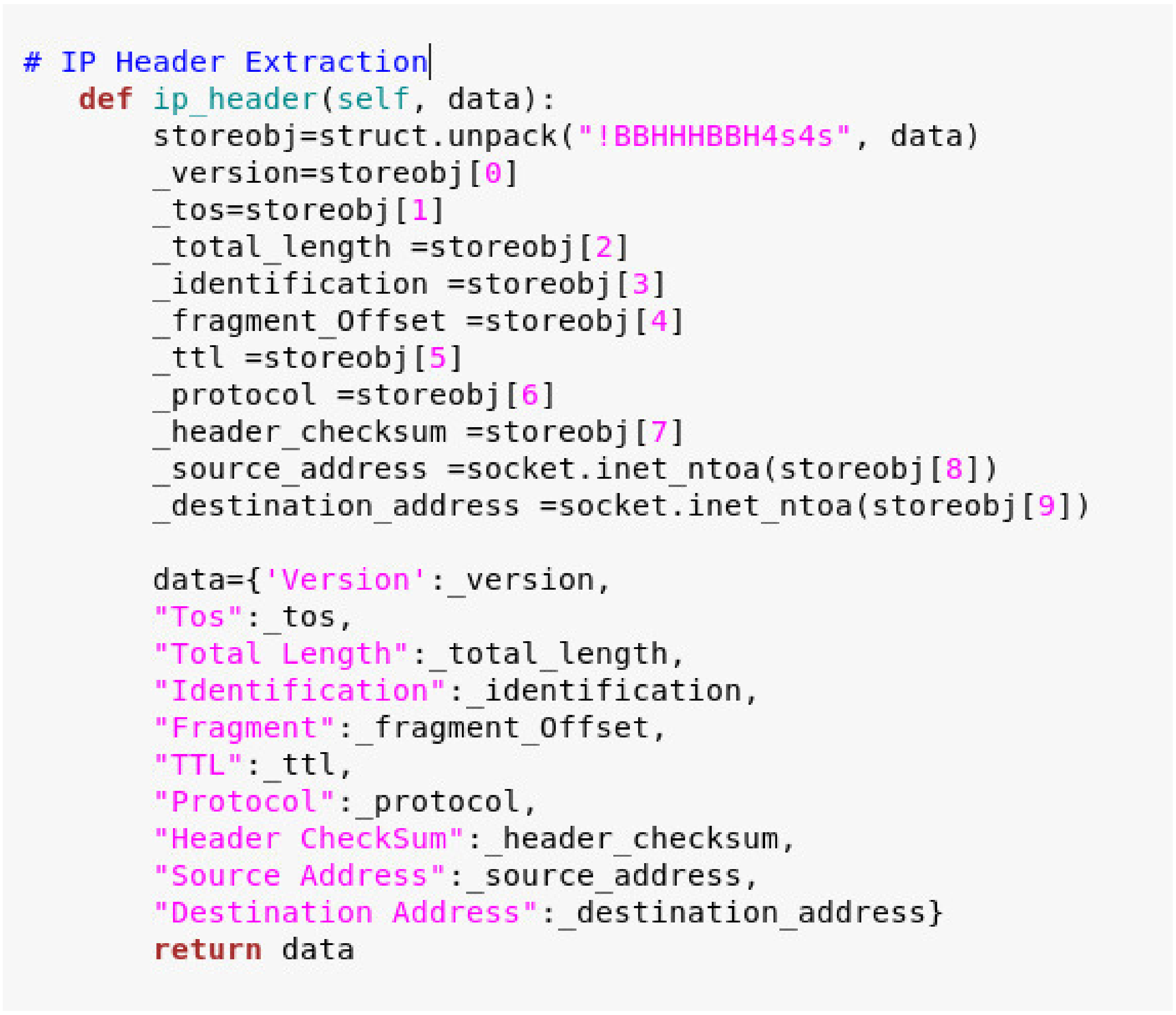
Hop Limit

8-bit unsigned integer. Decremented by 1 by each node that forwards the packet. The packet is discarded if Hop Limit is decremented to zero.

RFC 2460

Please refer to RFC 2460 for the complete Internet Protocol version 6 (IPv6) Specification.

Copyright 2006 - Matt Baxter - mjb@fatpipe.org



```python
# IP Header Extraction
    def ip_header(self, data):
        storeobj=struct.unpack("!BBHHHBBH4s4s", data)
        _version=storeobj[0]
        _tos=storeobj[1]
        _total_length =storeobj[2]
        _identification =storeobj[3]
        _fragment_Offset =storeobj[4]
        _ttl =storeobj[5]
        _protocol =storeobj[6]
        _header_checksum =storeobj[7]
        _source_address =socket.inet_ntoa(storeobj[8])
        _destination_address =socket.inet_ntoa(storeobj[9])

        data={'Version':_version,
        "Tos":_tos,
        "Total Length":_total_length,
        "Identification":_identification,
        "Fragment":_fragment_Offset,
        "TTL":_ttl,
        "Protocol":_protocol,
        "Header CheckSum":_header_checksum,
        "Source Address":_source_address,
        "Destination Address":_destination_address}
        return data
```

from above examples you got the basic idea how our program is exactly going to work. Actually, Now we are going to assemble all above previewed functions in one program so that our program can extract various types of informations during sniffing. well here to make this project easy to understand i am going to divide our project in two script.

First script is for sniffing packets and another script for extracting data from captured packets.

So, let's Start our Python Packet Sniffer Coding.

For This Purpose, We will Create 2 Script.

1. For Capturing Packets (pypackets.py)

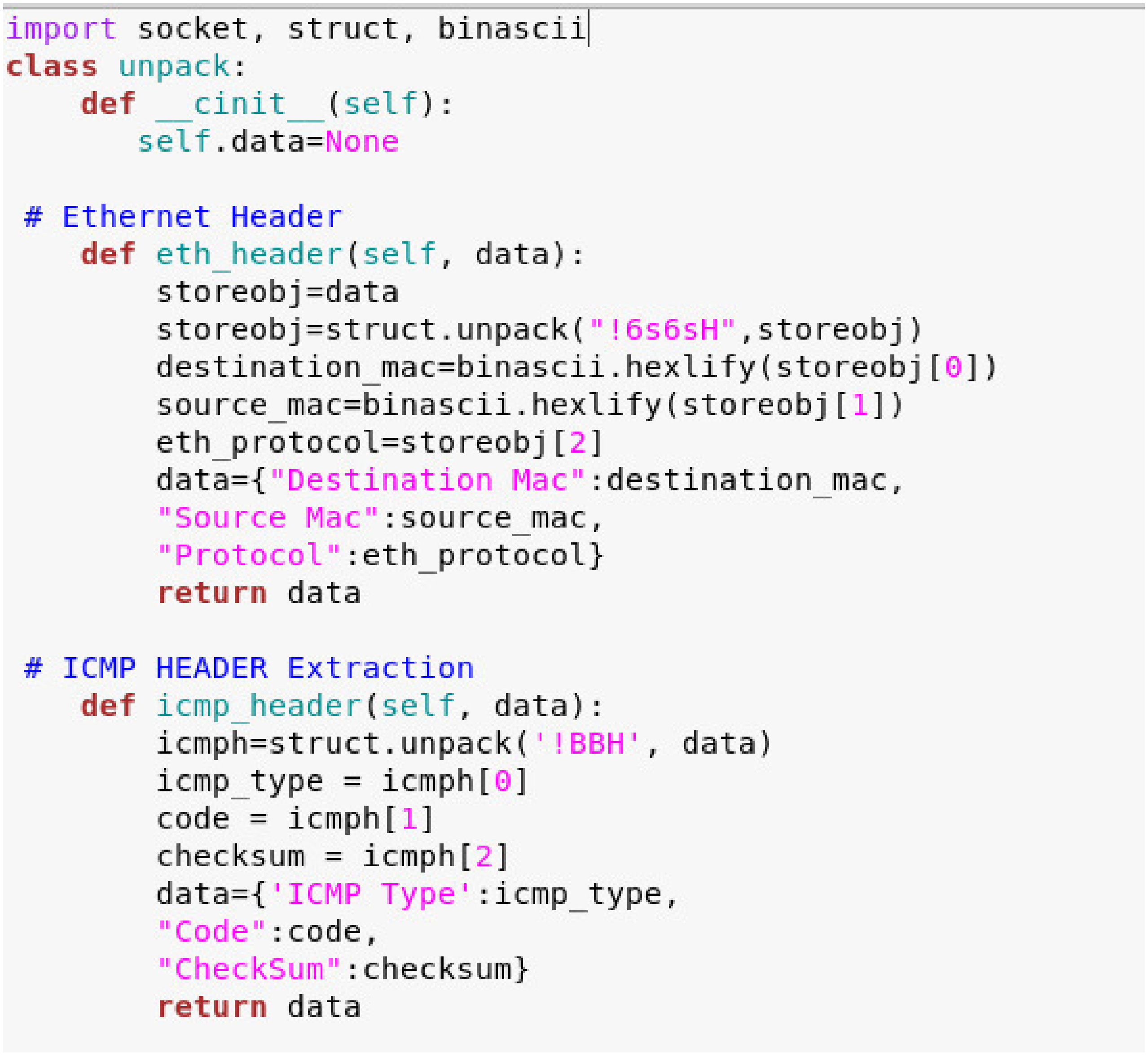
2. For Extracting Captured Data (pye.py)

So Here it's Our Demo Codes:

1. For Capturing Packets

```
import socket
import struct
import binascii
import os
import pye
# if operating system is windows
if os.name == "nt":
    s = socket.socket(socket.AF_INET,socket.SOCK_RAW,socket.IPPROTO_IP)
    s.bind(("YOUR_INTERFACE_IP",0))
    s.setsockopt(socket.IPPROTO_IP,socket.IP_HDRINCL,1)
    s.ioctl(socket.SIO_RCVALL,socket.RCVALL_ON)

# if operating system is linux
else:
    s=socket.socket(socket.PF_PACKET, socket.SOCK_RAW, socket.ntohs(0x0800))

# create loop
while True:
    # Capture packets from network
    pkt=s.recvfrom(65565)
    # extract packets with the help of pye.unpack class
    unpack=pye.unpack()
    print "\n\n===&gt;&gt; [+] ------------ Ethernet Header----- [+]"
    # print data on terminal
    for i in unpack.eth_header(pkt[0][0:14]).iteritems():
        a,b=i
        print "{} : {} | ".format(a,b),
    print "\n===&gt;&gt; [+] ------------ IP Header ------------[+]"
    for i in unpack.ip_header(pkt[0][14:34]).iteritems():
        a,b=i
        print "{} : {} | ".format(a,b),
    print "\n===&gt;&gt; [+] ------------ Tcp Header ----------- [+]"
    for  i in unpack.tcp_header(pkt[0][34:54]).iteritems():
        a,b=i
        print "{} : {} | ".format(a,b),
```

2.For Extracting Captured Data (pye.py)



```
import socket, struct, binascii
class unpack:
    def __cinit__(self):
       self.data=None

 # Ethernet Header
    def eth_header(self, data):
        storeobj=data
        storeobj=struct.unpack("!6s6sH",storeobj)
        destination_mac=binascii.hexlify(storeobj[0])
        source_mac=binascii.hexlify(storeobj[1])
        eth_protocol=storeobj[2]
        data={"Destination Mac":destination_mac,
        "Source Mac":source_mac,
        "Protocol":eth_protocol}
        return data

 # ICMP HEADER Extraction
    def icmp_header(self, data):
        icmph=struct.unpack('!BBH', data)
        icmp_type = icmph[0]
        code = icmph[1]
        checksum = icmph[2]
        data={'ICMP Type':icmp_type,
        "Code":code,
        "CheckSum":checksum}
        return data
```

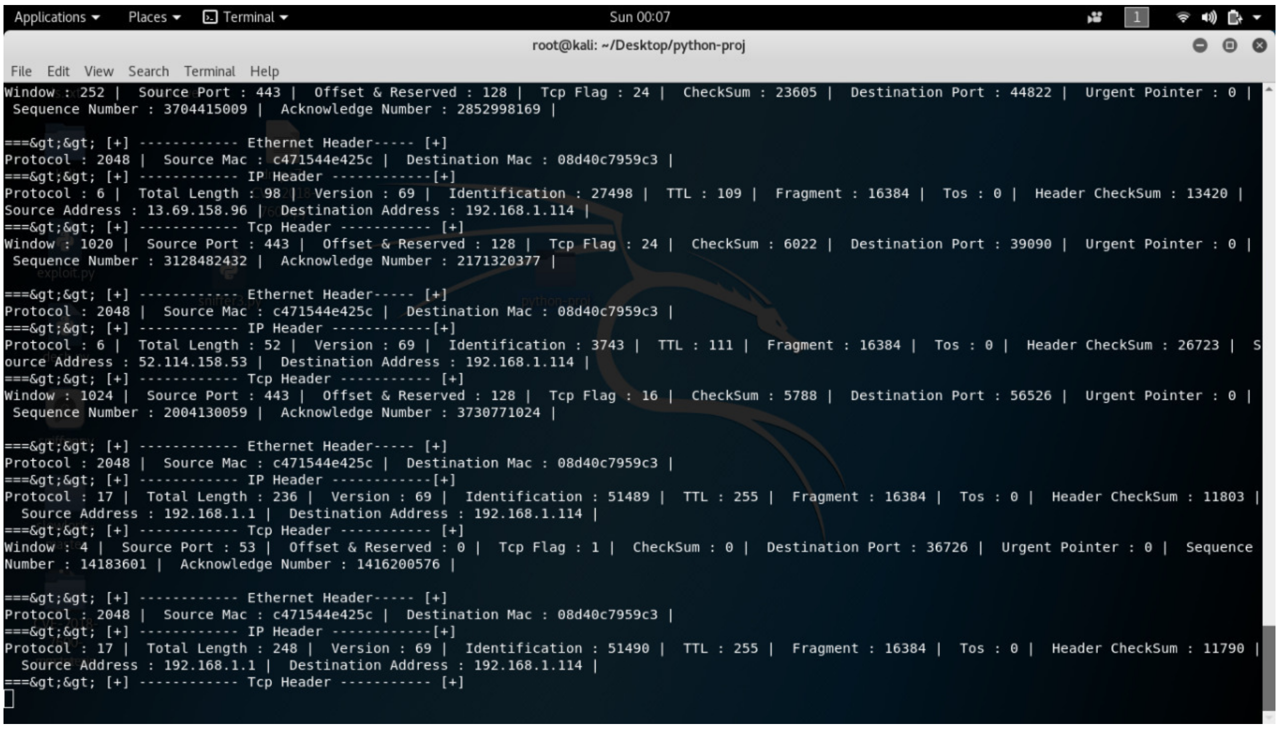
```
# UDP Header Extraction
   def udp_header(self, data):
       storeobj=struct.unpack('!HHHH', data)
       source_port = storeobj[0]
       dest_port = storeobj[1]
       length = storeobj[2]
       checksum = storeobj[3]
       data={"Source Port":source_port,
       "Destination Port":dest_port,
       "Length":length,
       "CheckSum":checksum}
       return data

# IP Header Extraction
   def ip_header(self, data):
       storeobj=struct.unpack("!BBHHHBBH4s4s", data)
       _version=storeobj[0]
       _tos=storeobj[1]
       _total_length =storeobj[2]
       _identification =storeobj[3]
       _fragment_Offset =storeobj[4]
       _ttl =storeobj[5]
       _protocol =storeobj[6]
       _header_checksum =storeobj[7]
       _source_address =socket.inet_ntoa(storeobj[8])
       _destination_address =socket.inet_ntoa(storeobj[9])

       data={'Version':_version,
       "Tos":_tos,
       "Total Length":_total_length,
       "Identification":_identification,
       "Fragment":_fragment_Offset,
       "TTL":_ttl,
       "Protocol":_protocol,
       "Header CheckSum":_header_checksum
```

```
        "Protocol":_protocol,
        "Header CheckSum":_header_checksum,
        "Source Address":_source_address,
        "Destination Address":_destination_address}
        return data

# Tcp Header Extraction
    def tcp_header(self, data):
        storeobj=struct.unpack('!HHLLBBHHH',data)
        _source_port =storeobj[0]
        _destination_port  =storeobj[1]
        _sequence_number  =storeobj[2]
        _acknowledge_number  =storeobj[3]
        _offset_reserved  =storeobj[4]
        _tcp_flag  =storeobj[5]
        _window  =storeobj[6]
        _checksum  =storeobj[7]
        _urgent_pointer =storeobj[8]
        data={"Source Port":_source_port,
        "Destination Port":_destination_port,
        "Sequence Number":_sequence_number,
        "Acknowledge Number":_acknowledge_number,
        "Offset & Reserved":_offset_reserved,
        "Tcp Flag":_tcp_flag,
        "Window":_window,
        "CheckSum":_checksum,
        "Urgent Pointer":_urgent_pointer
        }
        return data
```

```
# Mac Address Formating
    def mac_formater(a):
        b = "%.2x:%.2x:%.2x:%.2x:%.2x:%.2x" % (ord(a[0]), ord(a[1]), ord(a[2]), ord(a[3]), ord(a[4]) , ord(a[5]))
        return b

    def get_host(q):
        try:
            nk=socket.gethostbyaddr(q)
        except:
            k='Unknown'
        return k
```

Here, Above Codes Will Extract Provided Data Packets According To Their Specified Format.

Hence, Now Our Code Finished Here. so, let's see how it's working. For This Demo Trial here, I am using kali linux.


Applications
Places
Terminal
Sun 00:07
root@kali: ~/Desktop/python-proj
File Edit View Search Terminal Help

```
Window : 252 |  Source Port : 443 |  Offset & Reserved : 128 |  Tcp Flag : 24 |  CheckSum : 23605 |  Destination Port : 44822 |  Urgent Pointer : 0 |
 Sequence Number : 3704415009 |  Acknowledge Number : 2852998169 |

===&gt;&gt; [+] ------------ Ethernet Header----- [+]
Protocol : 2048 |  Source Mac : c471544e425c |  Destination Mac : 08d40c7959c3 |
===&gt;&gt; [+] ------------ IP Header ------------[+]
Protocol : 6 |  Total Length : 98 | Version : 69 |  Identification : 27498 |  TTL : 109 |  Fragment : 16384 |  Tos : 0 |  Header CheckSum : 13420 |
Source Address : 13.69.158.96 |  Destination Address : 192.168.1.114 |
===&gt;&gt; [+] ------------ Tcp Header ----------- [+]
Window : 1020 |  Source Port : 443 |  Offset & Reserved : 128 |  Tcp Flag : 24 |  CheckSum : 6022 |  Destination Port : 39090 |  Urgent Pointer : 0 |
 Sequence Number : 3128482432 |  Acknowledge Number : 2171320377 |

===&gt;&gt; [+] ------------ Ethernet Header----- [+]
Protocol : 2048 |  Source Mac : c471544e425c |  Destination Mac : 08d40c7959c3 |
===&gt;&gt; [+] ------------ IP Header ------------[+]
Protocol : 6 |  Total Length : 52 |  Version : 69 |  Identification : 3743 |  TTL : 111 |  Fragment : 16384 |  Tos : 0 |  Header CheckSum : 26723 |  S
ource Address : 52.114.158.53 |  Destination Address : 192.168.1.114 |
===&gt;&gt; [+] ------------ Tcp Header ----------- [+]
Window : 1024 |  Source Port : 443 |  Offset & Reserved : 128 |  Tcp Flag : 16 |  CheckSum : 5788 |  Destination Port : 56526 |  Urgent Pointer : 0 |
 Sequence Number : 2004130059 |  Acknowledge Number : 3730771024 |

===&gt;&gt; [+] ------------ Ethernet Header----- [+]
Protocol : 2048 |  Source Mac : c471544e425c |  Destination Mac : 08d40c7959c3 |
===&gt;&gt; [+] ------------ IP Header ------------[+]
Protocol : 17 |  Total Length : 236 |  Version : 69 |  Identification : 51489 |  TTL : 255 |  Fragment : 16384 |  Tos : 0 |  Header CheckSum : 11803 |
  Source Address : 192.168.1.1 |  Destination Address : 192.168.1.114 |
===&gt;&gt; [+] ------------ Tcp Header ----------- [+]
Window : 4 |  Source Port : 53 |  Offset & Reserved : 0 |  Tcp Flag : 1 |  CheckSum : 0 |  Destination Port : 36726 |  Urgent Pointer : 0 |  Sequence
Number : 14183601 |  Acknowledge Number : 1416200576 |

===&gt;&gt; [+] ------------ Ethernet Header----- [+]
Protocol : 2048 |  Source Mac : c471544e425c |  Destination Mac : 08d40c7959c3 |
===&gt;&gt; [+] ------------ IP Header ------------[+]
Protocol : 17 |  Total Length : 248 |  Version : 69 |  Identification : 51490 |  TTL : 255 |  Fragment : 16384 |  Tos : 0 |  Header CheckSum : 11790 |
  Source Address : 192.168.1.1 |  Destination Address : 192.168.1.114 |
===&gt;&gt; [+] ------------ Tcp Header ----------- [+]
```

OUR GUI

At this point we decided to creat a GUI for our sniffer and add some filtering tools to control the trafic.

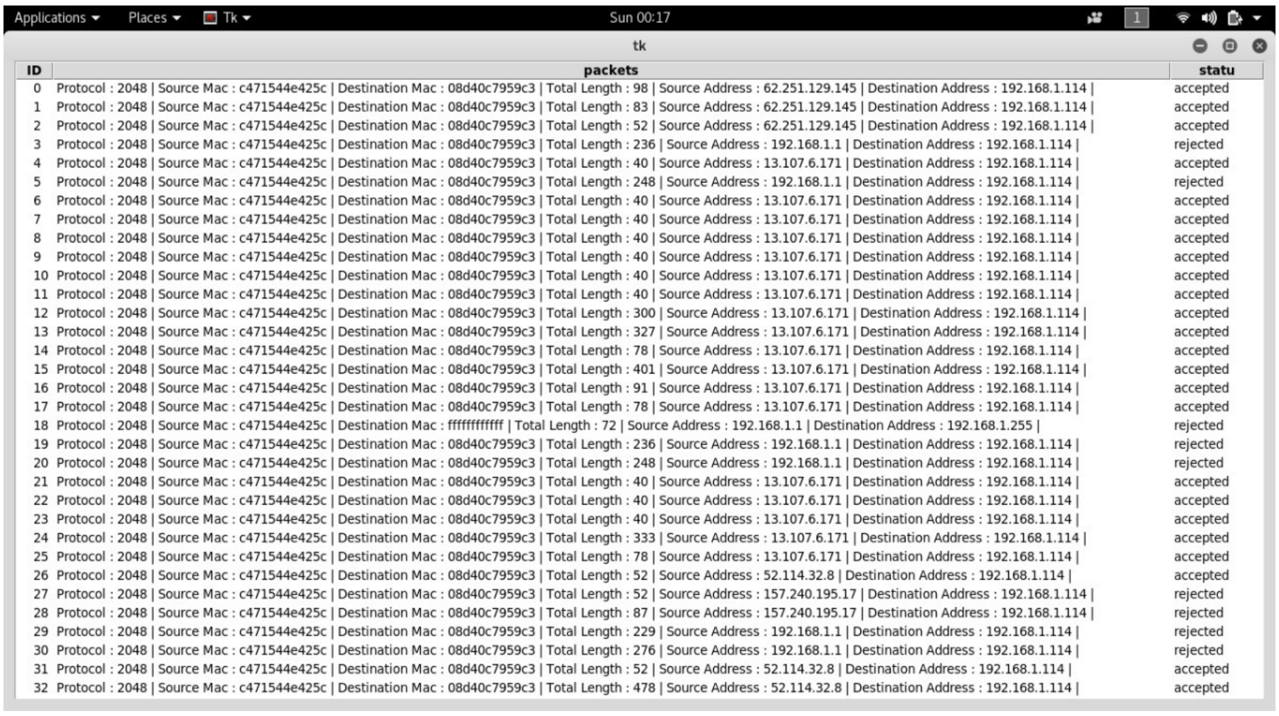
We did exactly the same thing as before but this time we added a fucntion that defines the ip addresses wish are not allowed as a source and we creat o gui using tkinter treeview.

```
root = Tk()

tv = ttk.Treeview(root,height = 33,selectmode = "extended")


tv.heading('#0' , text='ID')
tv.configure(column=('#packets','#statu'))
tv.heading('#packets' , text='packets')
tv.heading('#statu' , text='statu')
tv.column("#packets",minwidth=0,width=1200)
tv.column("#statu",minwidth=0,width=100)
tv.column("#0",minwidth=0,width=40)


def not_allowed_ip():
    a=["157.240.195.17","172.217.18.228","192.168.1.1"]
    return(a)
```

```
for k in range(200):

    # Capture packets from network
    pkt=s.recvfrom(65565)


    # extract packets with the help of pye.unpack class
    unpack=pye.unpack()

    ch=""
    n=1
    for i in unpack.eth_header(pkt[0][0:14]).iteritems():
        a,b=i
        ch=ch+str("{} : {} | ".format(a,b),)
    for i in unpack.ip_header(pkt[0][14:34]).iteritems():
        a,b=i
        if a=="Total Length" or a=="Source Address" or a=="Destination Address" :
            ch=ch+ str("{} : {} | ".format(a,b),)
        if a=="Source Address":
            if str(b) in not_allowed_ip():
                statu="rejected"
            else:
                statu="accepted"

    tv.insert('' , 'end' , '#{}'.format(str(k)) , text=str(k))
    tv.set('#{}'.format(str(k)), '#packets' , ch)

    tv.set('#{}'.format(str(k)), '#statu' , statu)
tv.pack()
root.mainloop()
```

But here we can see only 200 frame because the loop that add information to the tree will run only 200 times as defined in the code . We couldn’t provide a live adding of frames

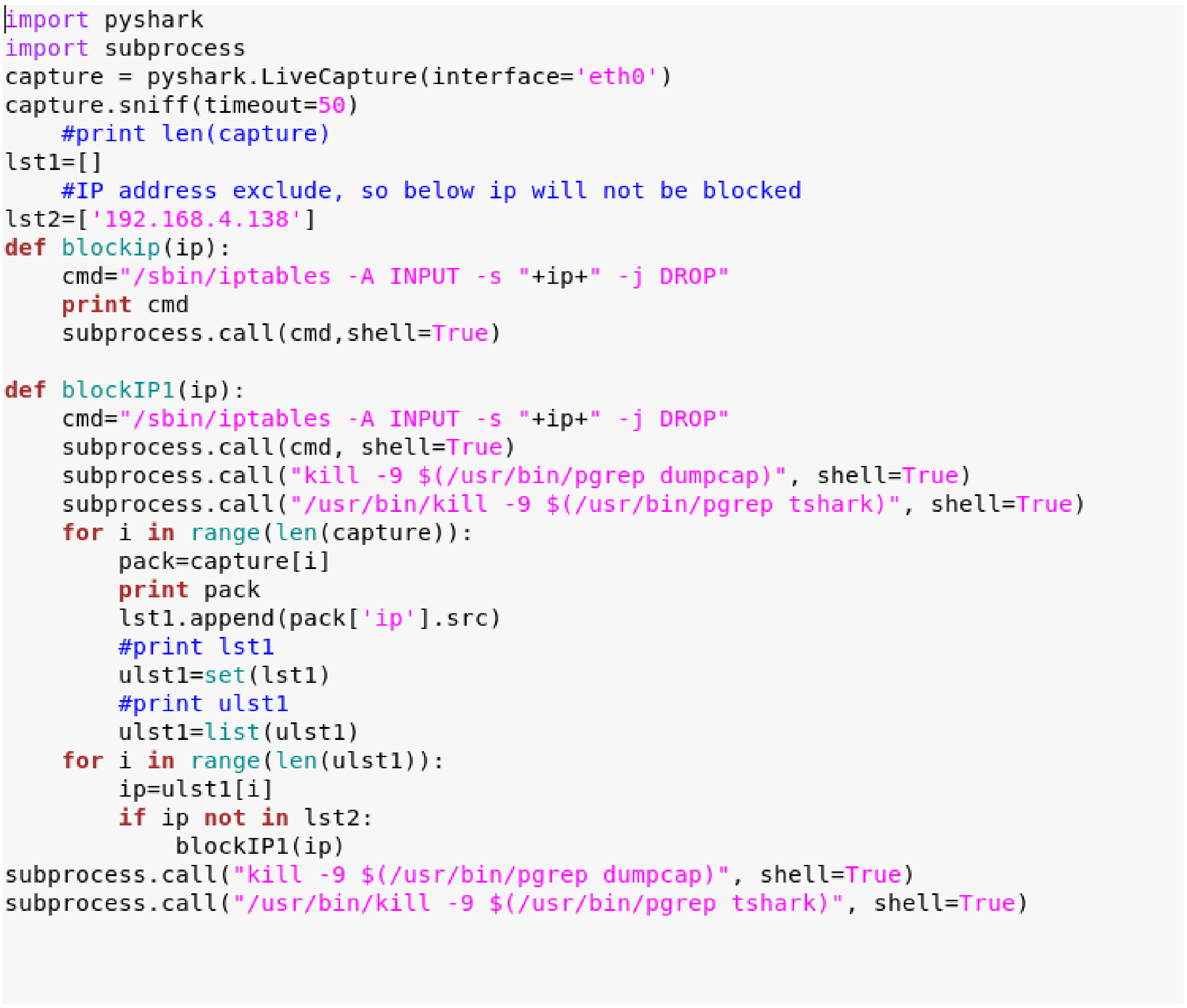
The final result



```
ID                                                                 packets                                                                              statu
0  Protocol : 2048 | Source Mac : c471544e425c | Destination Mac : 08d40c7959c3 | Total Length : 98 | Source Address : 62.251.129.145 | Destination Address : 192.168.1.114 |   accepted
1  Protocol : 2048 | Source Mac : c471544e425c | Destination Mac : 08d40c7959c3 | Total Length : 83 | Source Address : 62.251.129.145 | Destination Address : 192.168.1.114 |   accepted
2  Protocol : 2048 | Source Mac : c471544e425c | Destination Mac : 08d40c7959c3 | Total Length : 52 | Source Address : 62.251.129.145 | Destination Address : 192.168.1.114 |   accepted
3  Protocol : 2048 | Source Mac : c471544e425c | Destination Mac : 08d40c7959c3 | Total Length : 236 | Source Address : 192.168.1.1 | Destination Address : 192.168.1.114 |     rejected
4  Protocol : 2048 | Source Mac : c471544e425c | Destination Mac : 08d40c7959c3 | Total Length : 40 | Source Address : 13.107.6.171 | Destination Address : 192.168.1.114 |     accepted
5  Protocol : 2048 | Source Mac : c471544e425c | Destination Mac : 08d40c7959c3 | Total Length : 248 | Source Address : 192.168.1.1 | Destination Address : 192.168.1.114 |     rejected
6  Protocol : 2048 | Source Mac : c471544e425c | Destination Mac : 08d40c7959c3 | Total Length : 40 | Source Address : 13.107.6.171 | Destination Address : 192.168.1.114 |     accepted
7  Protocol : 2048 | Source Mac : c471544e425c | Destination Mac : 08d40c7959c3 | Total Length : 40 | Source Address : 13.107.6.171 | Destination Address : 192.168.1.114 |     accepted
8  Protocol : 2048 | Source Mac : c471544e425c | Destination Mac : 08d40c7959c3 | Total Length : 40 | Source Address : 13.107.6.171 | Destination Address : 192.168.1.114 |     accepted
9  Protocol : 2048 | Source Mac : c471544e425c | Destination Mac : 08d40c7959c3 | Total Length : 40 | Source Address : 13.107.6.171 | Destination Address : 192.168.1.114 |     accepted
10 Protocol : 2048 | Source Mac : c471544e425c | Destination Mac : 08d40c7959c3 | Total Length : 40 | Source Address : 13.107.6.171 | Destination Address : 192.168.1.114 |     accepted
11 Protocol : 2048 | Source Mac : c471544e425c | Destination Mac : 08d40c7959c3 | Total Length : 40 | Source Address : 13.107.6.171 | Destination Address : 192.168.1.114 |     accepted
12 Protocol : 2048 | Source Mac : c471544e425c | Destination Mac : 08d40c7959c3 | Total Length : 300 | Source Address : 13.107.6.171 | Destination Address : 192.168.1.114 |    accepted
13 Protocol : 2048 | Source Mac : c471544e425c | Destination Mac : 08d40c7959c3 | Total Length : 327 | Source Address : 13.107.6.171 | Destination Address : 192.168.1.114 |    accepted
14 Protocol : 2048 | Source Mac : c471544e425c | Destination Mac : 08d40c7959c3 | Total Length : 78 | Source Address : 13.107.6.171 | Destination Address : 192.168.1.114 |     accepted
15 Protocol : 2048 | Source Mac : c471544e425c | Destination Mac : 08d40c7959c3 | Total Length : 401 | Source Address : 13.107.6.171 | Destination Address : 192.168.1.114 |    accepted
16 Protocol : 2048 | Source Mac : c471544e425c | Destination Mac : 08d40c7959c3 | Total Length : 91 | Source Address : 13.107.6.171 | Destination Address : 192.168.1.114 |     accepted
17 Protocol : 2048 | Source Mac : c471544e425c | Destination Mac : 08d40c7959c3 | Total Length : 78 | Source Address : 13.107.6.171 | Destination Address : 192.168.1.114 |     accepted
18 Protocol : 2048 | Source Mac : c471544e425c | Destination Mac : ffffffffffff | Total Length : 72 | Source Address : 192.168.1.1 | Destination Address : 192.168.1.255 |       rejected
19 Protocol : 2048 | Source Mac : c471544e425c | Destination Mac : 08d40c7959c3 | Total Length : 236 | Source Address : 192.168.1.1 | Destination Address : 192.168.1.114 |     rejected
20 Protocol : 2048 | Source Mac : c471544e425c | Destination Mac : 08d40c7959c3 | Total Length : 248 | Source Address : 192.168.1.1 | Destination Address : 192.168.1.114 |     rejected
21 Protocol : 2048 | Source Mac : c471544e425c | Destination Mac : 08d40c7959c3 | Total Length : 40 | Source Address : 13.107.6.171 | Destination Address : 192.168.1.114 |     accepted
22 Protocol : 2048 | Source Mac : c471544e425c | Destination Mac : 08d40c7959c3 | Total Length : 40 | Source Address : 13.107.6.171 | Destination Address : 192.168.1.114 |     accepted
23 Protocol : 2048 | Source Mac : c471544e425c | Destination Mac : 08d40c7959c3 | Total Length : 40 | Source Address : 13.107.6.171 | Destination Address : 192.168.1.114 |     accepted
24 Protocol : 2048 | Source Mac : c471544e425c | Destination Mac : 08d40c7959c3 | Total Length : 333 | Source Address : 13.107.6.171 | Destination Address : 192.168.1.114 |    accepted
25 Protocol : 2048 | Source Mac : c471544e425c | Destination Mac : 08d40c7959c3 | Total Length : 78 | Source Address : 13.107.6.171 | Destination Address : 192.168.1.114 |     accepted
26 Protocol : 2048 | Source Mac : c471544e425c | Destination Mac : 08d40c7959c3 | Total Length : 52 | Source Address : 52.114.32.8 | Destination Address : 192.168.1.114 |      accepted
27 Protocol : 2048 | Source Mac : c471544e425c | Destination Mac : 08d40c7959c3 | Total Length : 52 | Source Address : 157.240.195.17 | Destination Address : 192.168.1.114 |   rejected
28 Protocol : 2048 | Source Mac : c471544e425c | Destination Mac : 08d40c7959c3 | Total Length : 87 | Source Address : 157.240.195.17 | Destination Address : 192.168.1.114 |   rejected
29 Protocol : 2048 | Source Mac : c471544e425c | Destination Mac : 08d40c7959c3 | Total Length : 229 | Source Address : 192.168.1.1 | Destination Address : 192.168.1.114 |     rejected
30 Protocol : 2048 | Source Mac : c471544e425c | Destination Mac : 08d40c7959c3 | Total Length : 276 | Source Address : 192.168.1.1 | Destination Address : 192.168.1.114 |     rejected
31 Protocol : 2048 | Source Mac : c471544e425c | Destination Mac : 08d40c7959c3 | Total Length : 52 | Source Address : 52.114.32.8 | Destination Address : 192.168.1.114 |      accepted
32 Protocol : 2048 | Source Mac : c471544e425c | Destination Mac : 08d40c7959c3 | Total Length : 478 | Source Address : 52.114.32.8 | Destination Address : 192.168.1.114 |     accepted
```

As a solution to do a real filtering

Use below python script to block IP address automatically in iptables.
we can use below script as block.py and place it in crontab



```
import pyshark
import subprocess
capture = pyshark.LiveCapture(interface='eth0')
capture.sniff(timeout=50)
    #print len(capture)
lst1=[]
    #IP address exclude, so below ip will not be blocked
lst2=['192.168.4.138']
def blockip(ip):
    cmd="/sbin/iptables -A INPUT -s "+ip+" -j DROP"
    print cmd
    subprocess.call(cmd,shell=True)

def blockIP1(ip):
    cmd="/sbin/iptables -A INPUT -s "+ip+" -j DROP"
    subprocess.call(cmd, shell=True)
    subprocess.call("kill -9 $(/usr/bin/pgrep dumpcap)", shell=True)
    subprocess.call("/usr/bin/kill -9 $(/usr/bin/pgrep tshark)", shell=True)
    for i in range(len(capture)):
        pack=capture[i]
        print pack
        lst1.append(pack['ip'].src)
        #print lst1
        ulst1=set(lst1)
        #print ulst1
        ulst1=list(ulst1)
    for i in range(len(ulst1)):
        ip=ulst1[i]
        if ip not in lst2:
            blockIP1(ip)
subprocess.call("kill -9 $(/usr/bin/pgrep dumpcap)", shell=True)
subprocess.call("/usr/bin/kill -9 $(/usr/bin/pgrep tshark)", shell=True)
```

CONCLUSION

For a better ip packet filtering, multiple firewall should be installed on different levels in some fashion. In matter of fact,Some packet-filtering firewalls will only be able to filter IP addresses and not the source TCP/UDP port, but having TCP or UDP filtering as a feature can provide much greater maneuverability, since traffic can be restricted for all incoming connections except those targeted by the firewall access rules.

Python libraries provides access to the flow of information. they can analyze captures and real time packets, extract information and return decisions.
However, making concrete decision regarding the firewall behavior is difficult since Python is a high level language. Thus, we ought to try using forms of parcers or commanding installed firewalls.