

1. Screenshots of R2 and KALI showing the netcat TCP chat

```
rtt min/avg/max/mdev = 0.356/0.471/0.784/0.146 ms
student@kali:~$ arp -a
? (10.10.10.2) at 00:00:00:00:00:03 [ether] on eth0
student@kali:~$ nc 10.10.10.2 5000
Hi
Hello
This is a part 1 message

yes it is
█
```

```
root@CN-R2: /home/student
File Edit Tabs Help
student@CN-R2:~$ sudo su
[sudo] password for student:
root@CN-R2:/home/student# nc -l 5000
Hi
Hello
This is a part 1 message

yes it is
█
```

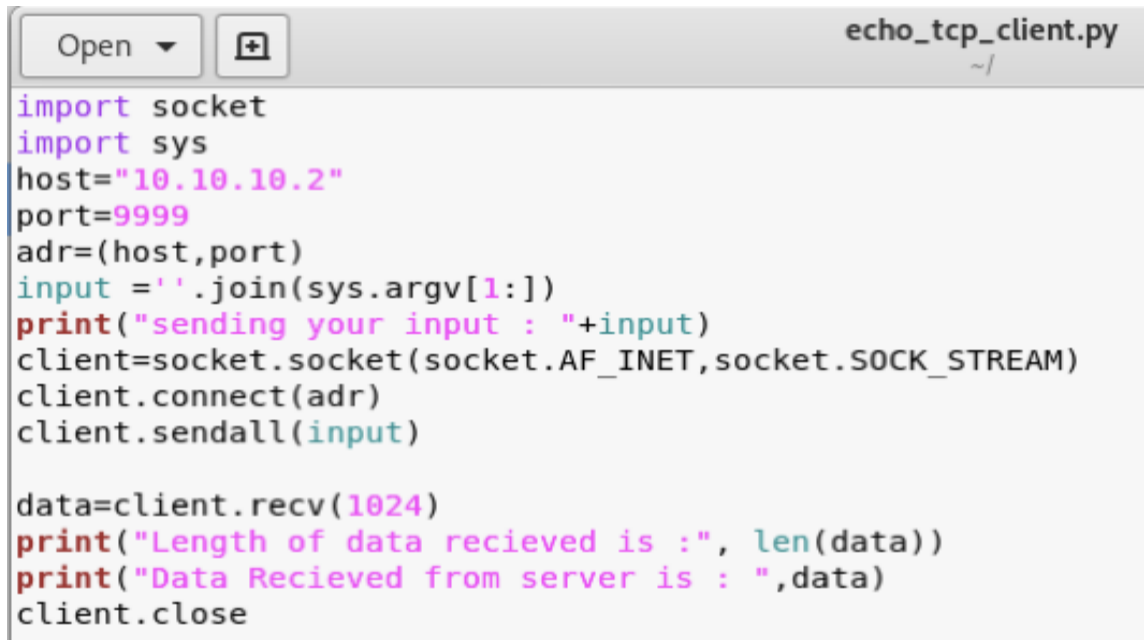
```
student@kali:~$ arp -a
? (10.10.10.2) at 00:00:00:00:00:03 [ether] on eth0
student@kali:~$ nc 10.10.10.2 5000
Hi
Hello
This is a part 1 message

yes it is
^C
student@kali:~$ mawk -W interactive '$0="KALI: "$0' | nc 10.10.10.2 5000
Hi
R2: Hello
This is Part 1 Message
R2: Yes It is
```

```
root@CN-R2: /home/student
File Edit Tabs Help
student@CN-R2:~$ sudo su
[sudo] password for student:
root@CN-R2:/home/student# nc -l 5000
Hi
Hello
This is a part 1 message

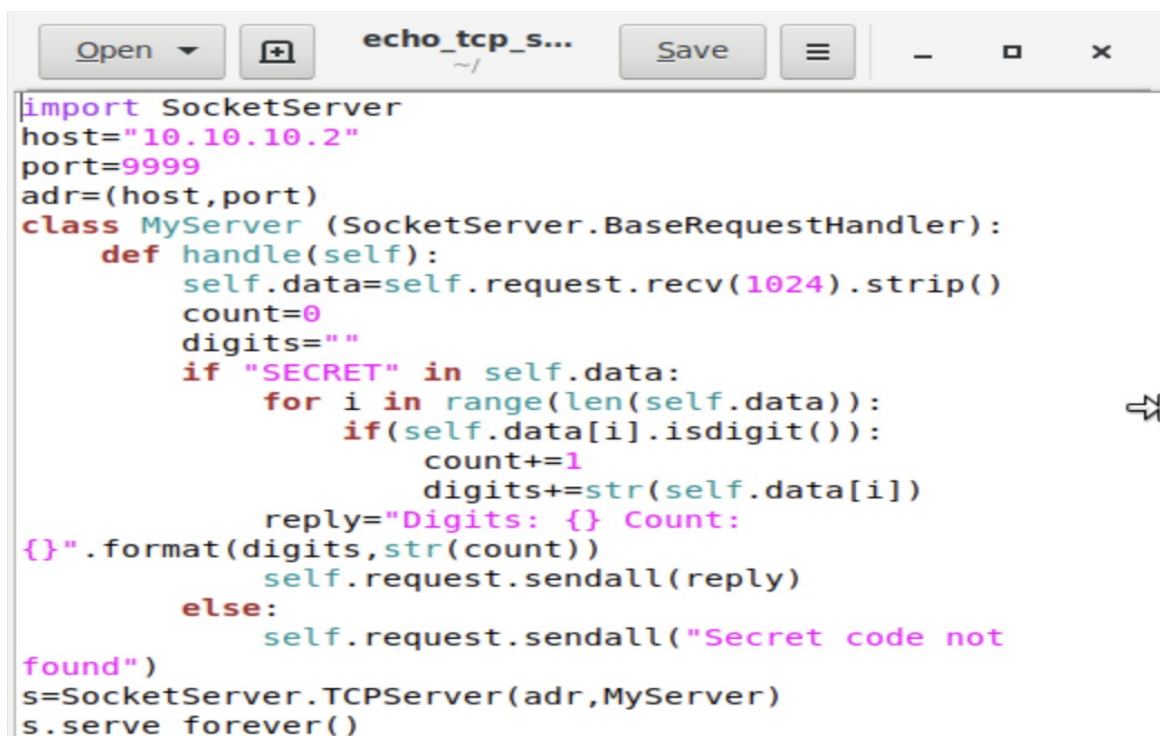
yes it is
root@CN-R2:/home/student# mawk -W interactive '$0="R2: "$0' | nc -l -p <port_number>
bash: syntax error near unexpected token `newline'
root@CN-R2:/home/student# mawk -W interactive '$0="R2: "$0' | nc -l -p <port_number>
bash: syntax error near unexpected token `newline'
root@CN-R2:/home/student# mawk -W interactive '$0="R2: "$0' | nc -l -p 5000
KALI: Hi
Hello
KALI: This is Part 1 Message
Yes It is
```

## 2. Screenshots of echo\_tcp\_server.py and echo\_tcp\_client.py



```
import socket
import sys
host="10.10.10.2"
port=9999
adr=(host,port)
input = ''.join(sys.argv[1:])
print("sending your input : "+input)
client=socket.socket(socket.AF_INET,socket.SOCK_STREAM)
client.connect(adr)
client.sendall(input)

data=client.recv(1024)
print("Length of data recieved is :", len(data))
print("Data Recieved from server is :",data)
client.close
```



```
import SocketServer
host="10.10.10.2"
port=9999
adr=(host,port)
class MyServer (SocketServer.BaseRequestHandler):
    def handle(self):
        self.data=self.request.recv(1024).strip()
        count=0
        digits=""
        if "SECRET" in self.data:
            for i in range(len(self.data)):
                if(self.data[i].isdigit()):
                    count+=1
                    digits+=str(self.data[i])
            reply="Digits: {} Count:
            {}".format(digits,str(count))
            self.request.sendall(reply)
        else:
            self.request.sendall("Secret code not
            found")
s=SocketServer.TCPServer(adr,MyServer)
s.serve_forever()
```

3. Screenshots of tcp\_file\_transfer\_server.py and tcp\_file\_transfer\_client.py

```
tcp_file_transfer_client.py
~/
Open ▾ [icon]

import socket
import os
port = 5000
host = "10.10.10.2"
addr=(host,port)
f=open("Kalifile.txt","w")
print("enter number of lines of data")
n= int(input())
print("enter data")
for i in range(n):
    f.write(raw_input()+"\n")
f.close()
f=open("kalifile.txt","rb")
l=f.read()
f.close()
client=socket.socket(socket.AF_INET, socket.SOCK_STREAM)
client.connect(addr)
client.send(l)
client.shutdown(socket.SHUT_WR)
client.close()
```

```
tcp_file_transf...
~/
Open ▾ [icon] Save [icon] [icon] [icon] [icon]

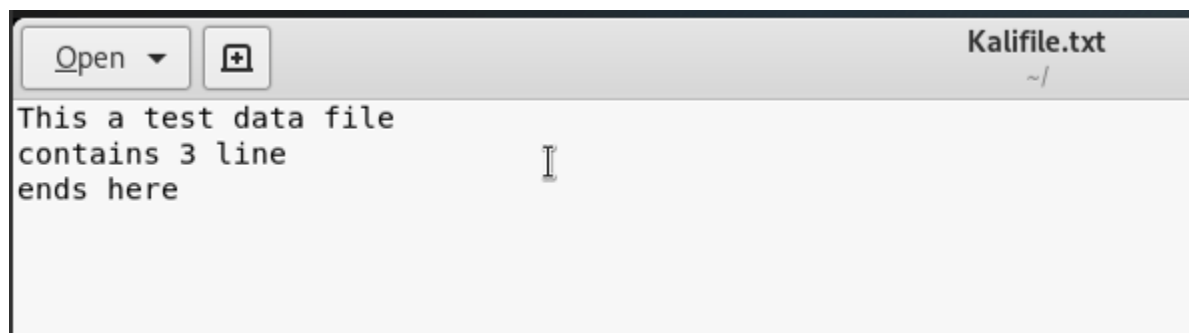
import SocketServer
import os

host="10.10.10.2"
port=5000
addr=(host,port)
class MyServer(SocketServer.BaseRequestHandler):
    def handle(self):
        while True:
            message=self.request.recv(1024).strip()
            print("The input from {}:{}".format(self.client_address[0],self.client_address[1]))
            if not message:
                break
            f=open("r2data.txt", "wb")
            f.write(message)
            f.close()
            f=open("r2data.txt","r")
            for l in f:
                print(l)
s=SocketServer.TCPServer(addr,MyServer)
s.serve_forever()
```

4. Screenshots showing the behavior of Part 2. Make sure to include cases with and without the secret code.

```
root@kali:/home/student# python echo_tcp_client.py This is a test
sending your input : Thisisatest
('Length of data recieved is : ', 21)
('Data Recieved from server is : ', 'Secret code not found')
root@kali:/home/student# python echo_tcp_client.py This is a test 234SECRET
with secret me55age
sending your input : Thisisatest234SECRETtestwithsecretme55age
('Length of data recieved is : ', 22)
('Data Recieved from server is : ', 'Digits: 23455 Count: 5')
```

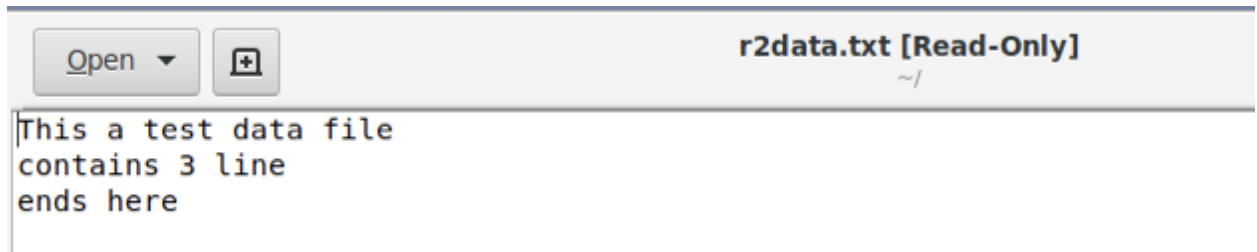
5. Screenshots showing the file transfer in Part 3: show the original file on KALI, the KALI file before transferring



The screenshot shows a text editor window with a title bar that includes an 'Open' button and a file icon. The file name 'Kalifile.txt' is displayed in the top right corner. The text area contains the following content:

```
This a test data file
contains 3 line
ends here
```

File after transferring



6. A. In netcat, you specified the port on which the server should listen but did not specify the port the server should use to send a message to the client. Which client port does your netcat server send to? Use Wireshark to answer the question and include a screenshot.

Here we can observe the message exchange from 38536 -> 4000 and 4000 - >38536. The netcat server is sending a message to the client using the port from where it got the message.

```
root@kali:/home/student# nc 10.10.10.2 4000
Hello
Hi
Test
test ok
```

```
student@CN-R2:~$ nc -l 4000
Hello
Hi
Test
test ok
```

No.	Time	Source	Destination	Protocol	Length	Info
26	4.854287750	10.10.10.3	10.10.10.2	TCP	71	38536 → 4000 [PSH, ACK] Seq=1 Ack=1 Win=227
27	4.854369243	10.10.10.2	10.10.10.3	TCP	66	4000 → 38536 [ACK] Seq=1 Ack=6 Win=227
58	14.119056176	10.10.10.2	10.10.10.3	TCP	74	4000 → 38536 [PSH, ACK] Seq=1 Ack=6 Win=227
59	14.119589488	10.10.10.3	10.10.10.2	TCP	66	38536 → 4000 [ACK] Seq=6 Ack=9 Win=229

**B. Briefly explain your code from Part 2 and Part 3. In your explanation, focus not on the syntax but on the TCP communication establishment and flow.**

Server:-

Initially, a TCP connection is opened by a server and it listens via the mentioned port. Using the method `socketserver.TCPServer(("10.10.10.2",5000), FileHandle)` that accepts host address, port, and Handler class to handle the received data.

Client:

The client connects to the given host address and port by following the AF\_INET: address format is host and port number. Once the connection is established it will send messages through that connection using `send()` method.

**C. What does the socket system call return?**

A socket is created using the socket system call, which also yields a socket descriptor, which is a small, non-negative integer that serves as the socket's file descriptor. The file descriptor is used by subsequent system functions associated with sockets to identify this particular socket. The system function `socket()` yields a file descriptor. Because data may be read and written on a socket using the standard `read()` and `write()` system methods, sockets are comparable to opened files. Like a file descriptor, the function returns a non-negative integer number that we designate as the socket descriptor, or -1 in the event of an error.

**D. What does the bind system call return? Who calls bind (client/server)?**

In network programming, the bind system call is used to link a socket to a certain local address and port number. Usually, a server application calls the bind function while using the Berkeley sockets API, which is popular for network programming in Unix-like operating systems. A socket is given a local protocol address via the `bind()` function. With the Internet protocols, the address is the combination of an IPv4

or IPv6 address (32-bit or 128-bit) address along with a 16-bit TCP port number. `bind()` returns 0 if it succeeds, -1 on error. Server calls `bind()` to create a socket and assign an address to it. In summary, the server often uses the `bind` system function to define the port and local address on which it will wait for incoming connections. When establishing a connection, however, the client frequently depends on the operating system to manage the allocation of a local port.

**E. Suppose you wanted to send an urgent message from a remote client to a server as fast as possible. Would you use UDP or TCP? Why?**

UDP (User Datagram Protocol) may be a better option than TCP (Transmission Control Protocol) if you need to convey a critical message from a remote client to a server as soon as feasible. The following are the causes:

**Low Overhead:** Compared to TCP, UDP has a lower overhead. Additional delay may be introduced by TCP features such as data sequencing, retransmission of dropped packets, and error checking. In comparison, UDP transmits short, time-sensitive communications more quickly due to its low overhead.

**No Connection Establishment:** Before data is sent over TCP, a connection establishment (three-way handshake) is necessary. There is a slight delay due to this initial setup. Because UDP is connectionless, it is free of this overhead. UDP is a better option if you need to deliver the message fast without waiting for setup, especially if it is urgent.

**Lack of Flow Control:** TCP has flow control features to make sure the sender doesn't send too much data at once to the recipient. Although this adds more delay, dependability benefits from it. Since UDP lacks flow control, data can be sent over the network as fast as it can.

It's crucial to remember that UDP does not offer the same degree of dependability as TCP. Delivery is not guaranteed, and there is no receipt acknowledgment. TCP may still be a superior option if your application requires assured delivery and dependability, even at the expense of some delay. The decision between UDP and TCP ultimately comes down to the particular needs of your application and how much importance you place on things like latency and dependability.

**F. What is Nagle's algorithm? What problem does it aim to solve and how?**

The goal of Nagle's method is to increase the effectiveness of tiny packet communication in computer networks using optimization based on assumptions. It attempts to solve the issue that, especially in interactive applications, short, frequent messages lead to needless overhead and decreased network efficiency. Sending brief messages one at a time might be inefficient in various network communication



circumstances, especially interactive applications. The overhead of a new packet header for every single interaction results in higher network use and perhaps higher delay.

**This is the operation of Nagle's algorithm:**

**Buffer small packets:** Small packets should be buffered using Nagle's technique, which prevents data from being sent instantly when a tiny quantity is sent by an application.

**Wait for ACK or fill the buffer:** Next, the algorithm waits for the recipient to acknowledge receipt of the previously delivered data (ACK) or until the buffer is filled with enough data.

**Send bigger packet:** Nagle's technique sends a larger packet that combines the buffered data once either the ACK is received or a certain amount of data is gathered.

Nagle's technique helps to decrease the overhead associated with the transmission of tiny messages by sending bigger packets when possible and delaying the delivery of smaller packets. This is particularly advantageous for interactive applications, where it is essential to minimize latency. Although Nagle's algorithm works well most of the time, there are several situations in which turning it off might be advantageous. For example, turning off Nagle's technique could be useful for applications that need low-latency communication and can handle a slight increase in bandwidth utilization.

#### **G. Explain one potential scenario in which delayed ACK could be problematic.**

A deliberate delay in transmitting acknowledgment packets for received data is known as a delayed ACK (acknowledgment) mechanism in networking. By sending fewer ACK packets, the goal is to improve network efficiency and increase throughput. On the other hand, delayed ACK may cause problems with performance in some situations. When only an insignificant amount of data needs to be exchanged, a client and server connection might provide challenges. Let's look at an example where a client sends a server a small amount of data and both entities use delayed ACK:

**Client transmits small data:** The client transmits, maybe less than the Maximum Segment Size (MSS), a little bit of data to the server.

**Server-side delayed acknowledgment (ACK):** When the server uses delayed ACK, it doesn't instantly reply with an acknowledgment to the client. It bides its time in the hopes of grouping several acknowledgments together.

**No more data from the client:** The server's acknowledgment is needlessly delayed if the client doesn't submit any more data right away.

**Ineffective communication:** In this instance, the client may send needless retransmissions as a result of the server's tardiness in providing the ACK. When low-latency communication is essential, like in interactive applications or real-time communication, this inefficiency might be very apparent.

Certain systems dynamically modify the usage of delayed acknowledgment depending on observed communication patterns in order to reduce this problem. For instance, the ACK may be sent right away to save needless retransmissions if only a small amount of data is provided and no rapid follow-up data is noticed.

