COMP 8505

Covert Communication Application

Final Project

Elton Sia (A00800541) & Thilina Ratnayake (A00802338)
11/28/2015

Table of Contents

Summary

The work contained in this COMP 8505 Final Project is a covert backdoor system which allows an attacker to send and receive messages using covert channels over TCP and UDP mediums, monitor a directory for file changes and covertly exfiltrate those files upon a change. The backdoor component utilizes process masking & raw sockets to minimize it's footprint and reduce the chance of detection.

Objective

- To bring together several stealth software and backdoor concepts covered in class into a single covert communication application.
- To learn how to use such an application in allowing stealthy access to a network or to exfiltrate data from systems within a network
- To design and implement a complete covert application that will allow a user(attacker) to communicate with a server(backdoor) installed on the victim machine inside a network.

Assumptions

- Firewall rules drops all packets
- The backdoor is already installed on the victim's machine as well as the libraries required to run the server

Features

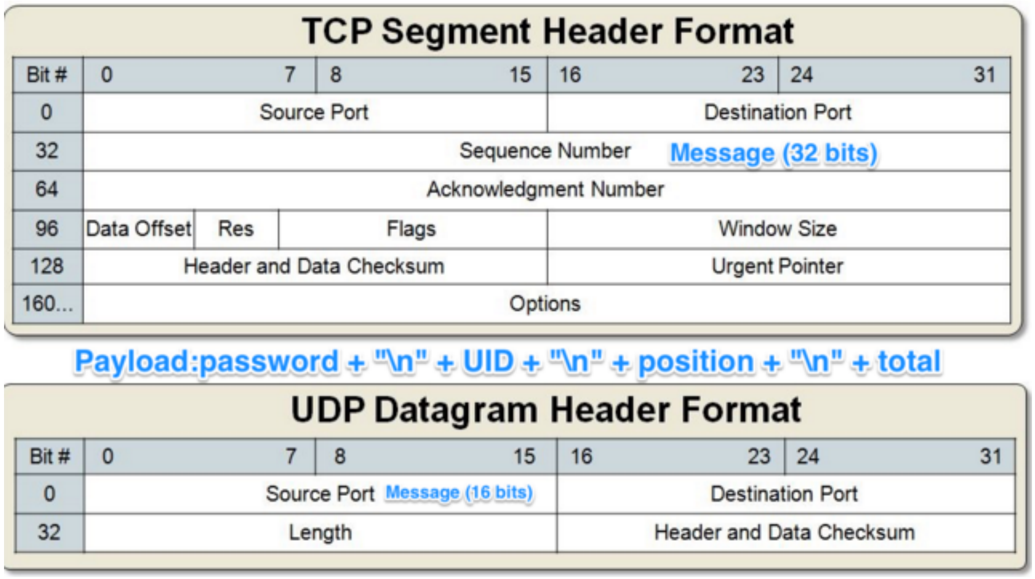
1. **Covert Channel (TCP & UDP)**
    a. The attacker will have the option of using either TCP or UDP for sending or receiving dat.
        i. When using TCP, the command will be hidden into the sequence number field of the packet.
        ii. When using UDP, the command will be hidden into the source port field of the packet.
    b. **Authentication**: To authenticate packets from the attacker, packets will have the following features:
        i. The **TTL key** of the packets received will have the value of 164. This will be the value used for filtering whether it is the attackers IP or not.

ii. Using a password to authenticate packets. An encrypted password will be placed inside the payload of every packet sent. Once a packet passes the TTL filter, it will begin decrypting the payload and check it against the password. If and only if the password matches, then will the command be deciphered.

c. **Sequencing:**

i. The payload field will contain the following information delimited by newline characters:
Password + "\n" + UID + "\n" + position + "\n" + total

1. The UID will allow the program to store commands split into multiple commands under a single **Unique Identifier**(UID)
2. The position and total indicate which message out of a sequence has been received (eg. Message 1 out of 3)



TCP Segment Header Format

Bit #	0 7	8 15	16 23	24 31
0	Source Port		Destination Port	
32	Sequence Number Message (32 bits)			
64	Acknowledgment Number			
96	Data Offset / Res	Flags	Window Size	
128	Header and Data Checksum		Urgent Pointer	
160...	Options			

Payload:password + "\n" + UID + "\n" + position + "\n" + total

UDP Datagram Header Format

Bit #	0 7	8 15	16 23	24 31
0	Source Port Message (16 bits)		Destination Port	
32	Length		Header and Data Checksum	

2. **Backdoor**

a. Process Masking: The backdoor program will have a different process name so as to mask it from users who may check processes that are running in the system.

b. Covert Channels: It can use both TCP and UDP connections to receive and SEND packets back to the attacker. When sending the output back to the attacker, the hidden data will be hidden inside the sequence number when using TCP and into the source port when using UDP.

3. **File Monitoring**

a. A separate process running on the backdoor that will monitor any file creation and modification on a directory.

b. When any changes occur on the directory(file creation and modification), it will send that file via the covert channels to the client(attacker).

Implementation

There are two components in implementing the covert communication application. One is the client (attacker) and the other is the server (backdoor).

Client

- External libraries required to be installed before running **client.py**
    - Scapy
    - Pycrypto
    - parseConfig (a configuration file parser that we built)
- To run the **client.py**.
    - Before running the program, make sure that the necessary changes in the configuration file has been made. (Local IP and ServerIP)
    - To run the program:
        - Python client.py

The client.py has two main components when it runs. The first component is an interactive command line where it allows the user (attacker) to send commands to the server (backdoor) and the server sending back an output. In this component, the user starts off with a command line interface asking it to input a command, once the user has chosen what command they want to send, we encrypt that command and then changing the encrypted command into bits. We then send it to our chunker method where we chunk the bits into the correct amount depending on the user input if they wanted to use **TCP** or **UDP**. If they had chosen TCP, we will be chunking the bits into 32 bits because we will hide the data into the sequence number field whereas if the user had chosen UDP, then we will be chunking the bits into 16 bits because we will hide the data into the source port field. Once we have the correct amount of bits, we then send it to our packet crafting method where we first check how many packets we will need to create and send and put a unique identifier on that set of packets. Once we have done that, we then create all the packets and inside the payload of the packet, we add in an encrypted authentication password to make sure that the server is indeed decrypting the correct packet. On top of that, we set the **TTL** of the packet to 164 before sending it over to the server. Once the packet is sent, we then listen for a response from the server in which we have set a timeout of 10 seconds in case that the output of the command requires many packets. Once we have received a packet, we check if the TTL matches ours and then we decrypt the payload and also check to see if the password matches with ours. Once it matches, we store the sequence number into an array, if it's TCP, and/or source port into an array, if it's UDP, and then change it into bits. Once we have the bits of all the sequence number/source port, we then decrypt them and combine them all together and print the output in the terminal.

The second component is running on a second process that is created at the start of the program. All the process does is listen for any packets on port 80 with the correct TTL and the correct authentication password on the payload. Once we receive, the first packet, we will be able to find out how many packets we are expecting from the payload. We then wait for all the packets we need and then sending it to the writeFile method where we re-create the file. We first change everything into characters and decrypt the whole thing. We then look for the NULL byte which will tell us where the file name is stored and where the data is stored. Once we have the file name and the data, we then re-create the file.

Server

- External Libraries required to be installed before running **server.py.**
  - Scapy
  - Pycrypto
  - Setproctitle
- To run the **server.py**
  - Before running the server.py, make sure to that the necessary changes in the configuration file has been made. (LocalIP, ClientIP)
  - To run the program:
    - Python server.py

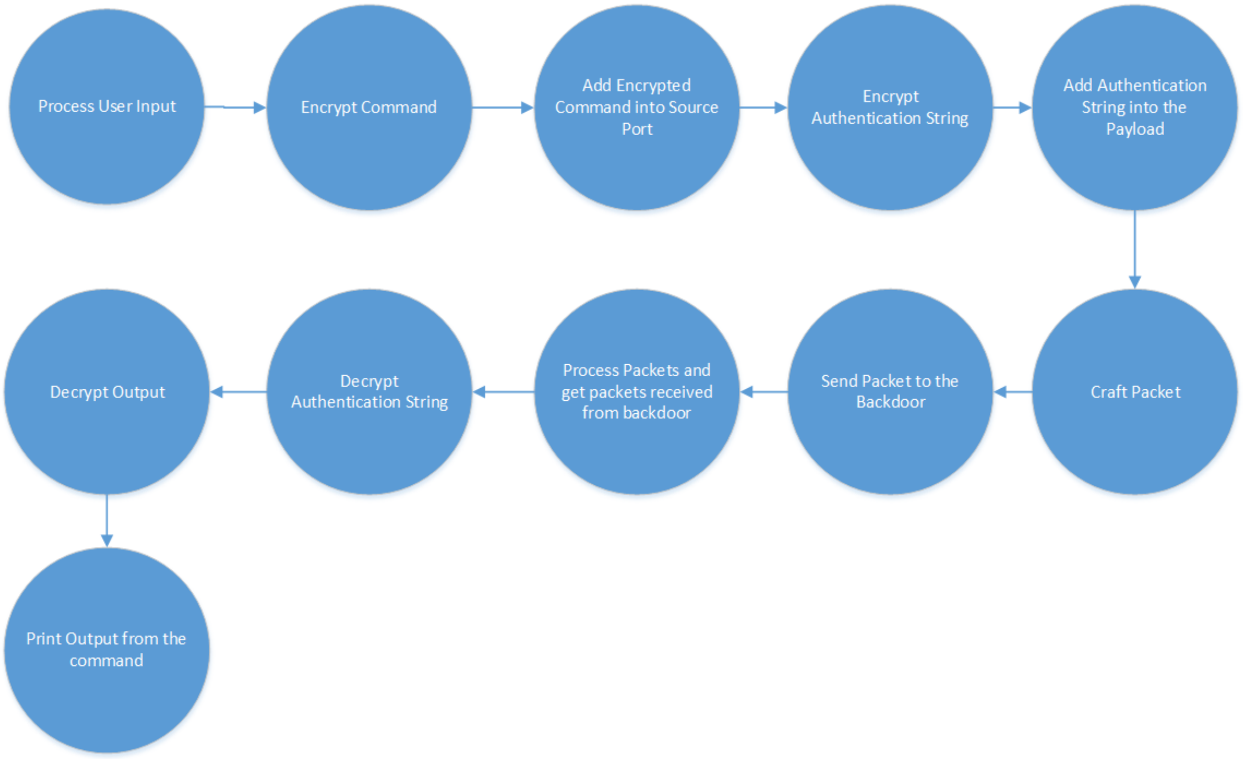
The server.py also has two main components when it runs. The first component of the server listens for all the packets. When it sniffs for packets, we check whether it matches the TTL key, we then decrypt the payload of that packet and check whether it matches the authentication password. If it matches, we can also get the total amount of packet we are expected to get because it is also in the payload. We then check whether the packet is a TCP packet or a UDP packet. If it's a TCP packet, we send it to our lengthChecker method where we check for the correct sequence number. In the sequence number, we are expecting to have 32 bits, once we have changed the sequence numbers into binary, we make sure that the binary representation is divisible to the nearest 8 division. For example, if the last packet contains only 14 bits, we append to the start two zeroes to make it a total of 16 bits. Once we have set the correct bits, we then chunk it all into 8 bits in order to change it to characters. We then decrypt and we get the command. We then run it in a sub-process and encrypt the output and send it back to the client.

The second component of the server is the file monitoring component. The server monitors a specific folder specified by the user and any file creation or modification inside that folder; we send a copy of that file to our client via the cover channels. Note that we use only UDP for sending the files because having a TCP connection will increase the chances of detecting our backdoor.

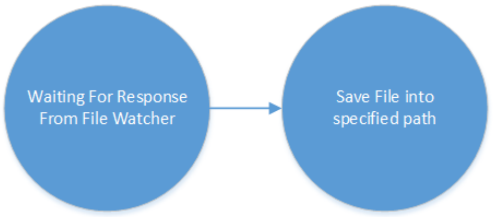
Design Work

Client – Finite State Machine

Thread 1

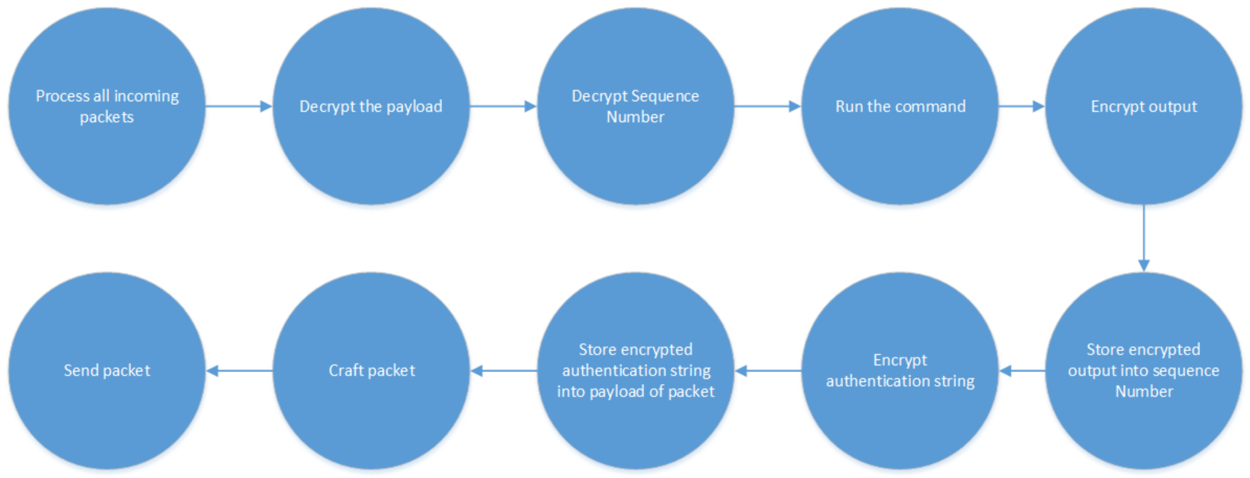

Process User Input
Encrypt Command
Add Encrypted Command into Source Port
Encrypt Authentication String
Add Authentication String into the Payload
Craft Packet
Send Packet to the Backdoor
Process Packets and get packets received from backdoor
Decrypt Authentication String
Decrypt Output
Print Output from the command

Thread 2

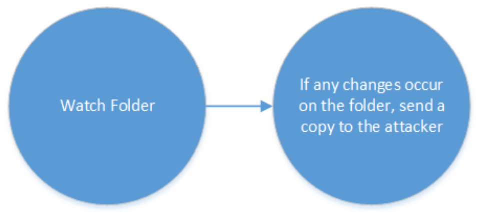

Waiting For Response From File Watcher
Save File into specified path

Server – Finite State Machine

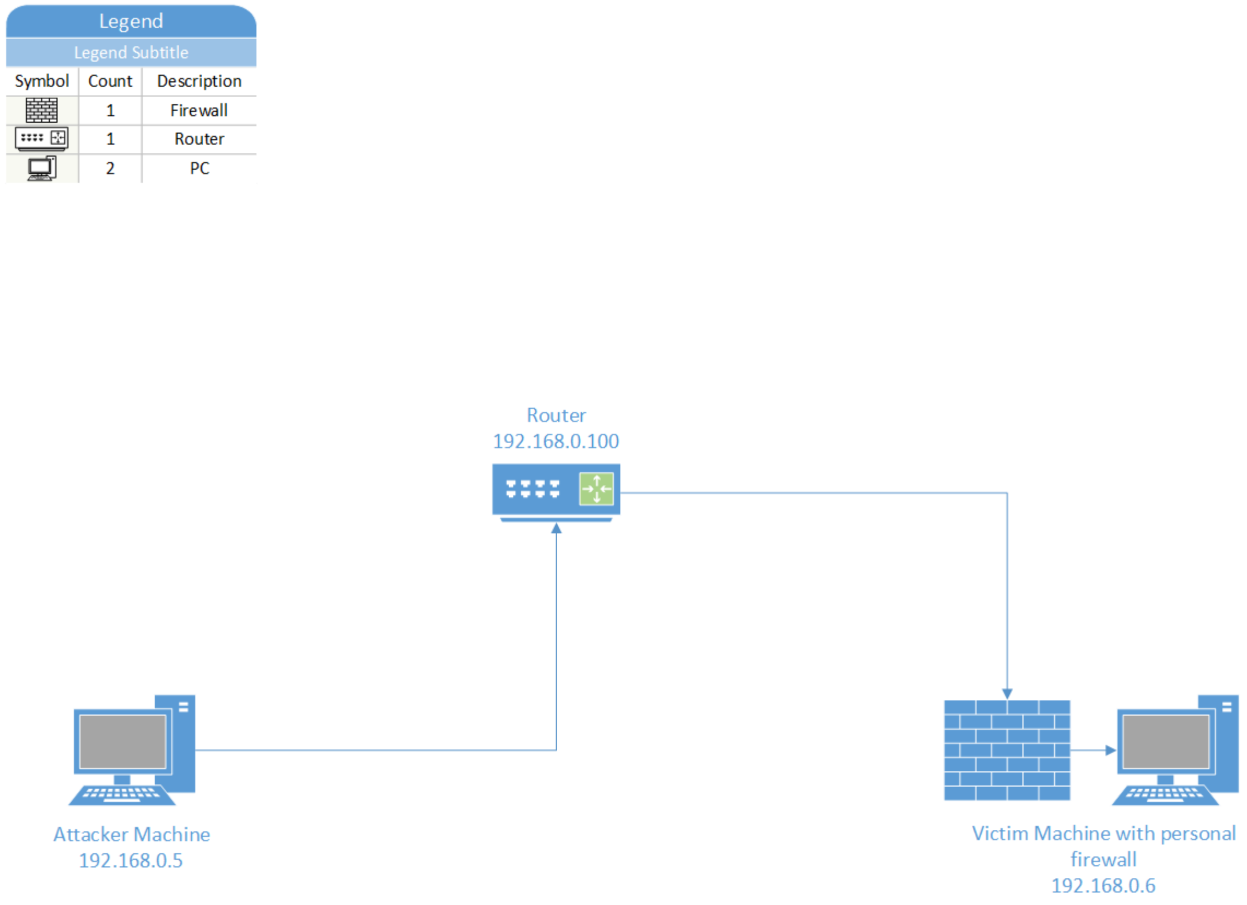
Thread 1


Process all incoming packets
Decrypt the payload
Decrypt Sequence Number
Run the command
Encrypt output
Store encrypted output into sequence Number
Encrypt authentication string
Store encrypted authentication string into payload of packet
Craft packet
Send packet

Thread 2


Watch Folder
If any changes occur on the folder, send a copy to the attacker

Network Diagram



Legend		
Legend Subtitle		
Symbol	Count	Description
▦	1	Firewall
▭	1	Router
🖥	2	PC

Detection

One way to detect our program is if the system administrator checks the sequence number (when we use TCP) on Wireshark because they will not be in sequence since we hide our data inside the sequence number when using TCP thus resulting in different sequence number. This will alert any system administrator if they see this and will get suspicious right away. Another way to detect our program is if the system administrator checks the source ports (when we use UDP) on Wireshark because they will be very different from one packet to the next. Note that we are sending our packets as fast as we can since this is just proof of concept. One way to mitigate that is to send our packets a lot slower.

Another way for a system administrator to detect the covert channels is if they had already saved a whitelist for the processes running on their machine. This way, they can simply just compare the processes running on the machine with the whitelist and see which ones do not match the list. They can also automate this and have a script running where it always checks for new processes being created that's not in the whitelist.

Prevention

As a network administrator or systems analyst, there are a number of ways in which to detect and prevent a backdoor like the one detailed in this document from succeeding.

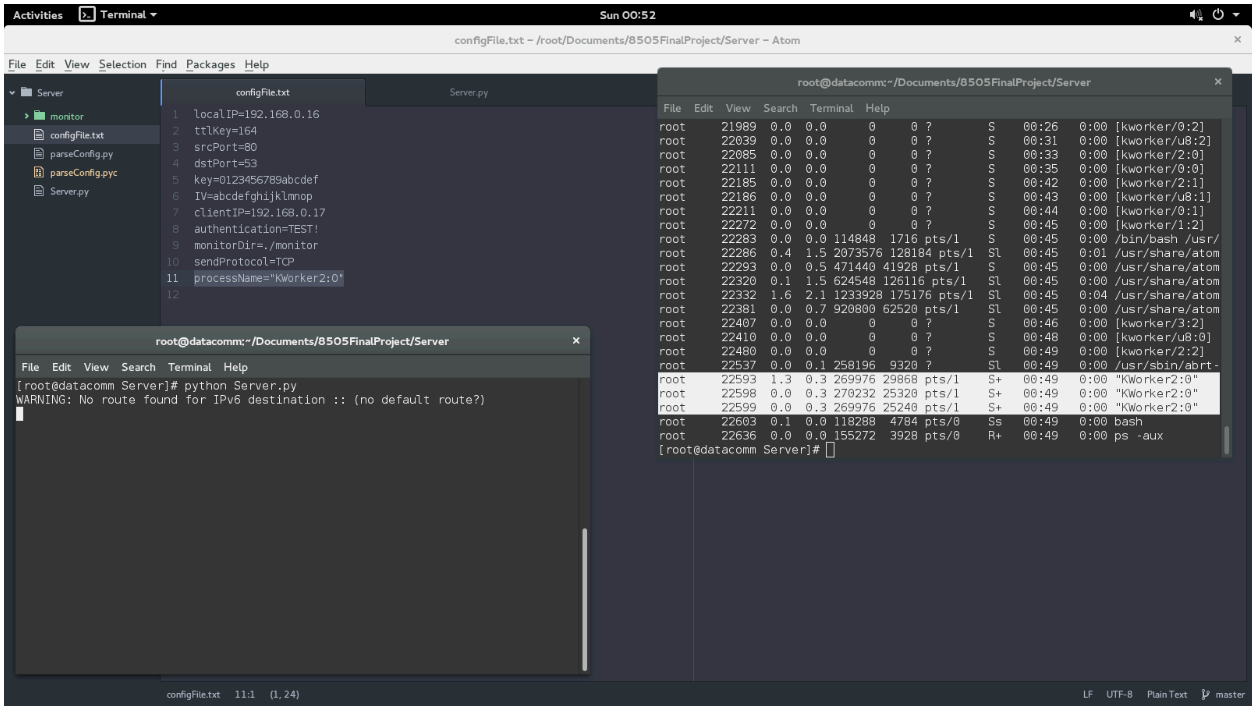
1. **Process Monitoring & Whitelisting.** An analyst can ensure that no fake processes spin up by taking a snapshot of the system on first-run and then continuously maintaining a whitelist as time goes on. If a process starts up with a similar name when it shouldn't, the administrator can choose to be notified and then act as necessary.
2. **External Firewall:** If a target was separated from the internet via firewall or edge router, this backdoor would be useless because there would be no way for the firewall to forward commands to the victim (especially because it's using raw sockets).
3. **Deep Packet Inspection:** If the firewall were checking the packet contents, it could look for anomalies / suspicious traits, for example:
   a. Looking for inconsistent sequence numbers in packets. (Packet one with sequence number 100000 and packet two with 102912041)
   b. Looking at messages arriving from one host via UDP with widely different ports.

Tests and Results

#	Name	Resource	Expected	Actual	Result	Figure
1	Server process title is masked when it is running	Server.py	Process name is masked	As expected	Pass	1
2	Firewall rule DROPS every packet	Iptables	Firewall rules DROPS everything	As expected	Pass	2
3	Client sends encrypted commands to the server (TCP)	Client.py	Client sends an encrypted command to the server	As expected	Pass	3
4	Server decrypts the encrypted command and runs the command and sends encrypted output to the client (TCP)	Server.py	Server decrypts command and runs it then send an encrypted output back to the client	As expected	Pass	4.1, 4.2
5	Client receives the encrypted output and decrypts it and prints it (TCP)	Client.py	Client receives the output and decrypts it then prints it	As expected	Pass	5

6	Server monitors specific folder and on file creation sends it to the client (UDP)	Server.py	If any file is created on the folder, send it to the client	As expected	Pass	6.1, 6.2
7	Client receives the file from the server and re-creates it (Newly created file) (UDP)	Client.py	Re-create the file that was sent by the server	As expected	Pass	7.1, 7.2, 7.3
8	Server monitors specific folder and on file modification sends it to the client (UDP)	Server.py	If any file is modified on the folder, send it to the client	As expected	Pass	8
9	Client receives the file from the server and re-creates it (Modified) (UDP)	Client.py	Re-create the file that was sent by the server	As expected	Pass	9
10	Client sends encrypted commands to the server (UDP)	Client.py	Client sends an encrypted command to the server	As expected	Pass	10
11	Server decrypts the encrypted command and runs the command and sends encrypted output to the client (UDP)	Server.py	Server decrypts command and runs it then send an encrypted output back to the client	As expected	Pass	11.1, 11.2
12	Client receives the encrypted output and decrypts it and prints it (UDP)	Client.py	Client receives the output and decrypts it then prints it	As expected	Pass	12

Figure 1: Process title changed


Activities
Terminal
Sun 00:52
configFile.txt – /root/Documents/8505FinalProject/Server – Atom
File Edit View Selection Find Packages Help
Server
monitor
configFile.txt
parseConfig.py
parseConfig.pyc
Server.py
configFile.txt 11:1 (1, 24)
LF UTF-8 Plain Text master

```
localIP=192.168.0.16
ttlKey=164
srcPort=80
dstPort=53
key=0123456789abcdef
IV=abcdefghijklmnop
clientIP=192.168.0.17
authentication=TEST!
monitorDir=./monitor
sendProtocol=TCP
processName="KWorker2:0"
```

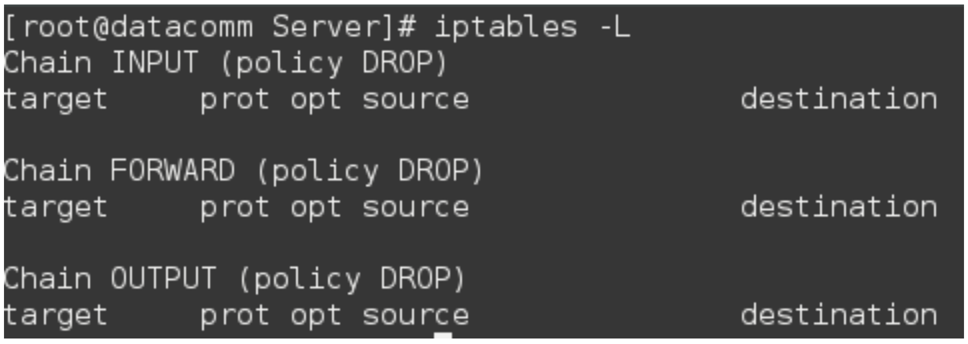
root@datacomm:~/Documents/8505FinalProject/Server

```
[root@datacomm Server]# python Server.py
WARNING: No route found for IPv6 destination :: (no default route?)
```

root@datacomm:~/Documents/8505FinalProject/Server

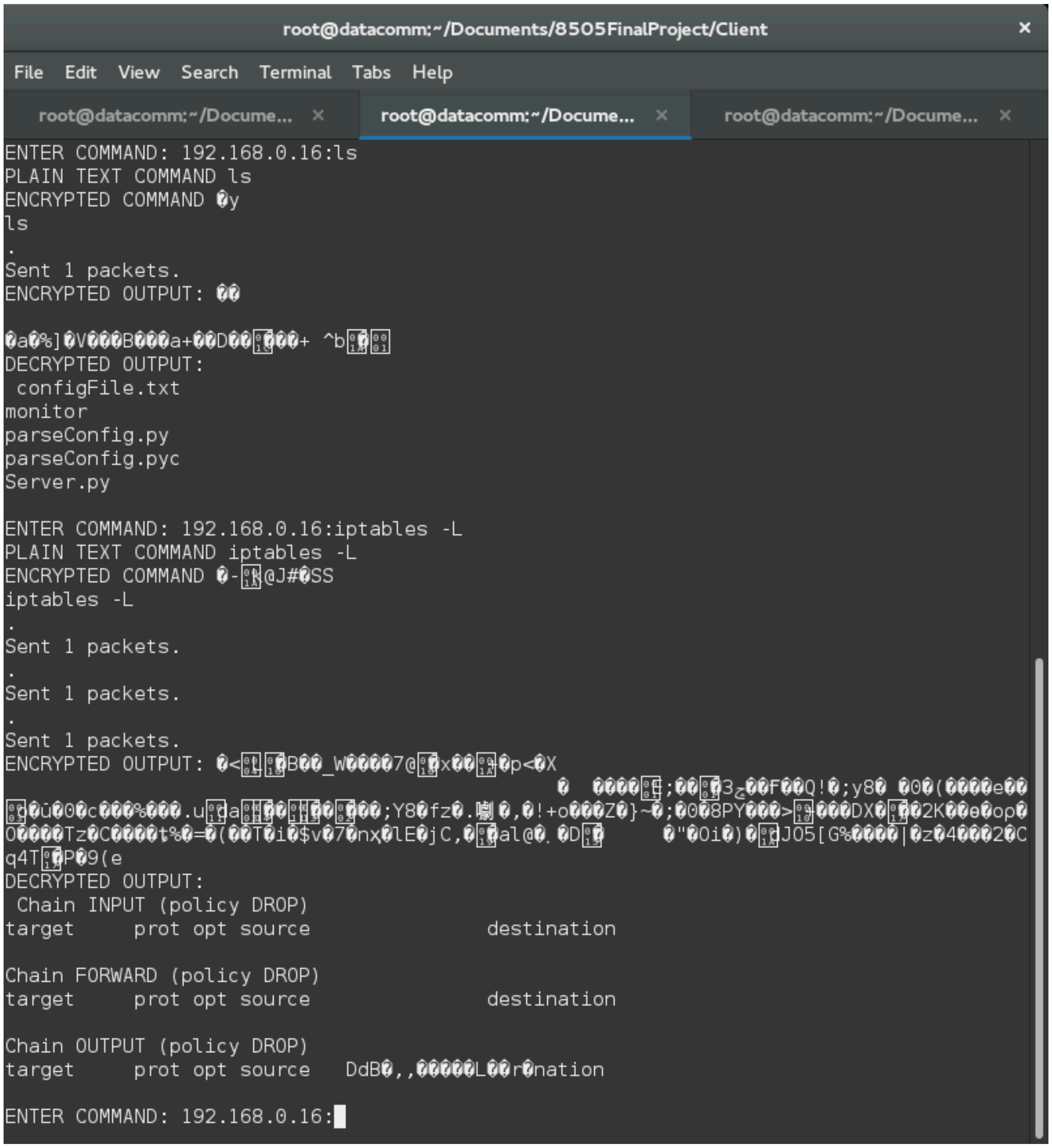
```
root      21989  0.0  0.0      0     0 ?        S    00:26   0:00 [kworker/0:2]
root      22039  0.0  0.0      0     0 ?        S    00:31   0:00 [kworker/u8:2]
root      22085  0.0  0.0      0     0 ?        S    00:33   0:00 [kworker/2:0]
root      22111  0.0  0.0      0     0 ?        S    00:35   0:00 [kworker/0:0]
root      22185  0.0  0.0      0     0 ?        S    00:42   0:00 [kworker/2:1]
root      22186  0.0  0.0      0     0 ?        S    00:43   0:00 [kworker/u8:1]
root      22211  0.0  0.0      0     0 ?        S    00:44   0:00 [kworker/0:1]
root      22272  0.0  0.0      0     0 ?        S    00:45   0:00 [kworker/1:2]
root      22283  0.0  0.0 114848  1716 pts/1    S    00:45   0:00 /bin/bash /usr/
root      22286  0.4  1.5 2073576 128184 pts/1  Sl   00:45   0:01 /usr/share/atom
root      22293  0.0  0.5 471440 41928 pts/1    S    00:45   0:00 /usr/share/atom
root      22320  0.1  1.5 624548 126116 pts/1   Sl   00:45   0:00 /usr/share/atom
root      22332  1.6  2.1 1233928 175176 pts/1  Sl   00:45   0:04 /usr/share/atom
root      22381  0.0  0.7 920800 62520 pts/1    Sl   00:45   0:00 /usr/share/atom
root      22407  0.0  0.0      0     0 ?        S    00:46   0:00 [kworker/3:2]
root      22410  0.0  0.0      0     0 ?        S    00:48   0:00 [kworker/u8:0]
root      22480  0.0  0.0      0     0 ?        S    00:49   0:00 [kworker/2:2]
root      22537  0.0  0.1 258196  9320 ?        Sl   00:49   0:00 /usr/sbin/abrt-
root      22593  1.3  0.3 269976 29868 pts/1    S+   00:49   0:00 "KWorker2:0"
root      22598  0.0  0.3 270232 25320 pts/1    S+   00:49   0:00 "KWorker2:0"
root      22599  0.0  0.3 269976 25240 pts/1    S+   00:49   0:00 "KWorker2:0"
root      22603  0.1  0.0 118288  4784 pts/0    Ss   00:49   0:00 bash
root      22636  0.0  0.0 155272  3928 pts/0    R+   00:49   0:00 ps -aux
[root@datacomm Server]#
```

Figure 2: Firewall rule drops all packets



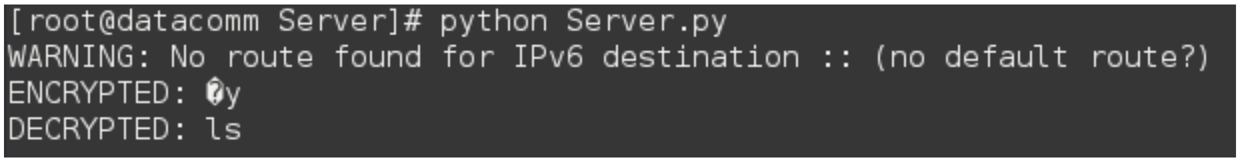
```
[root@datacomm Server]# iptables -L
Chain INPUT (policy DROP)
target     prot opt source               destination

Chain FORWARD (policy DROP)
target     prot opt source               destination

Chain OUTPUT (policy DROP)
target     prot opt source               destination
```

Figure 3:Client sending encrypted commands


root@datacomm:~/Documents/8505FinalProject/Client
File Edit View Search Terminal Tabs Help
root@datacomm:~/Docume...
root@datacomm:~/Docume...
root@datacomm:~/Docume...

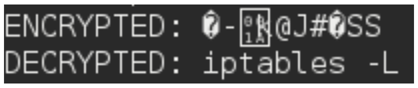
```
ENTER COMMAND: 192.168.0.16:ls
PLAIN TEXT COMMAND ls
ENCRYPTED COMMAND �y
ls
.
Sent 1 packets.
ENCRYPTED OUTPUT: ��

�a�%]�V���B���a+��D��▯���+ ^b▯�▯
DECRYPTED OUTPUT:
 configFile.txt
monitor
parseConfig.py
parseConfig.pyc
Server.py

ENTER COMMAND: 192.168.0.16:iptables -L
PLAIN TEXT COMMAND iptables -L
ENCRYPTED COMMAND �-▯@J#�SS
iptables -L
.
Sent 1 packets.
.
Sent 1 packets.
.
Sent 1 packets.
ENCRYPTED OUTPUT: �<▯▯�B��_W����7@▯�x��▯�p<�X
                                                 �  ����▯;��▯�3¿��F��Q!�;y8� �0(����e��
▯�ú�0�c���%���.u▯a▯▯�▯▯�▯���;Y8�fz�.▯�,�!+o���Z�}~�;�0�8PY���>▯���DX�▯�2K��e�op�
O����Tz�C����t%�=�(��T�i�$v�7�nx�lE�jC,�▯al@�.�D▯      �"�0i�)�▯dJ05[G%����|�z�4���2�C
q4T▯�P�9(e
DECRYPTED OUTPUT:
 Chain INPUT (policy DROP)
target     prot opt source               destination

Chain FORWARD (policy DROP)
target     prot opt source               destination

Chain OUTPUT (policy DROP)
target     prot opt source   DdB�,,�����L��r�nation

ENTER COMMAND: 192.168.0.16:
```

Figure 4.1: Encrypted and decrypted commands



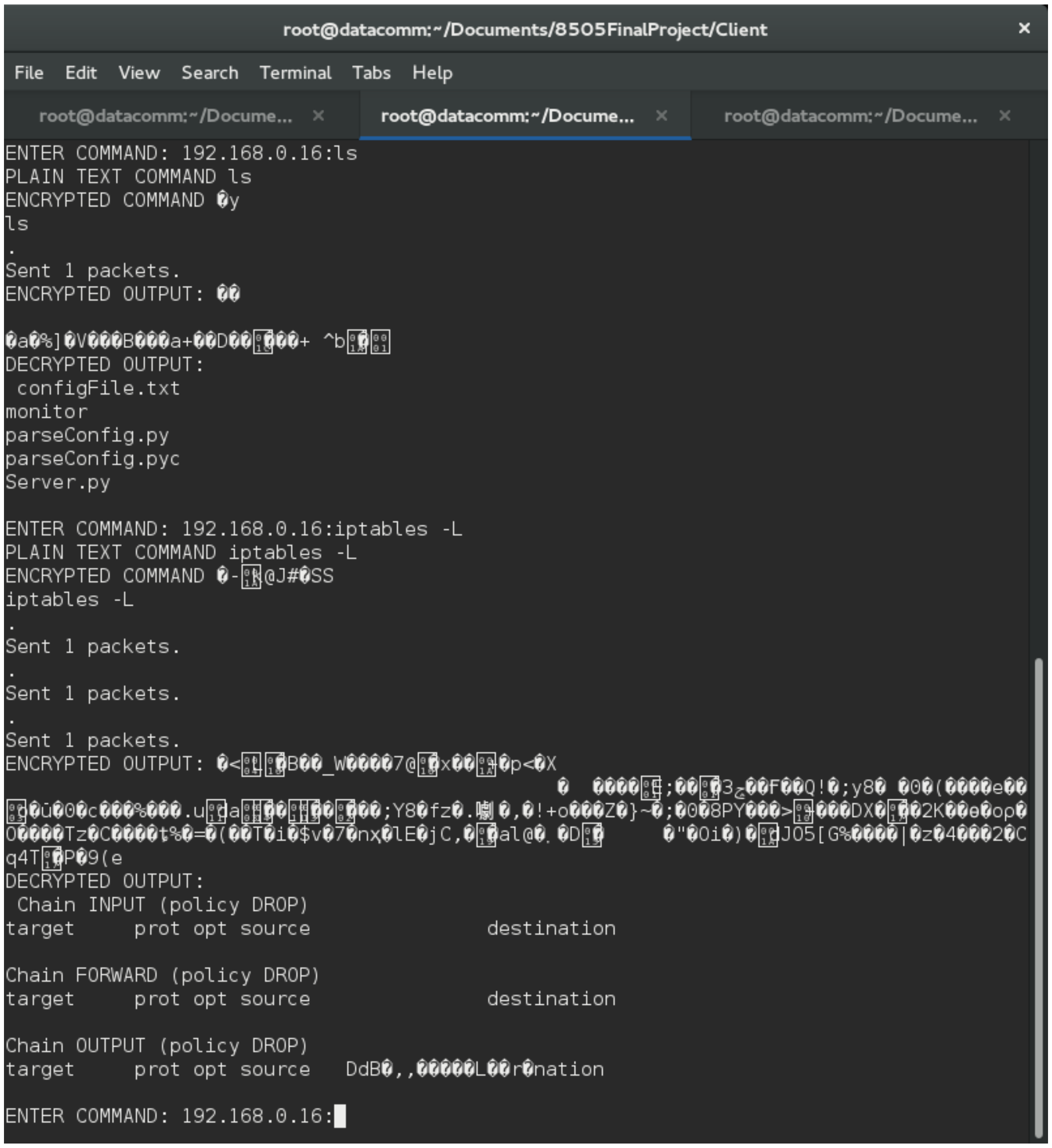
```
[root@datacomm Server]# python Server.py
WARNING: No route found for IPv6 destination :: (no default route?)
ENCRYPTED: �y
DECRYPTED: ls
```

Figure 4.2: Encrypted and decrypted commands



```
ENCRYPTED: �-�@J#�SS
DECRYPTED: iptables -L
```

Figure 5: Shows the output



root@datacomm:~/Documents/8505FinalProject/Client
File Edit View Search Terminal Tabs Help
root@datacomm:~/Docume...
root@datacomm:~/Docume...
root@datacomm:~/Docume...

```
ENTER COMMAND: 192.168.0.16:ls
PLAIN TEXT COMMAND ls
ENCRYPTED COMMAND �y
ls
.
Sent 1 packets.
ENCRYPTED OUTPUT: ��

�a�%]�V���B���a+��D����+ ^b��
DECRYPTED OUTPUT:
 configFile.txt
monitor
parseConfig.py
parseConfig.pyc
Server.py

ENTER COMMAND: 192.168.0.16:iptables -L
PLAIN TEXT COMMAND iptables -L
ENCRYPTED COMMAND �-�@J#�SS
iptables -L
.
Sent 1 packets.
.
Sent 1 packets.
.
Sent 1 packets.
ENCRYPTED OUTPUT: �<L�B��_W����7@�x���p<�X
                                               �  ����;���3��F��Q!�;y8� �0�(����e��
�g�ū�0�c���%���.uda��������;Y8�fz�.�,�!+o���Z�}~�;�0�8PY���>���DX��2K��e�op�
0����Tz�C����t%�=�(��T�i�$v�7�nx�lE�jC,��al@�.�D�     �"�0i�)�dJ05[G%����|�z�4���2�C
q4T�P�9(e
DECRYPTED OUTPUT:
 Chain INPUT (policy DROP)
target     prot opt source               destination

Chain FORWARD (policy DROP)
target     prot opt source               destination

Chain OUTPUT (policy DROP)
target     prot opt source   DdB�,,�����L��r�nation

ENTER COMMAND: 192.168.0.16:
```

Figure 6: File creation

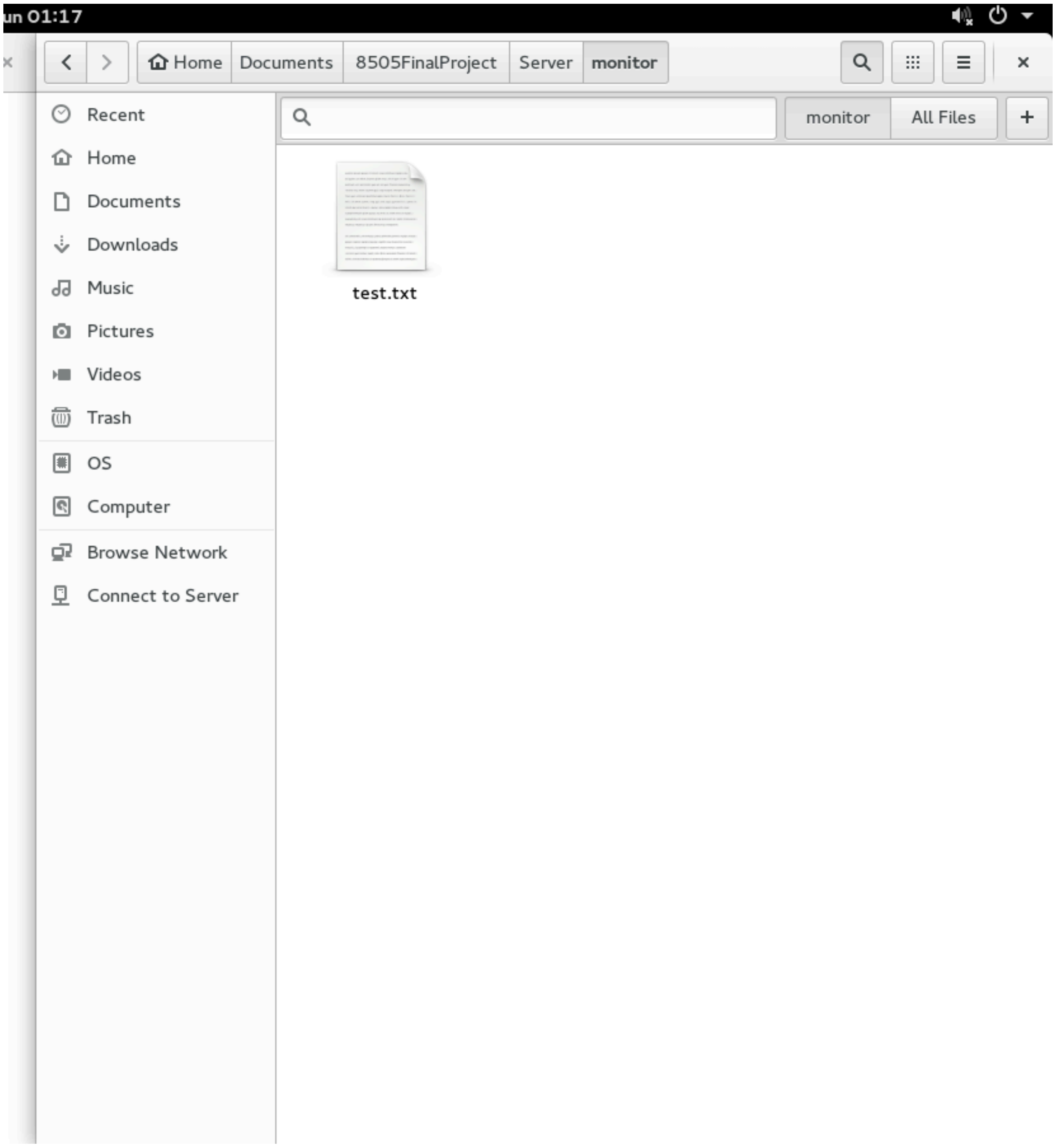
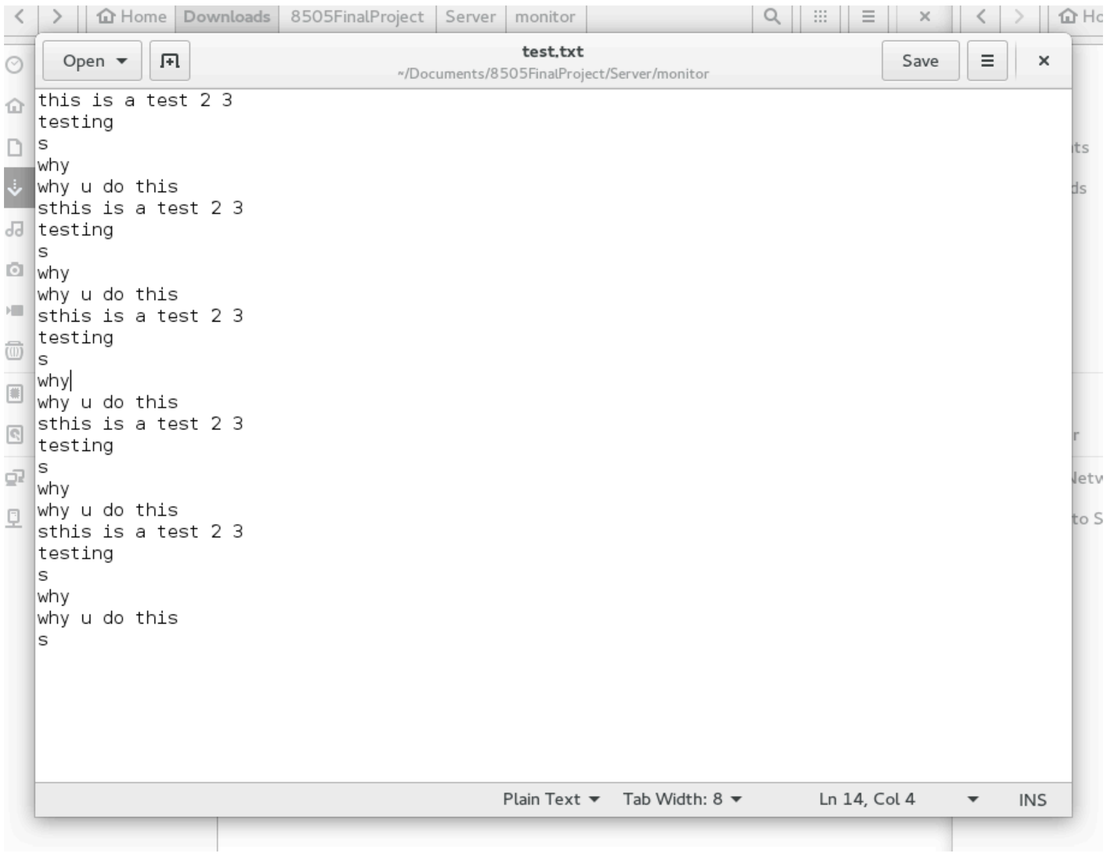


Figure 6.2: Shows the data to be compared to figure 7.3



```
this is a test 2 3
testing
s
why
why u do this
sthis is a test 2 3
testing
s
why
why u do this
sthis is a test 2 3
testing
s
why
why u do this
sthis is a test 2 3
testing
s
why
why u do this
sthis is a test 2 3
testing
s
why
why u do this
s
```

Figure 7.1: Shows empty folder

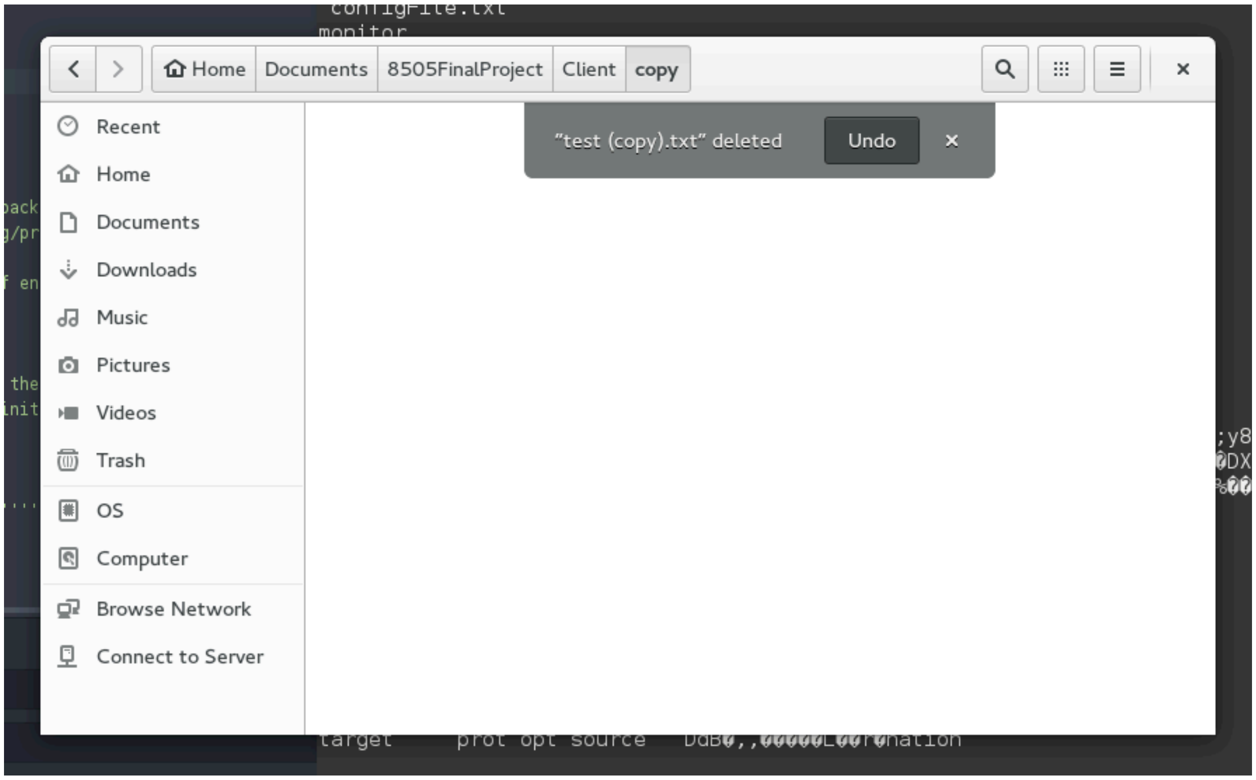
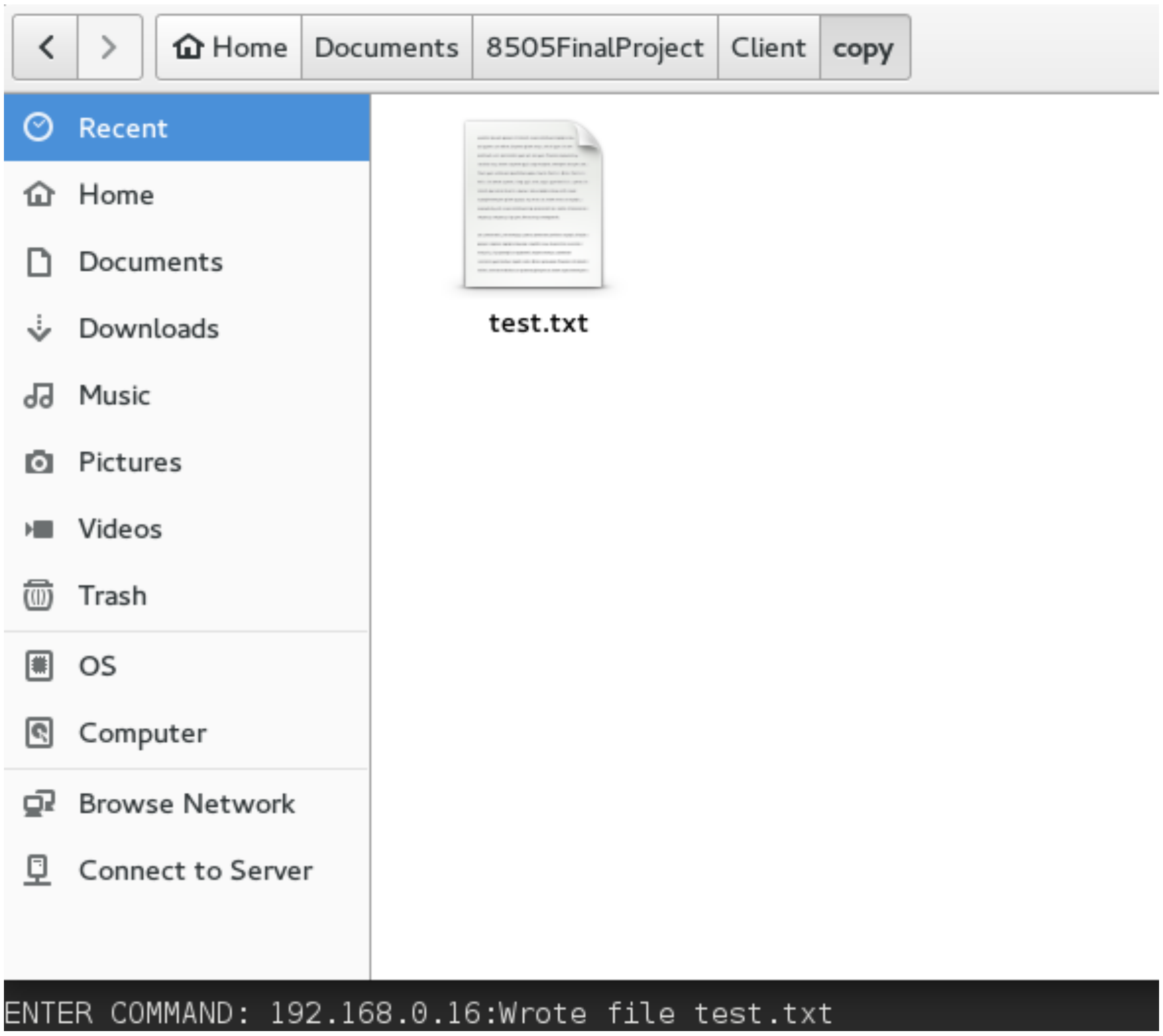
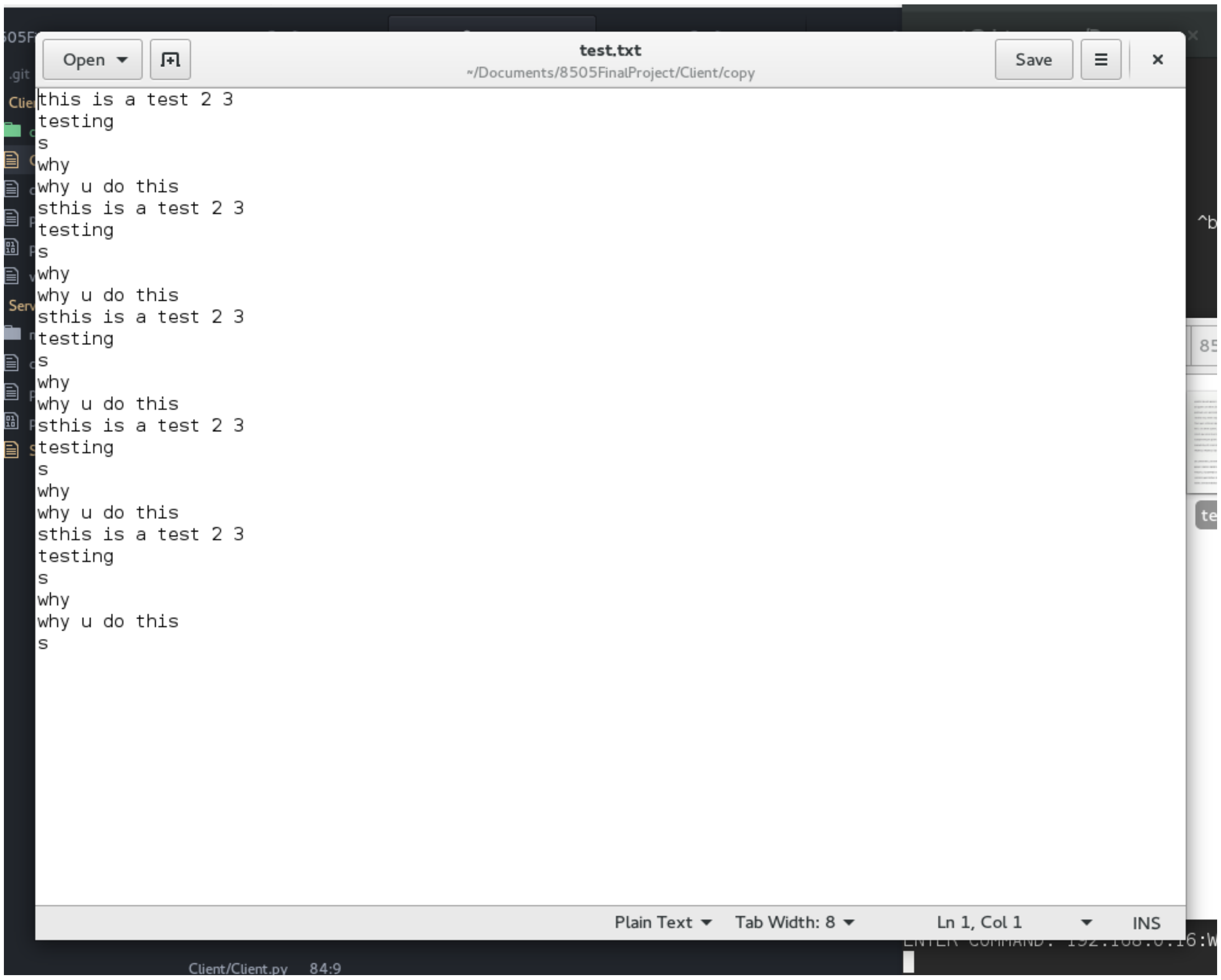


Figure 7.2: Shows the file re-created with the correct name



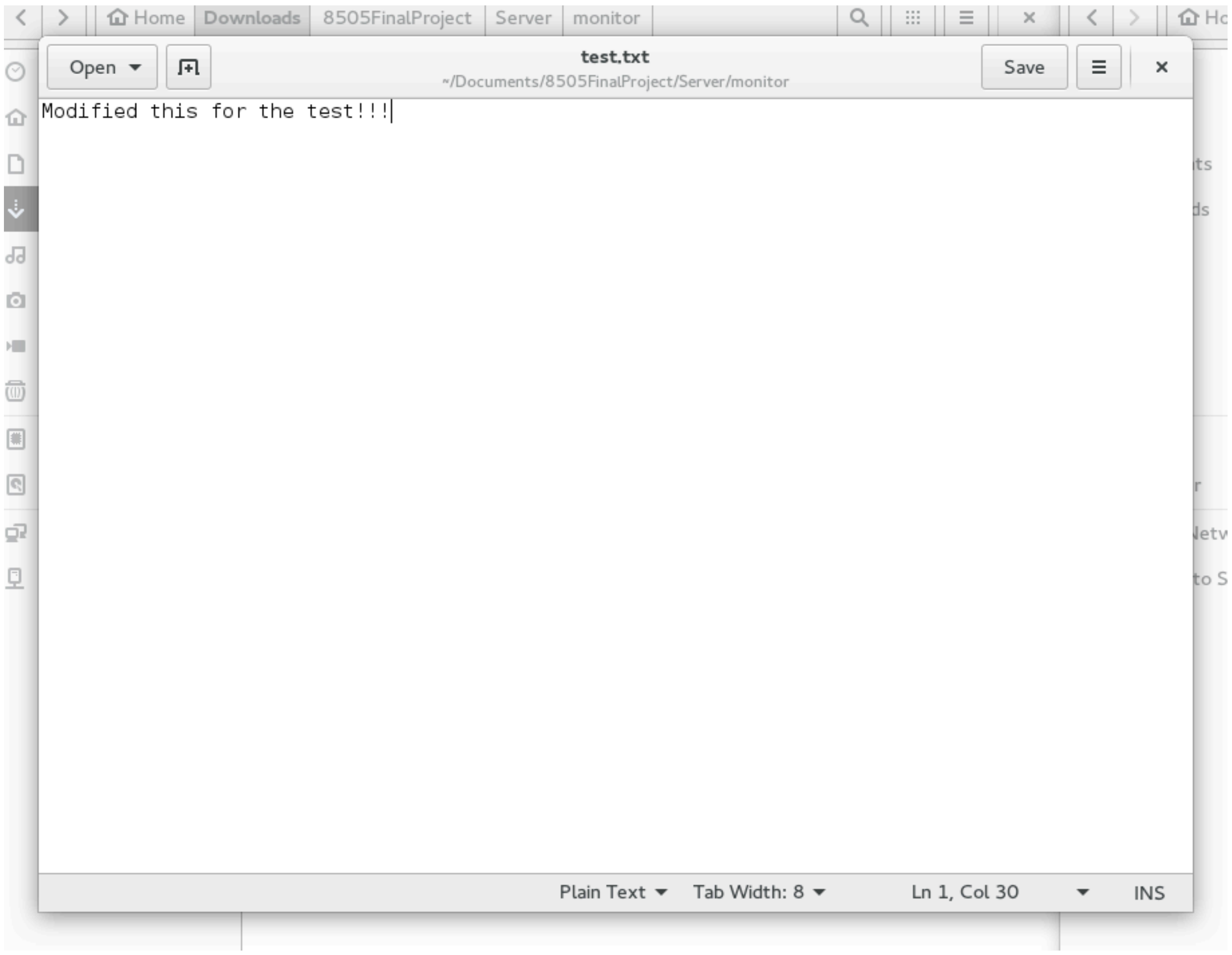
```
ENTER COMMAND: 192.168.0.16:Wrote file test.txt
```

Figure 7.3: Shows the data that matches with figure 6.2



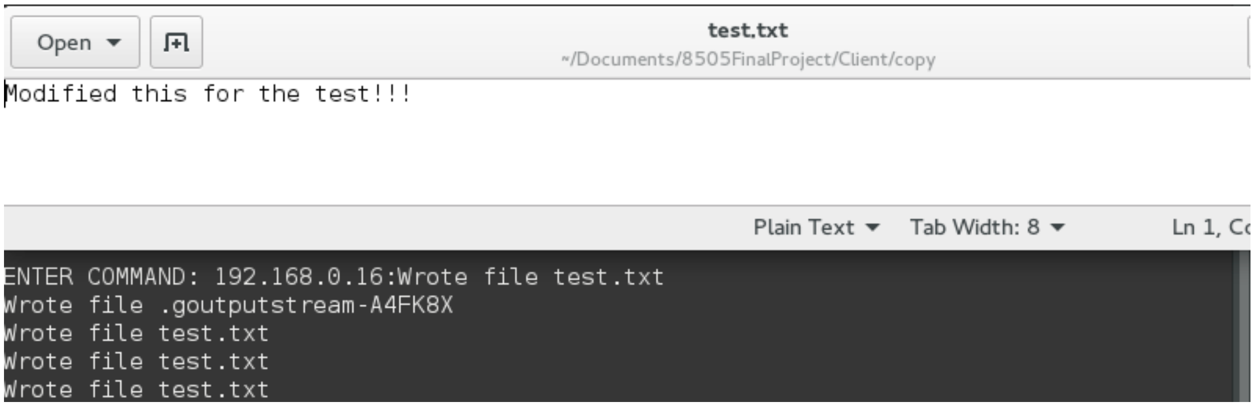
```
this is a test 2 3
testing
s
why
why u do this
sthis is a test 2 3
testing
s
why
why u do this
sthis is a test 2 3
testing
s
why
why u do this
sthis is a test 2 3
testing
s
why
why u do this
sthis is a test 2 3
testing
s
why
why u do this
s
```

Figure 8: Changed the data inside the text file



```
Modified this for the test!!!
```

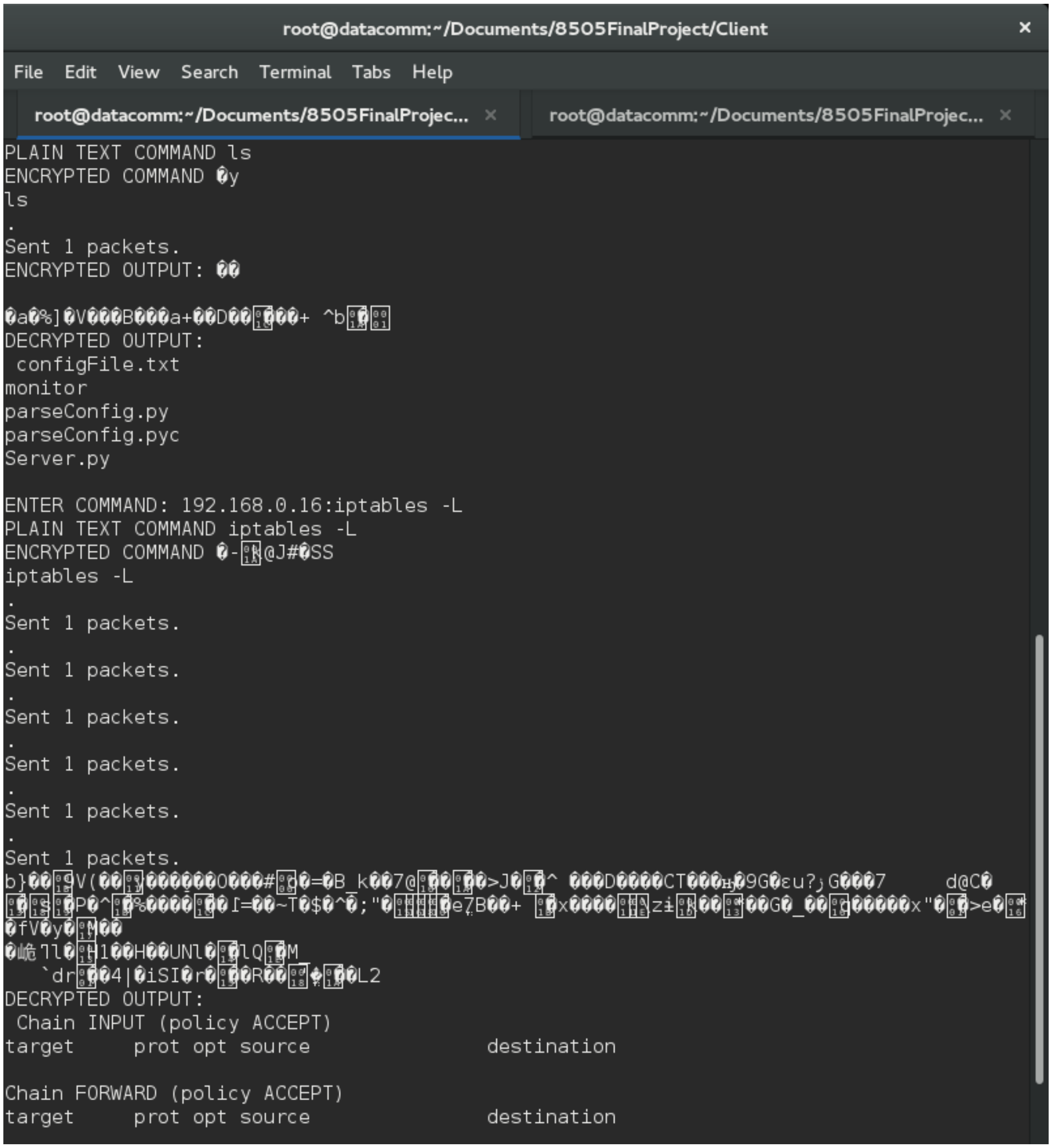
Figure 9:



```
Modified this for the test!!!
```

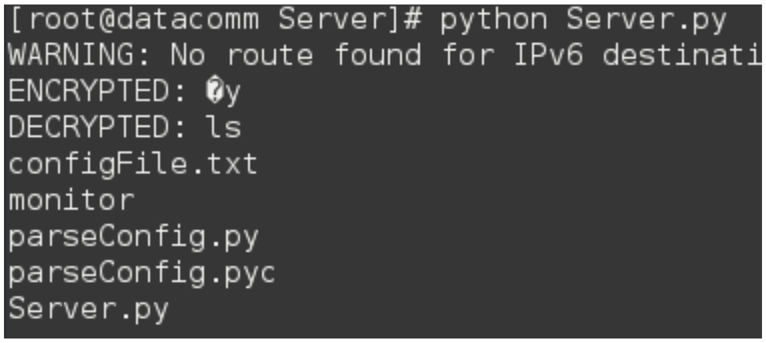
```
ENTER COMMAND: 192.168.0.16:Wrote file test.txt
Wrote file .goutputstream-A4FK8X
Wrote file test.txt
Wrote file test.txt
Wrote file test.txt
```

Figure 10: Sending encrypted commands (UDP)


root@datacomm:~/Documents/8505FinalProject/Client
File Edit View Search Terminal Tabs Help
root@datacomm:~/Documents/8505FinalProjec...
root@datacomm:~/Documents/8505FinalProjec...

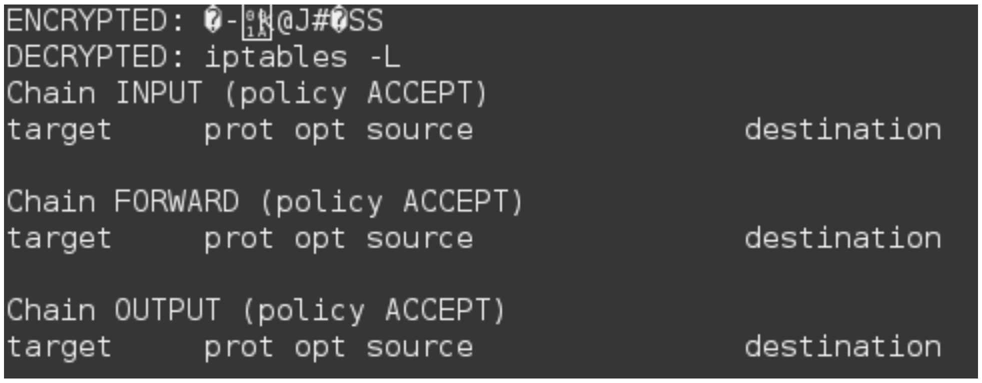
```
PLAIN TEXT COMMAND ls
ENCRYPTED COMMAND �y
ls
.
Sent 1 packets.
ENCRYPTED OUTPUT: ��

�a�%]�V���B���a+��D������+ ^b���
DECRYPTED OUTPUT:
 configFile.txt
monitor
parseConfig.py
parseConfig.pyc
Server.py

ENTER COMMAND: 192.168.0.16:iptables -L
PLAIN TEXT COMMAND iptables -L
ENCRYPTED COMMAND �-�k@J#�SS
iptables -L
.
Sent 1 packets.
.
Sent 1 packets.
.
Sent 1 packets.
.
Sent 1 packets.
.
Sent 1 packets.
.
Sent 1 packets.
b}���9V(���y��������0����#�g�=�B_k��7@�������>J���^ ���D����CT���Ҥ�9G�εu?ⱼG���7     d@C�
����s��P�^��%������Ɪ=��~T�$�^�;"������e7B��+ ��x���������zi��k�����G�_����g�����x"����>e���
�fV�y��M��
�岷ᄀl���H1��H��UNl���lQ��M
   `dr���4|�iSI�r����R�����L2
DECRYPTED OUTPUT:
 Chain INPUT (policy ACCEPT)
target     prot opt source               destination

Chain FORWARD (policy ACCEPT)
target     prot opt source               destination
```

Figure 11.1: Shows the encrypted and decrypted command



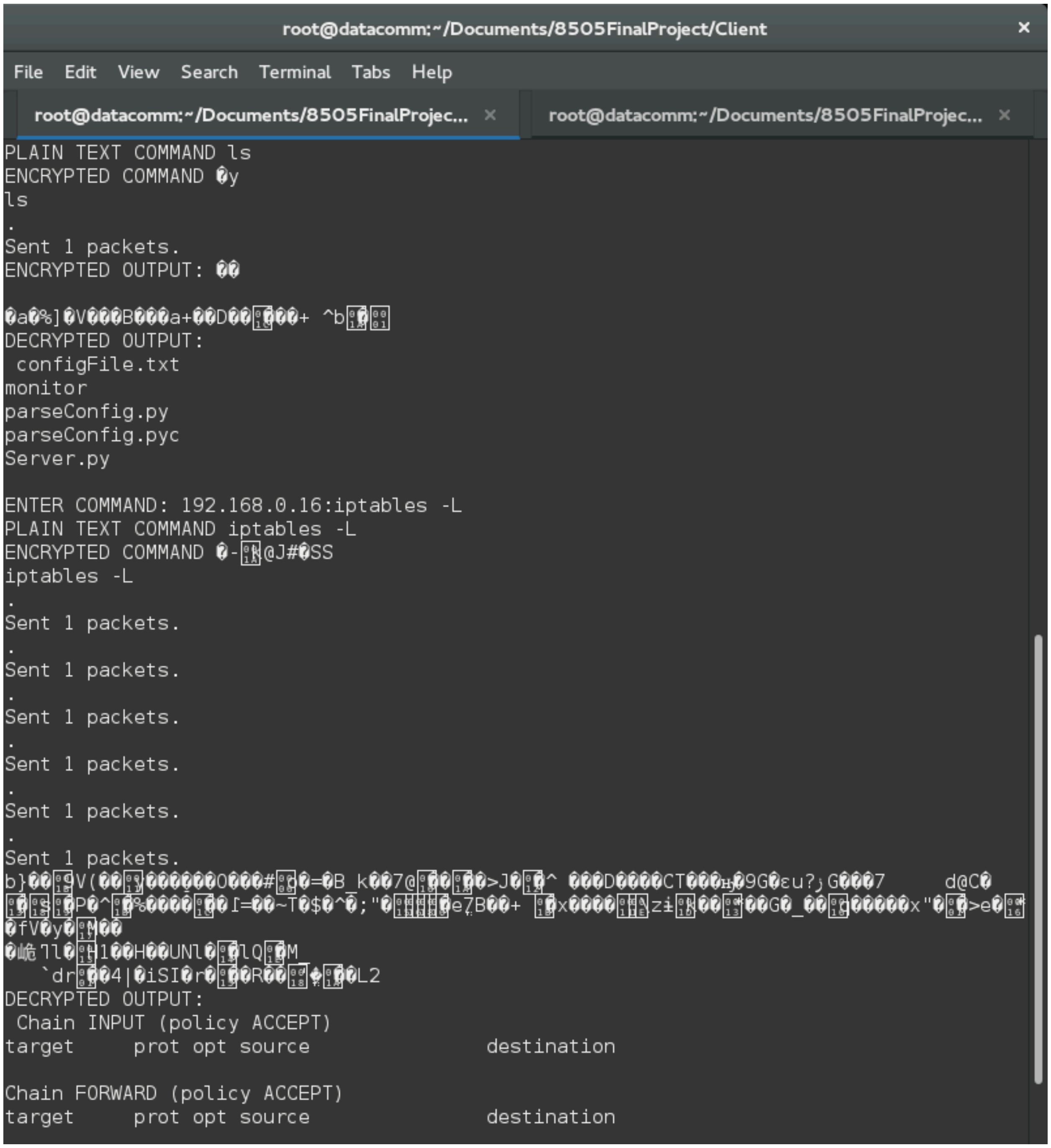
```
[root@datacomm Server]# python Server.py
WARNING: No route found for IPv6 destinati
ENCRYPTED: �y
DECRYPTED: ls
configFile.txt
monitor
parseConfig.py
parseConfig.pyc
Server.py
```

Figure 11.2: Shows the encrypted and decrypted command



```
ENCRYPTED: �-[illegible]@J#�SS
DECRYPTED: iptables -L
Chain INPUT (policy ACCEPT)
target     prot opt source               destination

Chain FORWARD (policy ACCEPT)
target     prot opt source               destination

Chain OUTPUT (policy ACCEPT)
target     prot opt source               destination
```

Figure 12: Shows the encrypted output and the decrypted output


root@datacomm:~/Documents/8505FinalProject/Client
File Edit View Search Terminal Tabs Help
root@datacomm:~/Documents/8505FinalProjec...
root@datacomm:~/Documents/8505FinalProjec...

```
PLAIN TEXT COMMAND ls
ENCRYPTED COMMAND �y
ls
.
Sent 1 packets.
ENCRYPTED OUTPUT: ��

�a�%]�V���B���a+��D��[illegible]���+ ^b[illegible]�[illegible]
DECRYPTED OUTPUT:
 configFile.txt
monitor
parseConfig.py
parseConfig.pyc
Server.py

ENTER COMMAND: 192.168.0.16:iptables -L
PLAIN TEXT COMMAND iptables -L
ENCRYPTED COMMAND �-[illegible]k@J#�SS
iptables -L
.
Sent 1 packets.
.
Sent 1 packets.
.
Sent 1 packets.
.
Sent 1 packets.
.
Sent 1 packets.
.
Sent 1 packets.
b}��[illegible]9V(��[illegible]y��������0����#[illegible]g�=�B_k��7@[illegible]��[illegible]��>J�[illegible]�^ ���D����CT���ӈ�9G�εu?;G���7     d@C�
[illegible]�[illegible]s[illegible]�P�^[illegible]�%����[illegible]��I=��~T�$�^�;"�[illegible]�e7B��+ [illegible]�x����[illegible]\zi[illegible]k��[illegible]��G�_��[illegible]g�����x"�[illegible]�>e�[illegible]
�fV�y�[illegible]M��
�峗7l�[illegible]H1��H��UNl�[illegible]�lQ[illegible]�M
   `dr[illegible]��4|�iSI�r�[illegible]��R��[illegible]�[illegible]��L2
DECRYPTED OUTPUT:
 Chain INPUT (policy ACCEPT)
target     prot opt source               destination

Chain FORWARD (policy ACCEPT)
target     prot opt source               destination
```

Pseudo-code

Client.py

```
1. Read values from config file.
2. Create two processes, one for sending / receiving commands, the second will be listening for file
   transmissions.
```

<u>Sending Commands</u>

```
1. Prompt user for command
2. Encrypt the command
3. Break command down into ASCII and then binary
4. Depending on the protocol, chunk the command into appropriate sizes.
    a. E.g. If using TCP, we will stuff the data in the sequence number which can carry 32 bits)
    b. If using UDP, stuff into source port which can carry 16 bits
5. For each chunk, craft a packet.
    a. Ensure that packet contains our "signature" (TTL of 71 & Password in the payload)
    b. Add some sequencing options to the packet (a UID for each packet in the transmission, a
       position number and total)
6. Send each packet.
7. Listen for response.
```

<u>Listening for command responses</u>

```
1. Handle all IP packets
2. Check packet to see if it matches characteristics (e.g. TTL of 71, and password in payload)
3. Grab data from covert channel fields.
4. Add each message to a list (store by UID)
    a. After adding , check list to see if all messages in that transmission have arrived
         i. If it has, reconstruct the bits to ASCII to a human readable string.
        ii. Decrypt the output.
```

<u>Listen for files</u>

```
1.  Handle all IP packets coming in on port 80
2. Check packet to see if it matches characteristics (e.g. TTL of 71, and password in payload)
3. Grab data from covert channel fields.
4. Add each message to a list (store by UID)
    a. After adding, check list to see if all messages in that transmission have arrived
1. If it has, reconstruct the binary.
2. Decrypt the binary.
3. Write the binary to a file.
```

Server.py

1. Read values from config file.
2. Start two processes, one for receiving commands/sending output and the one for for monitoring a file directory.

Listening For Commands

1. Handle all IP packets
2. Check packet to see if it matches characteristics (e.g. TTL of 71, and password in payload)
3. Grab data from covert channel fields.
4. Add each message to a list (store by UID)
    - **a.** After adding , check list to see if all messages in that transmission have arrived
        - **i.** If it has, reconstruct the bits to ASCII to a human readable string.
        - **ii.** Decrypt the output.
        - **iii.** The output is the command, so execute the command.
            1. Get the output of the command then Encrypt the message
            2. Break message down into ASCII and then binary
            3. Depending on the protocol, chunk the message into appropriate sizes.
                - a. E.g. If using TCP, we will stuff the data in the sequence number which can carry 32 bits)
                - b. If using UDP, stuff into source port which can carry 16 bits
            4. For each chunk, craft a packet.
                - a. Ensure that packet contains our “signature” (TTL of 71 & Password in the payload)
                - b. Add some sequencing options to the packet (a UID for each packet in the transmission, a position number and total)
            5. Send each packet.

Monitor Files

1. Observe directory for changes.
2. On change, raise an event.
3. Handle event.
    - a. If it’s a deletion, do nothing.
    - b. If it’s an addition (which covers file renames as well) then
        - i. Get file.
        - ii. Break file down into binary.
        - iii. Encrypt that binary.
            1. Depending on the protocol, chunk the message into appropriate sizes.
                - a. E.g. If using TCP, we will stuff the data in the sequence number which can carry 32 bits)

   b. If using UDP, stuff into source port which can carry 16 bits
2. For each chunk, craft a packet.
   a. Ensure that packet contains our “signature” (TTL of 71 & Password in the payload)
   b. Add some sequencing options to the packet (a UID for each packet in the transmission, a position number and total)
3. Send each packet.