Covert Communication Application

Comp 8505 Final Assignment

Shane Spoor

Mat Siwoski

# Table of Contents

[**Table of Contents**](#_o8m9snahw1ua) **1**

[**General Information**](#_yqugbo708dtd) **3**

[Overview](#_qfzawe4xvwfn) 3

[Features](#_m64m7ue2f6yz) 3

[Server](#_p3uy98mhr78c) 3

[Client](#_fru5ljpqwz3g) 3

[Constraints](#_dvt5rt6g8v8d) 3

[Dependencies](#_ahk1hog4usxj) 4

[Running the Application](#_u9hgsa4rjz0d) 4

[**High Level Design**](#_rj7l34t03p4a) **6**

[**State Diagrams**](#_ii9o88qoaxhr) **7**

[Backdoor](#_sf51e5uh17z8) 7

[Remote Client Command Handling](#_l913p5rf061n) 7

[**Pseudocode**](#_2e1rggty8j69) **8**

[Main Application](#_e0jwzznli9kg) 8

[Backdoor Server](#_ghl0ko1fb0p) 8

[Generic Backdoor Server](#_jmsn04kkoqvv) 8

[TCP Backdoor Server](#_hx5xlikkby9j) 10

[UDP Backdoor Server](#_psnq511owwu7) 11

[Backdoor Client](#_hqgo8db4shqp) 11

[Generic Backdoor Client](#_5kk95eqr1qqo) 11

[TCP Backdoor Client](#_9cb1h5hr3fji) 12

[UDP Backdoor Client](#_56pv3bv1gqib) 13

[Command](#_ltl8wk55cby2) 13

[Command](#_lpy6h15r7s7n) 13

[Shell Command](#_24rnxecjbrg5) 13

[Watch Command](#_jx1316lqyakg) 14

[Utility](#_64o8nee71kba) 15

[**Testing**](#_k38e3san07lz) **16**

[Test 1: Argument Help](#_m2n1d1tto84) 16

[Test 2: TCP Server Shell Commands](#_nq9yt8lf2j5q) 16

[Test 3: UDP Server Shell Commands](#_h4ersz3fvm5f) 17

[Test 4: UDP Server Watch Commands](#_7pzujc3ih4g9) 19

[Test 5: TCP Server Watch Commands](#_1uw1p2irnomb) 19

[Test 6: Masked Process](#_1v2y9p9wnqit) 20

[Test 7: Rapidly Send Commands Between the Server/Client](#_pcl5nhwczhry) 20

[Test 8: Firewall/Port Knocking rules](#_sf0yrjb7ia8q) 21

# General Information

## Overview

This project involves designing and implementing a complete covert application that will allow a user to open a port (that is otherwise closed) on a firewall and communicate with a “disguised” backdoor application. The backdoor application will accept commands and execute them; the results of the command execution will be sent back to the remote client application.

## Features

The application has the following features:

### Server

* Accepts packets regardless of Firewall rules once a service port is opened
* Runs as a masked process
* Packets are only accepted upon authentication using an encrypted password
* Packets extract and run either a SHELL or WATCH command
  + File changes are sent back to the attacker
  + Directory changes are sent back to the attacker
* Results of the commands are sent back to the attacker

### Client

* Connect and control victim machine
* Accepts and decodes knock sequence
* Provide access to the port and service that contains encrypted data containing results of either the shell commands or watch commands
* Able to execute commands on the victim machine
* Commands are encrypted
* Able to watch for file changes or directory changes
  + File changes are sent back to the attacker
  + Directory changes are sent back to the attacker

## Constraints

* This application must be implemented for either Linux or Windows.
* The attacker must be able to set the IP, ports and whether to use TCP or UDP for the attack

## Dependencies

The application requires the following Python packages to be installed:

* iNotify
* Scapy
* pycrypto
* setProctitle
* libpcap

In the event that these packages are not installed, run the following commands as root to install them:

pip install inotify

pip install scapy

pip install pycrypto

pip install setproctitle

Pip install libpcap

## Running the Application

First, install the dependencies above using your Python 2 package manager of choice.

To run the backdoor server:

python main.py server listen port client port [-m process name] [-p password] [-k aes key]

where

* server is the literal string server
* listen port is the port on which the server will listen for backdoor client connections (1-65535 inclusive)
* client port is the port to which the server will send the client's results (1-65535 inclusive)
* process name will replace the backdoor server's process name so that it's harder to find
* password is a password added to packets so that the server can tell if a packet bound for the listen port is a client trying to connect and so that the client and server can ensure that packets were properly decrypted
* aes key is the key to use for AES encryption (applied to all packets except the initial client connection)

To run the backdoor client:

python main.py client listen port server port -s server host [-p password] [-k aes key]

where

* client is the literal string client
* listen port is the port on which the client will listen for backdoor server command results (1-65535 inclusive)
* server port is the port on which the server will listen for client connections (1-65535 inclusive)
* server host is the backdoor server's host name or IP (mandatory when the program is used in client mode even though it's technically "optional")
* password and aes key: same as the server documentation above

The client will continuously prompt for commands, send them to the server, and display their results. To exit the prompt, type Ctrl+D or Ctrl+C

# 

# 

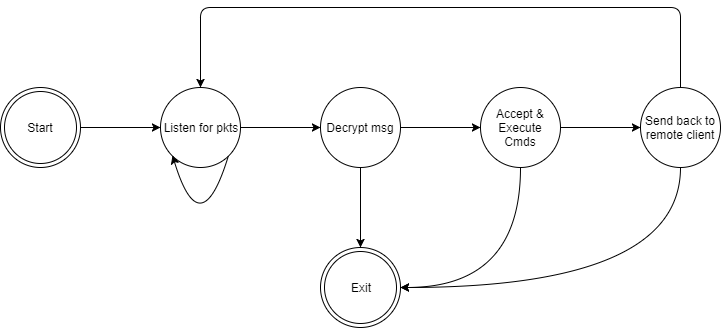
# High Level Design

|  |  |
| --- | --- |
| **Design Features** | **Description** |
| Encryption | Both the client and the server should be able to encrypt and decrypt data. Only packets that have been authenticated will be accepted by the application. |
| Port Knocking | Port knocking will be used to access and deliver the encrypted data. |
| Exfiltration | Attackers will be able to search (or watch a directory or file for events) for a particular file and send back the file contents covertly. If a file is changed, it is sent automatically. Upon completion of the exfiltration, the ports are automatically closed. User is able to specify how/when the ports are closed as well. |
| Disguised Process | The application will run as a disguised process. |
| Command Handler | Commands will be sent and run on the victim machine. Results of the data will be sent back to the attacker upon execution. |
| Ignore Firewall | The application will ignore the Firewall. |
| File/directory modifications | If there is a modification of a file/directory or creation of a file/directory, the file/directory contents are sent to the attacker. |

# 

# State Diagrams

## Backdoor



# 

## Remote Client Command Handling

# RemoteClientCmds.png

# Pseudocode

## Main Application

This project includes both TCP and UDP backdoors. The listening server will be specified by command line parameters, but both servers will have the same functionality. This is the Main Application that will accept parameters:

Main():

Parse arguments

Check for validity

If server:

If TCP:

Start TCPBackdoorServer

Else if UDP

Start UDPBackdoorServer

Run server

If Client

If TCP:

Start TCPBackdoorClient

Else if UDP

Start UDPBackdoorClient

Run Client

While True:

Listen for shell commands from the User

Send Shell Command

Receive Shell Command results

Print Shell Command Results

## Backdoor Server

### Generic Backdoor Server

This project includes both TCP and UDP backdoors. The listening server will be specified by command line parameters, but both servers will have the same functionality. This is the Generic Backdoor Server:

BackdoorServer():

Initialize all necessary variables

BackdoorServer.run():

mask\_process()

Create queue

while true:

Listen for client()

Create thread for result queue

while true:

command = Command.from\_stream(self)

if command from SHELL:

result = command.exec(Queue)

send\_result(result)

If command from Watch Command

Create Thread to handle watch command

else:

Break

BackdoorServer.mask\_process():

change process name according to config file

BackdoorServer.port\_knock():

Create the knocked ports array

For each port in the knocked ports array

Send the port knock using Scapy

BackdoorServer.recv():

BackdoorServer.listen():

while true:

packet = sniff for possible authentication packet

if packet is authentication packet:

send response acknowledging authentication

return client information

BackdoorServer.recv\_command():

While true:

read bytes from packets originating from current client

decrypt bytes

If Shell command

Run shell command

Else if Watch Command

Run watch command

Else

end

return decrypted bytes

BackdoorServer.send\_result(result):

Create Covert Socket

For the amount of knock tries

Port knock on created socket

Sleep

If no connection

Break

Create encryptor

Create first packet with payload and get the result length

Encrypt payload

Send payload

If result length > first packet payload

While result length isn’t reached:

Create packets with remaining payload

Encrypt payload

Send payload

Shutdown covert socket

BackdoorServer.result\_queue(queue):

While true:

Get the queues info

Send the result

Close the queued item

### TCP Backdoor Server

This project includes a TCP backdoor.

TCPBackdoorServer()

Initialize all necessary fields

TCPBackdoorServer.Send()  
 Create the packet

Send the packet

TCPBackdoorServer.receive()

Sniff for packets

Return payload

TCPBackdoorServer.listen()

Create a random source port

If authenticated: //check if it’s a client

Enumerate through the packet and get all required info

Sniff for packets

### UDP Backdoor Server

This project includes a UDP backdoor.

UDPBackdoorServer()

Initialize all necessary fields

UDPBackdoorServer.Send()  
 Create the packet

Send the packet

UDPBackdoorServer.receive()

Sniff for packets

Return payload

UDPBackdoorServer.listen()

Create a random source port

If authenticated: //check if it’s a client

Enumerate through the packet and get all required info

## Backdoor Client

### Generic Backdoor Client

As with the backdoor server, there will be different clients for TCP and UDP, but they have the same basic functionality.

BackdoorClient():

Initialize all necessary variables

BackdoorClient.run():

connect()

while there are commands: # user input

command = next command

send(command.to\_bytes())

result = recv\_result()

# do something with result

# close the connection with the backdoor

command = Command(Command.END)

send(command.to\_bytes())

BackdoorClient.recv\_result():

Create listening socket

Listen for packets  
 If packet received:

Decrypt Packet

verify password

Check for payload length

If payload length hasn’t been reached:

Receive remaining payload packets

Decrypt Packet

Reorder the packets

If result equals SHELL

Run Shell Command

Else if result equals Watch Command

Run Watch Command

Else

Quit

BackdoorClient.connect():

BackdoorClient.send(payload):

BackdoorClient.send\_command(bytes):

Create encryptor

Create Payload

Encrypt Payload

Send Payload

### TCP Backdoor Client

This project includes a TCP backdoor client.

TCPBackdoorClient()

Initialize all necessary fields

TCPBackdoorClient.connect():

Insert password into packet for server authentication

Create packet

Send(packet)

TCPBackdoorClient.send(packet):

Create Packet

Send(packet)

### UDP Backdoor Client

This project includes an UDP backdoor client.

UDPBackdoorClient()

Initialize all necessary fields

UDPBackdoorClient.connect():

Insert password into packet for server authentication

Create packet

Send(packet)

UDPBackdoorClient.send(packet):

Create Packet

Send(packet)

## Command

As per the requirements, the application must be able to handle Shell commands from the attacker which includes executing and returning the result of the shell command. As well, the application needs to handle Watch Commands on a directory or a file:

### Command

Command.run():

Command.to\_bytes():

Command.from\_bytes():

### Shell Command

This will handle all the SHELL commands that the attacker chooses to use against the victim.

ShellCommand():

Initialize all necessary fields

ShellCommand.to\_bytes():

Create bytes

ShellCommand.from\_bytes():

Get command length

ShellCommand.run():

Open process

Get Result

Put result on Queue

ShellCommand.Result():

Initialize all necessary fields

ShellCommand.Result.to\_bytes():

Append bytes  
 Create byte

ShellCommand.Result.from\_bytes():

Unpack struct  
 Return result

### Watch Command

This will handle all the WATCH commands that the attacker chooses to use against the victim.

WatchCommand():

Initialize all necessary fields

WatchCommand.to\_bytes():

Create bytes

WatchCommand.from\_bytes():

Get command length

WatchCommand.run():

Check if we are needing to WATCH a file or Directory

Initialize iNotify and make epoll block indefinitely

Add iNotify watch

For each event generated:

If event:

Read contents of the change

Get Result

Add result to queue

WatchCommand.Result():

Initialize all necessary fields

WatchCommand.Result.to\_bytes():

Append bytes  
 Create byte

WatchCommand.Result.from\_bytes():

Unpack struct  
 Return result

## Utility

This project needs to be able to create packets with passwords and payload. We created two functions for this handling: one that creates a 8 byte packet with the password and a 32 byte packet with the payload.

Make\_8\_byte\_String(instr):

Check that the password is divisible by 8 characters

If the password length isn’t divisible by 8:

Pad remaining portion till divisible by 8

Unpack the struct

Repack the struct

Make\_32\_byte\_String(instr):

Check that the payload is 32 characters

If the password length isn’t divisible by 32:

Pad remaining portion till divisible by 32

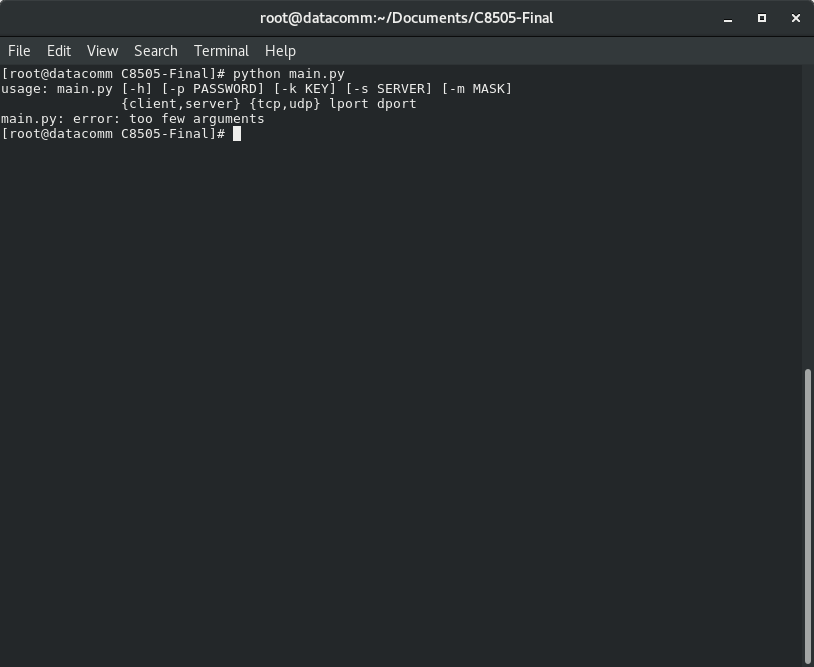
# 

# Testing

|  |  |  |
| --- | --- | --- |
| Test # | Test Description | Result |
| 1 | Display help screen with all available arguments | Passed (Fig. 1) |
| 2 | As a TCP Server, accept connections and handle the Shell Command | Passed (Figs. 2, 3, 4) |
| 3 | As a UDP Server, accept connections and handle the Shell Command | Passed (Fig. 5, 6, 7) |
| 4 | As a UDP Server, accept connections and handle the Watch Command for a File and Directory | Passed (Fig. 8,9) |
| 5 | As a TCP Server, accept connections and handle the Watch Command for a File and Directory | Passed (Fig 10, 11) |
| 6 | Mask the service on the Server (Victim) | Passed (Fig. 12) |
| 7 | Rapidly add commands to the server | Passed (Fig. 13) |
| 8 | Firewall/Port Knock | Passed (Fig. 14, 15) |

## Test 1: Argument Help

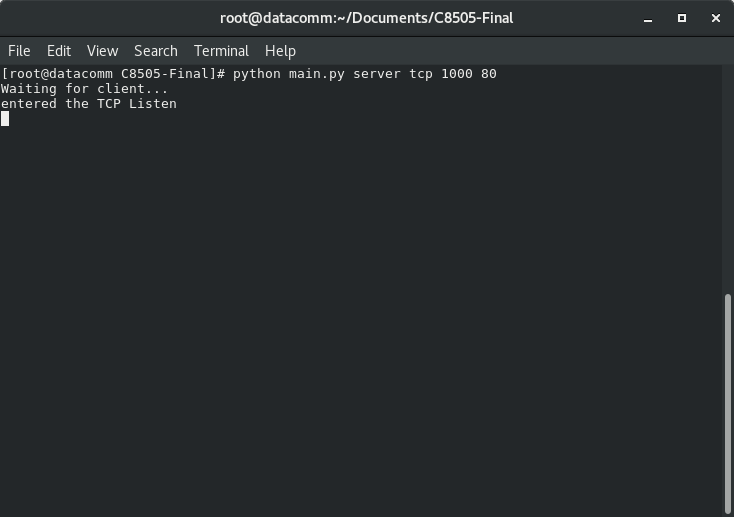
Invoking the program with -h shows a list of the program’s options and allowable values for them



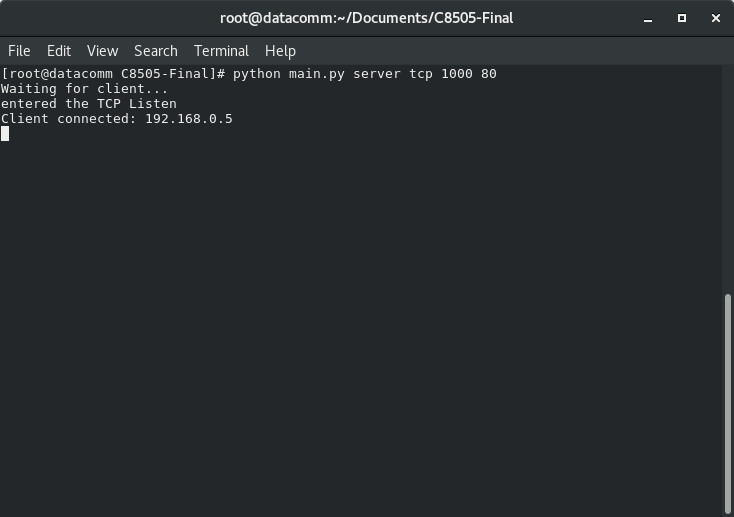
*Fig 1) Arguments for running the application*

## Test 2: TCP Server Shell Commands

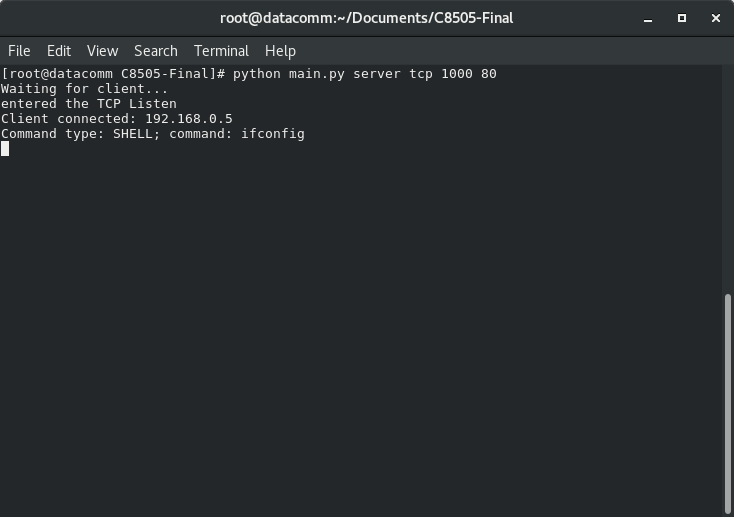
As the TCP Server, the program is able to accept a connection from the client and accept the Shell Commands.



*Fig 2) TCP Shell Server Waiting*



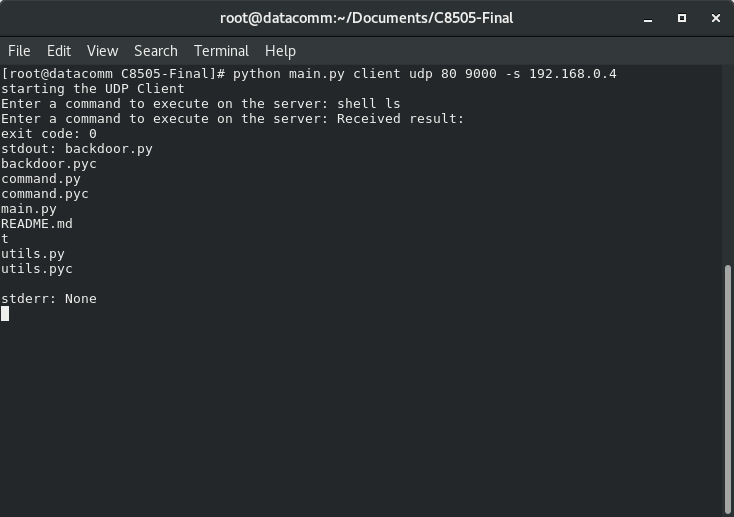
*Fig 3) TCP Server received connection*



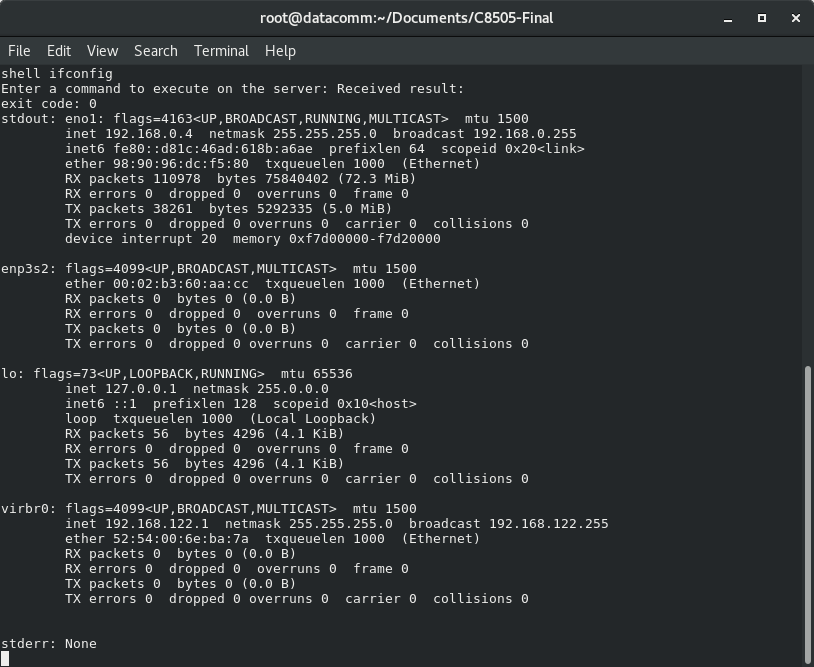
*Fig 4) TCP Server received shell command*

## Test 3: UDP Server Shell Commands

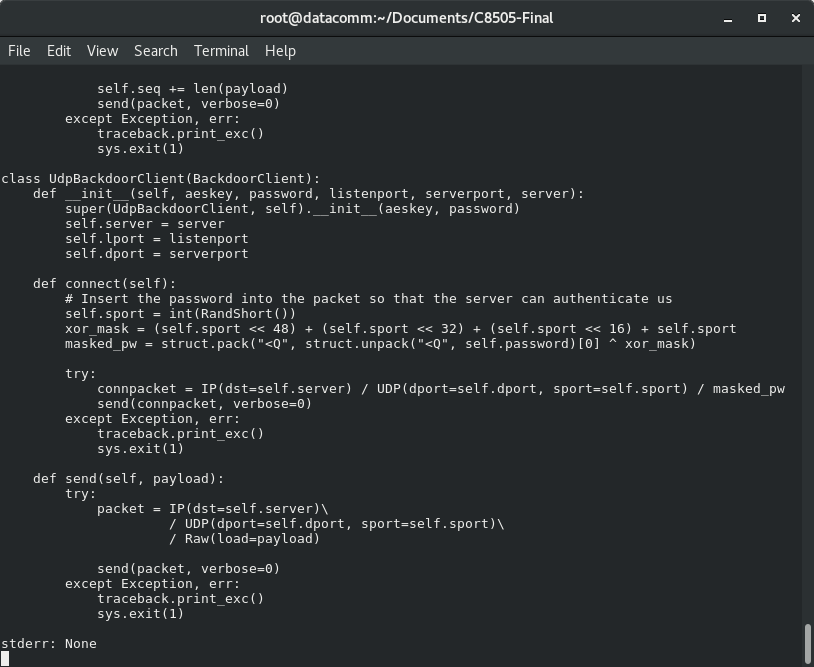
As the TCP Server, the program is able to accept a connection from the client and accept the Shell Commands.



*Fig 5) LS as the command*

**

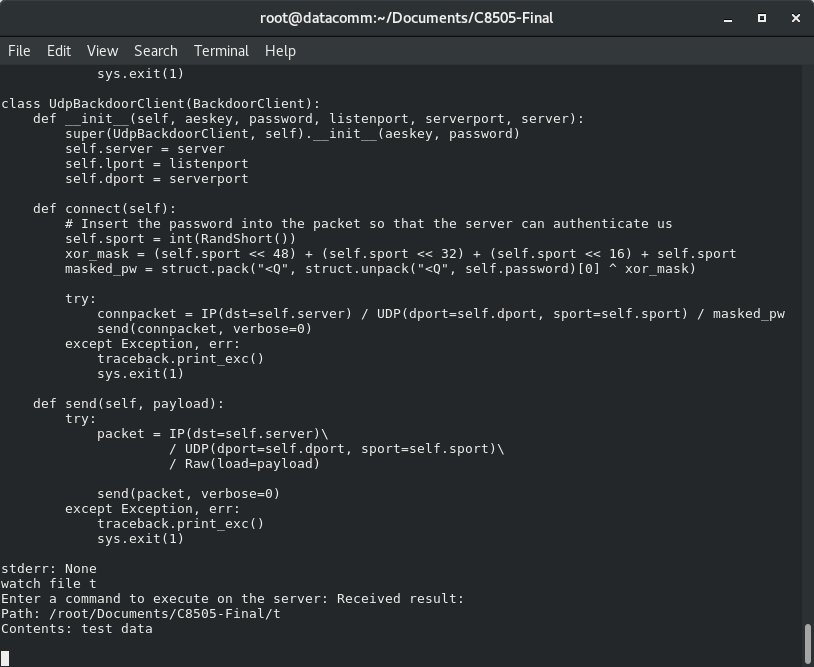
*Fig 6) Ifconfig as the command*

**

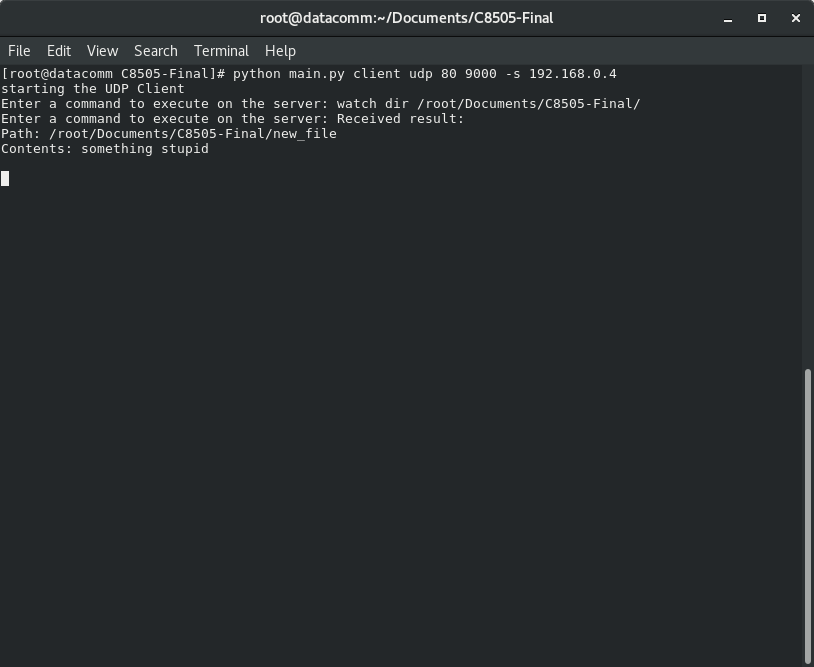
*Fig 7) Cat a file*

## Test 4: UDP Server Watch Commands

As the TCP Server, the program is able to accept a connection from the client and accept the Watch Commands.



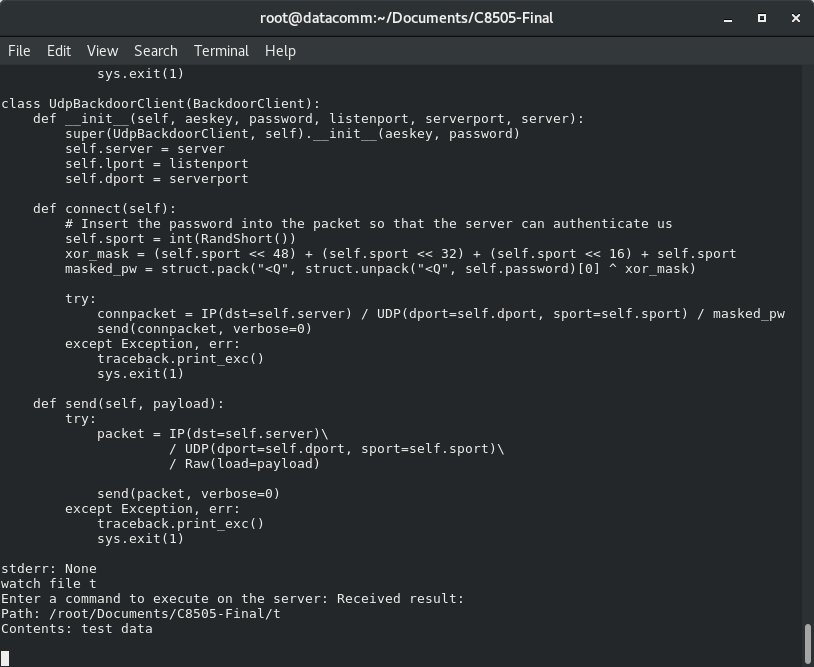
*Fig 8) Watch for a file*



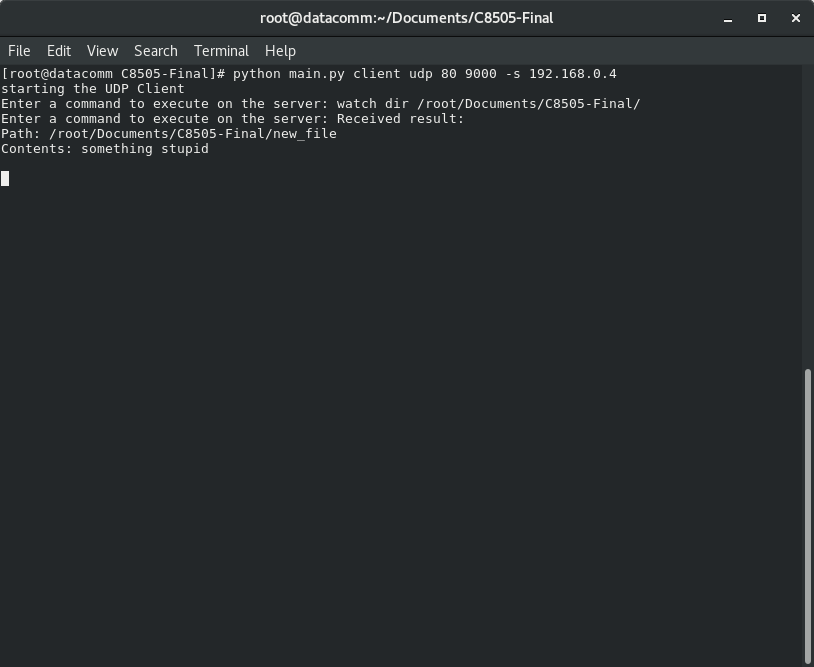
*Fig 9) Watch for a directory*

## Test 5: TCP Server Watch Commands

As the TCP Server, the program is able to accept a connection from the client and accept the Watch Commands.



*Fig 10) Watch for a file*

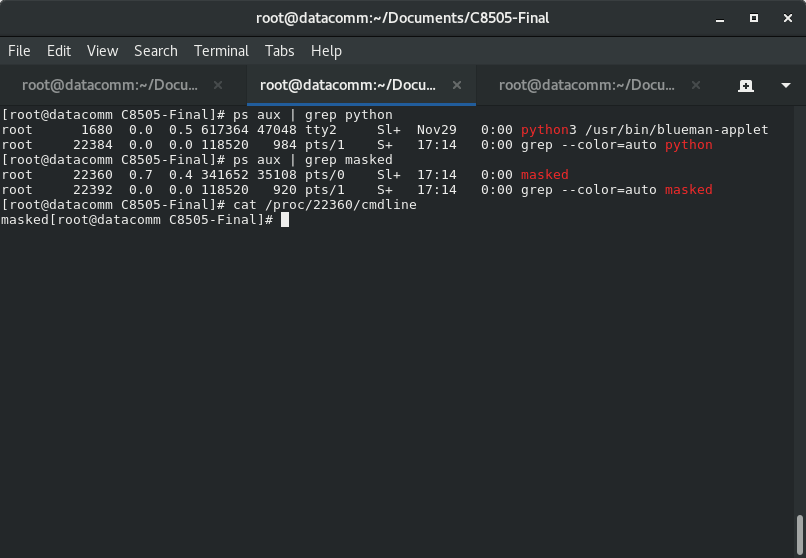


*Fig 11) Watch for a directory*

## 

## Test 6: Masked Process

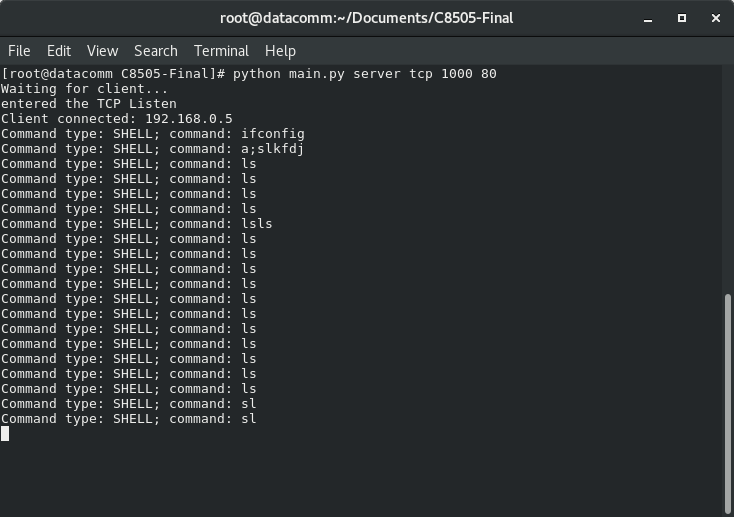
The program runs as a masked process



*Fig 12) Masked Process on the victim*

## Test 7: Rapidly Send Commands Between the Server/Client

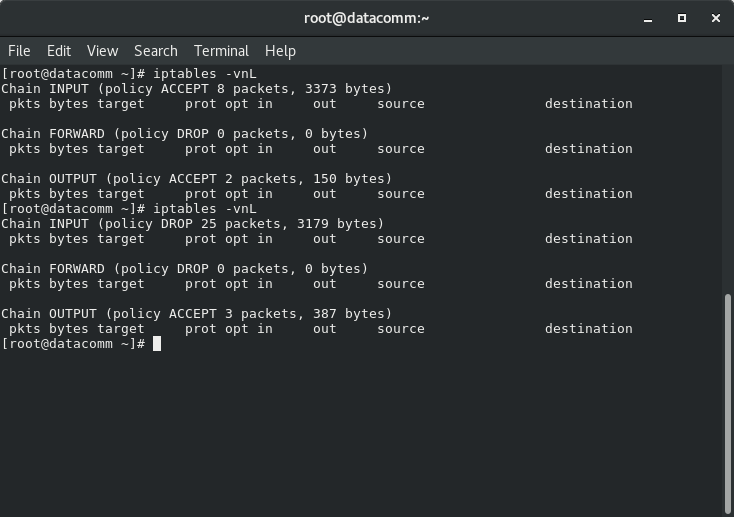
Spam with multiple commands and ensure that the application is able to support the commands.



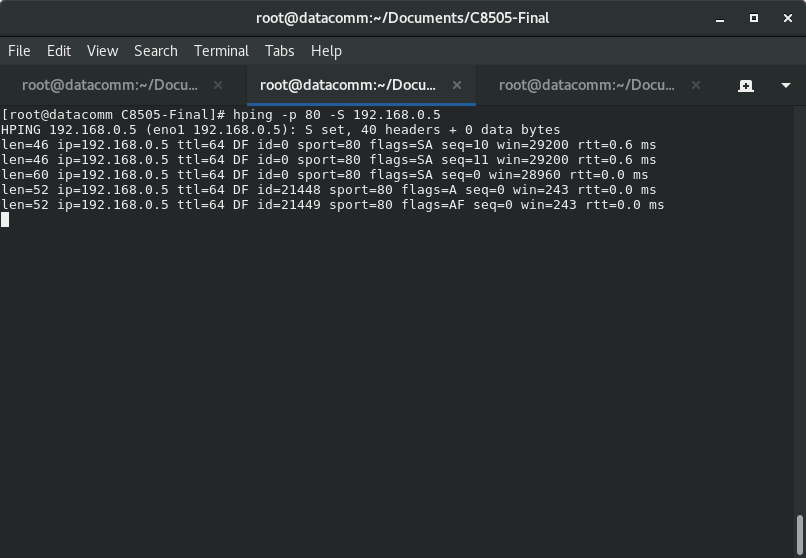
*Fig 13) Spam the client with commands*

## Test 8: Firewall/Port Knocking rules

Application alters firewall for port knock



*Fig 14) Firewall rules altered for port knock*



*Fig 15) hping the client*