COMP8505 Assignment 1

To analyze and critique Craig Rowland’s program and rectify its weaknesses

Xinghua Wei

Frederick Wong

September 27, 2020

Table of Contents

1. Summary

In this assignment, we are going to examine the “covert_tcp.c” program written by Craig Rowland, and identify weaknesses within his program. Then, we are going to introduce our own implementation of a covert channel, and show that the weaknesses found in Craig Rowland’s implementation have been rectified. We will also show steps of how to run Craig’s code and our code. In addition, we will attach a demo video of our implementation in the submission, as well as packet captures from Wireshark to support our test cases.

2. Introduction

A covert channel is a mechanism that allows users to send and receive data without the permission of the system. Attackers often make use of this technique to infiltrate the system of their target, either for retrieving data or modifying data. This can be achieved by embedding the data inside unintended fields of the packet, such as the IP ID field, or the IP TTL field. In our implementation, we have chosen to code our covert channel in Python, and used the TCP flags as a filter to embed our data in the IP TTL field of each packet, without showing any signs of infiltration in the payload of the frame. One key difference between our implementations is that Rowland’s implementation transfers data inside a file, while our implementation transfers data in the input buffer of the terminal. However, this should not affect the purpose of this assignment, which is to study how covert channel works.

3. Method

3.1 Machines used

For this assignment, two Linux Fedora machines are being used, and both of them are connected to the same local network. One of them will be the server listening, and the other will be the client sending data to the server.

Below are the IP addresses of the server and the client in our setup.

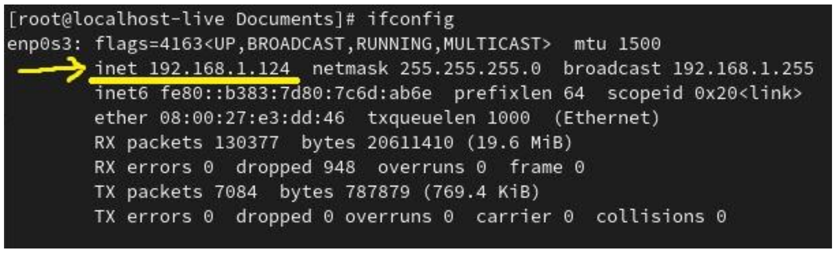
```
Server IP address: 10.0.0.206
```

```
Client IP address: 10.0.0.174
```

To find out the IP addresses of your machines, you can use the `ifconfig` function as such:

1. Open terminal
2. Type the following command

   ```
   # ifconfig
   ```



```
[root@localhost-live Documents]# ifconfig
enp0s3: flags=4163<UP,BROADCAST,RUNNING,MULTICAST>  mtu 1500
        inet 192.168.1.124  netmask 255.255.255.0  broadcast 192.168.1.255
        inet6 fe80::b383:7d80:7c6d:ab6e  prefixlen 64  scopeid 0x20<link>
        ether 08:00:27:e3:dd:46  txqueuelen 1000  (Ethernet)
        RX packets 130377  bytes 20611410 (19.6 MiB)
        RX errors 0  dropped 948  overruns 0  frame 0
        TX packets 7084  bytes 787879 (769.4 KiB)
        TX errors 0  dropped 0 overruns 0  carrier 0  collisions 0
```

For this device, the IPv4 address is `192.168.1.124` . You will need to figure out the IP addresses of your server and client if you are interested in running the code.

3.2 Compilation and execution of Craig Rowland's implementation

<u>On both machines:</u>

1. Download the code and save it.

2. Open terminal within where you saved the code.
3. Log in as root

   ```
   # sudo su
   ```

   Then, provide the administrator's password to authenticate.
4. Compile the code

   ```
   # cc -o covert covert_tcp.c
   ```

   You may see warnings after executing the above line, it can be ignored for this assignment.
5. Assign one machine to be the server of your choice, and the other to be the client.

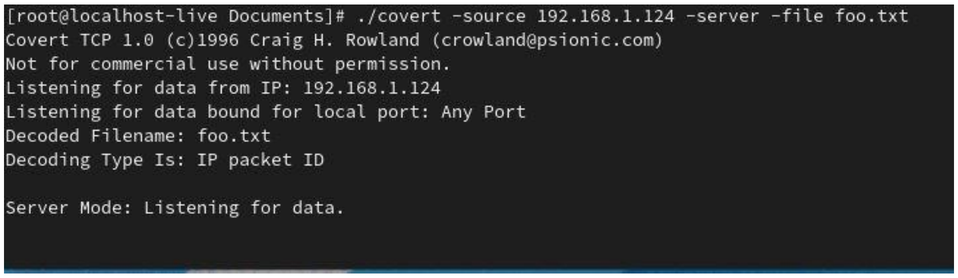
<u>On Server machine:</u>

1. Run the executables

   ```
   # ./covert -source <client's IP> -server -file foo.txt
   ```

   Replace `<client's IP>` with client's actual IP address without the `< >`.

   Now, the machine should be listening for data.



```
[root@localhost-live Documents]# ./covert -source 192.168.1.124 -server -file foo.txt
Covert TCP 1.0 (c)1996 Craig H. Rowland (crowland@psionic.com)
Not for commercial use without permission.
Listening for data from IP: 192.168.1.124
Listening for data bound for local port: Any Port
Decoded Filename: foo.txt
Decoding Type Is: IP packet ID

Server Mode: Listening for data.
```

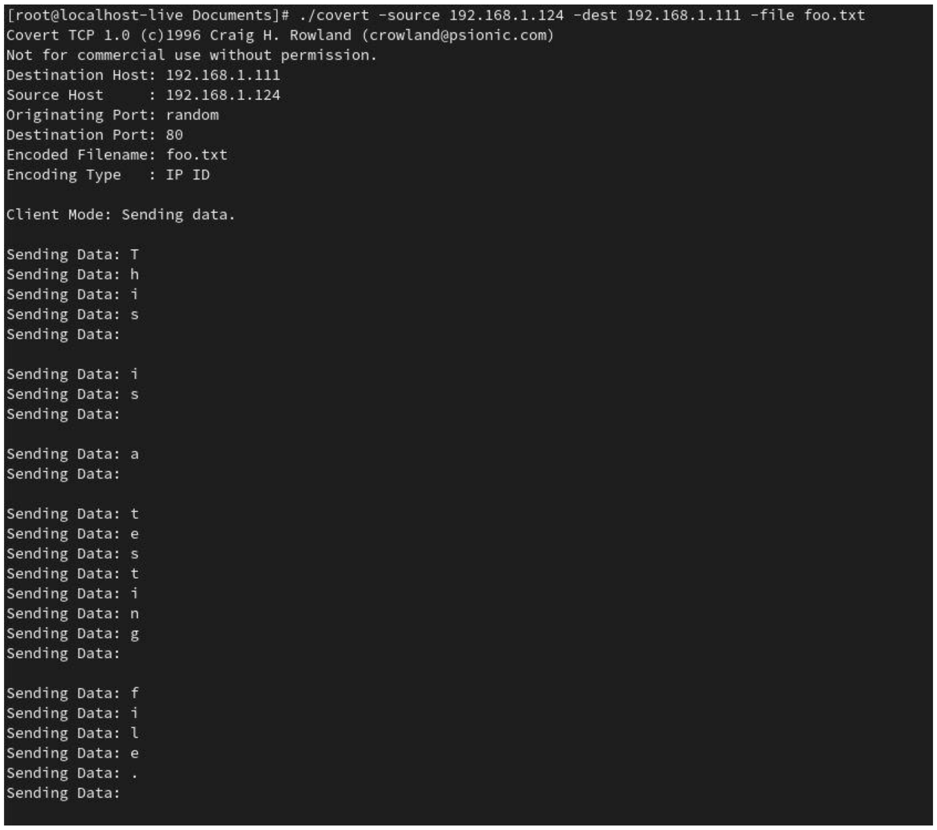
<u>On Client machine:</u>

1. Create a file to be transferred, put in some texts and saved it as "foo.txt"

2. Run the executables

   ```
   # ./covert -source <client’s IP> -dest <server’s IP> -file foo.txt
   ```

   Replace `<client’s IP>` and `<server’s IP>` with their actual IP addresses without the < >.



```
[root@localhost-live Documents]# ./covert -source 192.168.1.124 -dest 192.168.1.111 -file foo.txt
Covert TCP 1.0 (c)1996 Craig H. Rowland (crowland@psionic.com)
Not for commercial use without permission.
Destination Host: 192.168.1.111
Source Host     : 192.168.1.124
Originating Port: random
Destination Port: 80
Encoded Filename: foo.txt
Encoding Type   : IP ID

Client Mode: Sending data.

Sending Data: T
Sending Data: h
Sending Data: i
Sending Data: s
Sending Data:

Sending Data: i
Sending Data: s
Sending Data:

Sending Data: a
Sending Data:

Sending Data: t
Sending Data: e
Sending Data: s
Sending Data: t
Sending Data: i
Sending Data: n
Sending Data: g
Sending Data:

Sending Data: f
Sending Data: i
Sending Data: l
Sending Data: e
Sending Data: .
Sending Data:
```

3.3 Compilation and execution of our implementation

<u>On both machines:</u>

1. Download the code and save it.

2. Open terminal within where you saved the code.

3. Log in as root

   ```
   # sudo su
   ```

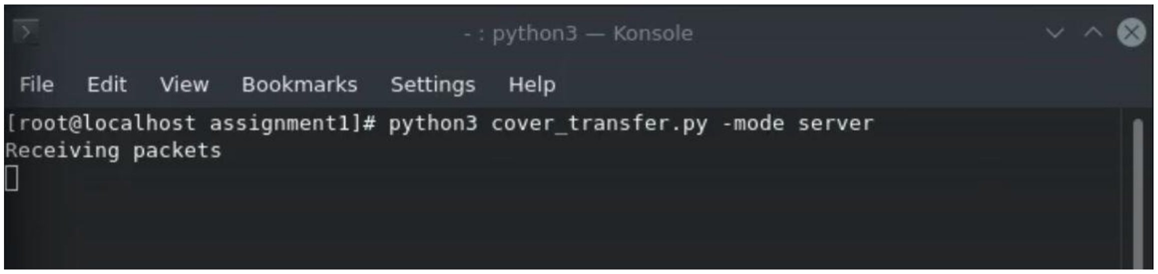
   Then, provide the administrator’s password to authenticate.

4. Assign one machine to be the server of your choice, and the other to be the client.

<u>On server machine:</u>

1. Execute the code

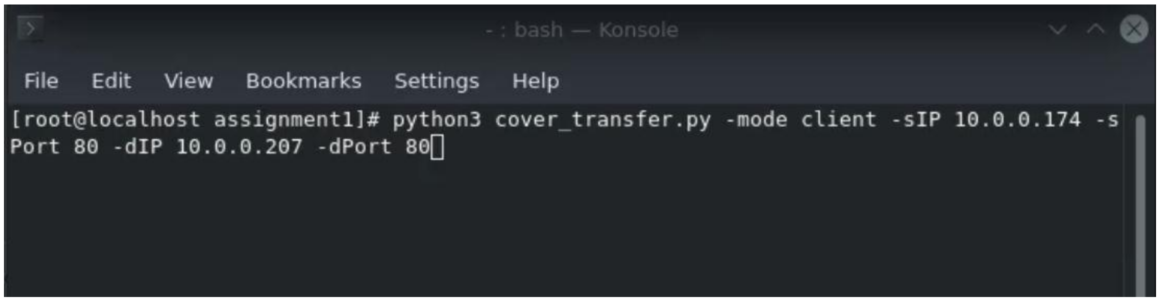
   ```
   # python3 covert_channel -mode server
   ```



```
[root@localhost assignment1]# python3 cover_transfer.py -mode server
Receiving packets
```

<u>On client machine:</u>

1.
   ```
   # python3 covert_channel -mode client -sIP
   10.0.0.174 -sPort 80 -dIP 10.0.0.207 -dPort 80
   ```



```
[root@localhost assignment1]# python3 cover_transfer.py -mode client -sIP 10.0.0.174 -s
Port 80 -dIP 10.0.0.207 -dPort 80
```

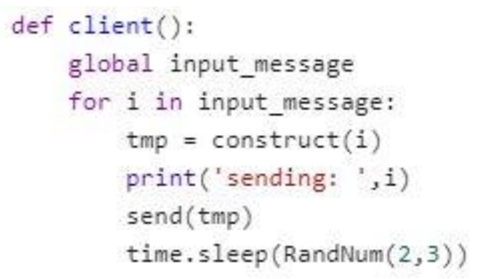
4. Critiques of Rowland’s implementation

4.1 Poorly designed architecture

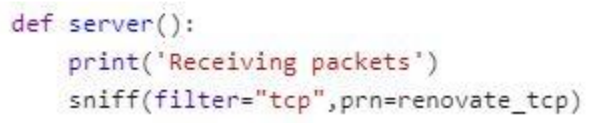
In Rowland’s implementation, we can see that forgepacket(…) function spams from line 184 to line 452. This is a super poorly structured function that can be broken down into smaller functions. For example, both client mode and server mode are written inside the forgepacket(…) function, when in fact, there could be a function for client mode and a function for server mode. By separating them and avoiding having long functions in our code, it increases the readability of our codes.

4.2 Rectification

To rectify this problem, we have separated the client mode and the server mode as such.

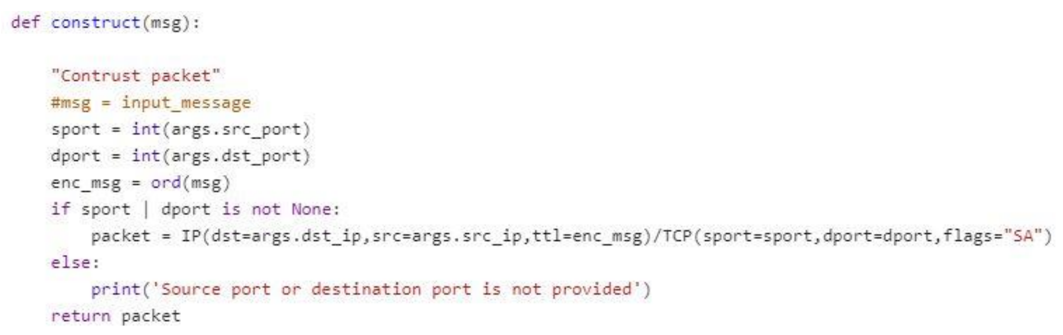


```
def client():
    global input_message
    for i in input_message:
        tmp = construct(i)
        print('sending: ',i)
        send(tmp)
        time.sleep(RandNum(2,3))
```



```
def server():
    print('Receiving packets')
    sniff(filter="tcp",prn=renovate_tcp)
```

Although Python has simplified the implementation a lot, we still separate it from the main function for its readability and maintainability. Within these two functions, we can see that we called a couple of other functions, such as `construct()` and `renovate_tcp()`. This rectifies the problem we found in Rowland's implementation.

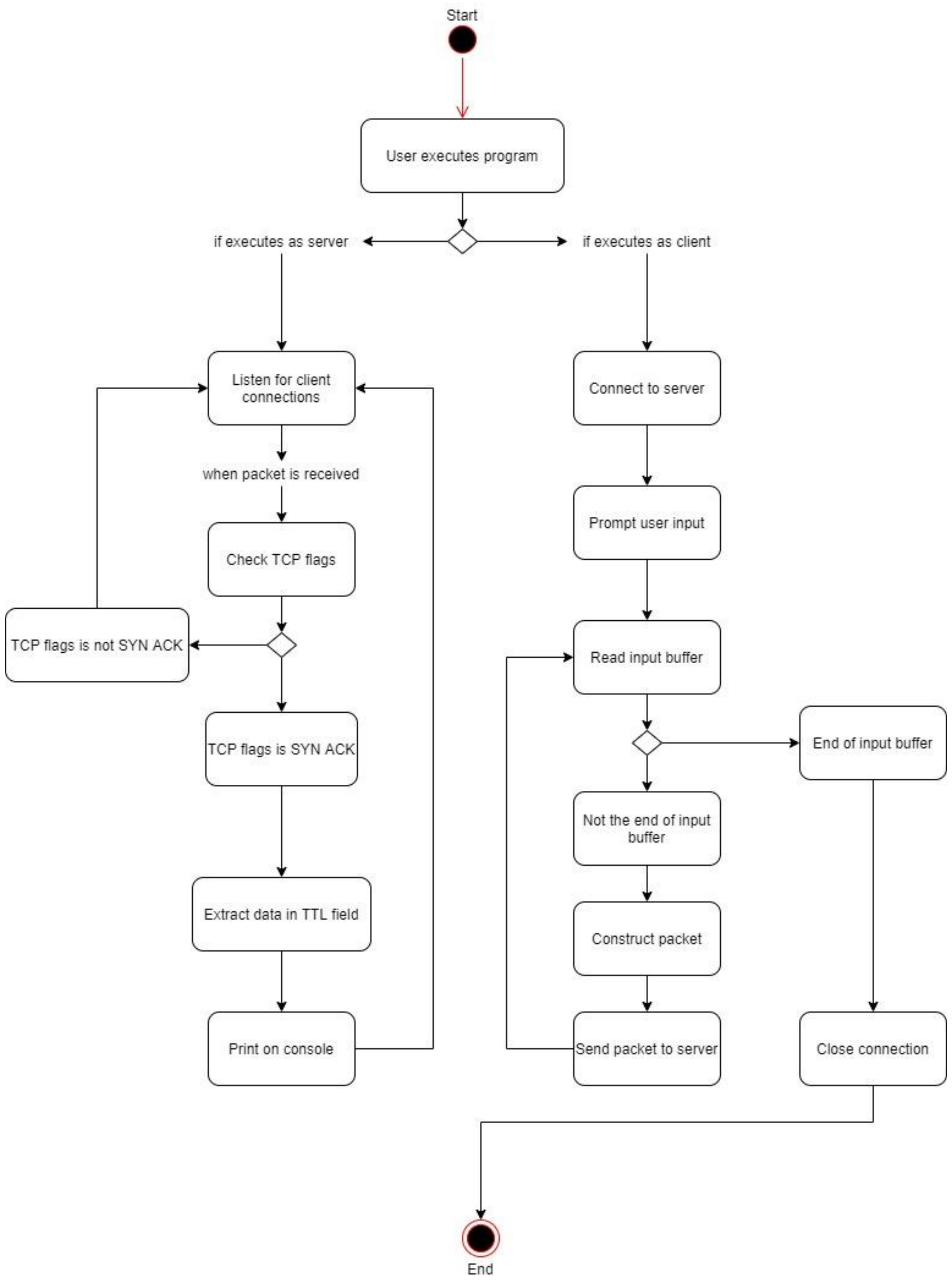


```
def construct(msg):

    "Contrust packet"
    #msg = input_message
    sport = int(args.src_port)
    dport = int(args.dst_port)
    enc_msg = ord(msg)
    if sport | dport is not None:
        packet = IP(dst=args.dst_ip,src=args.src_ip,ttl=enc_msg)/TCP(sport=sport,dport=dport,flags="SA")
    else:
        print('Source port or destination port is not provided')
    return packet
```

5. Design

5.1 Flowchart



5.2 Explanation

In our implementation, we have chosen to use Python as our programming language. Mainly because in Python, we can utilize the Scapy library. Scapy can manage packet traffics, "forge or decode packets, send them on the wire, capture them, and match requests and replies" (https://en.wikipedia.org/wiki/Scapy).
Moreover, we have chosen to use the TCP flags as the filter to distinguish between covert messages and regular messages. If the TCP flags is set as SYN ACK, our program will filter out this packet and examine its IP header.
In the IP header, we have chosen to use the TTL field to embed our data within. The reason for that is because we can view the TTL field easily in Wireshark, so that we know if our covert channel is transmitting data as we intended.
Of course, the choice of using SYN ACK as the filter and TTL field as the embedment are up to the developers, and we chose these for the simplicity in debugging process.

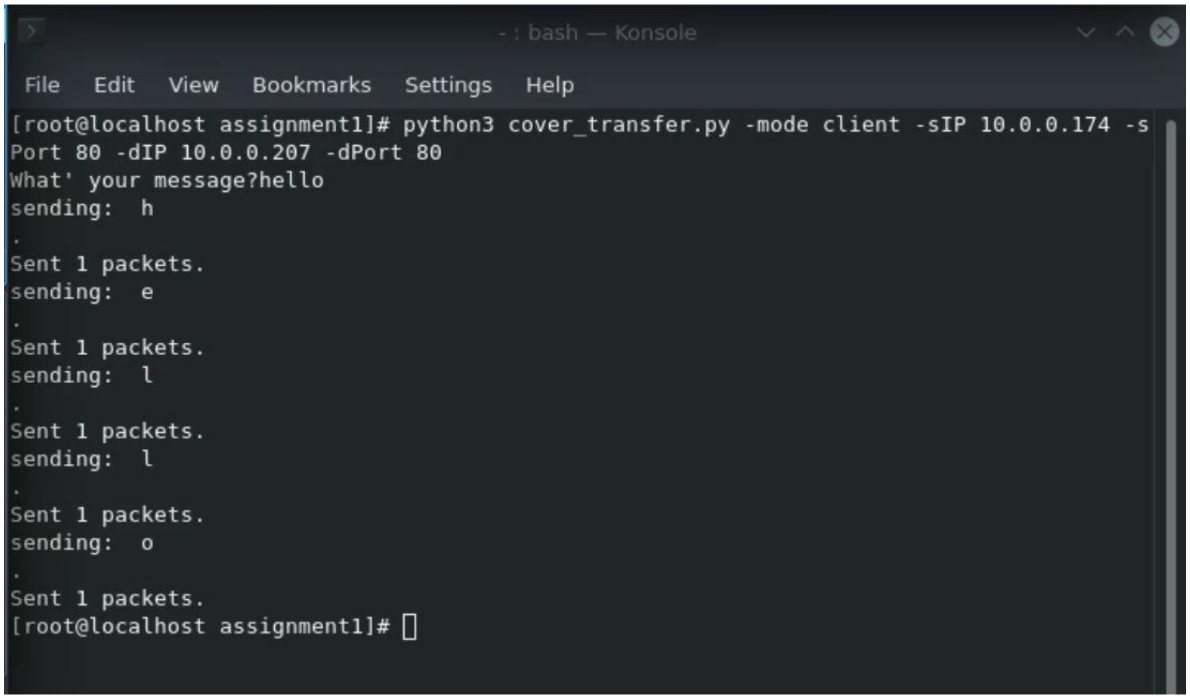
6. Testing

6.1 Test cases

Rule #	Test Description	Tool used	Expected Results	Pass/Fail
1	To see if the server can receive data embedded in the TTL field from each packet, and forge them correctly.	Python3	Client machine should successfully send all of the user inputs. Server machine should receive all data sent by client and forge them back accordingly.	Pass

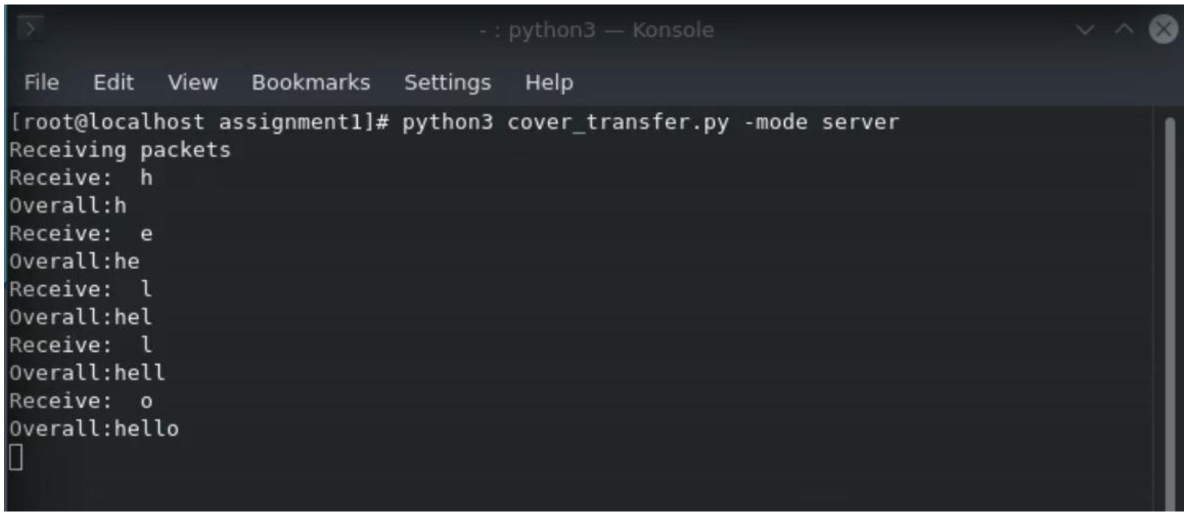
6.2 Supporting Evidences

Below are screenshots from our demo video. Full video can be found in the submission folder. Wireshark captures are also included in the submission.



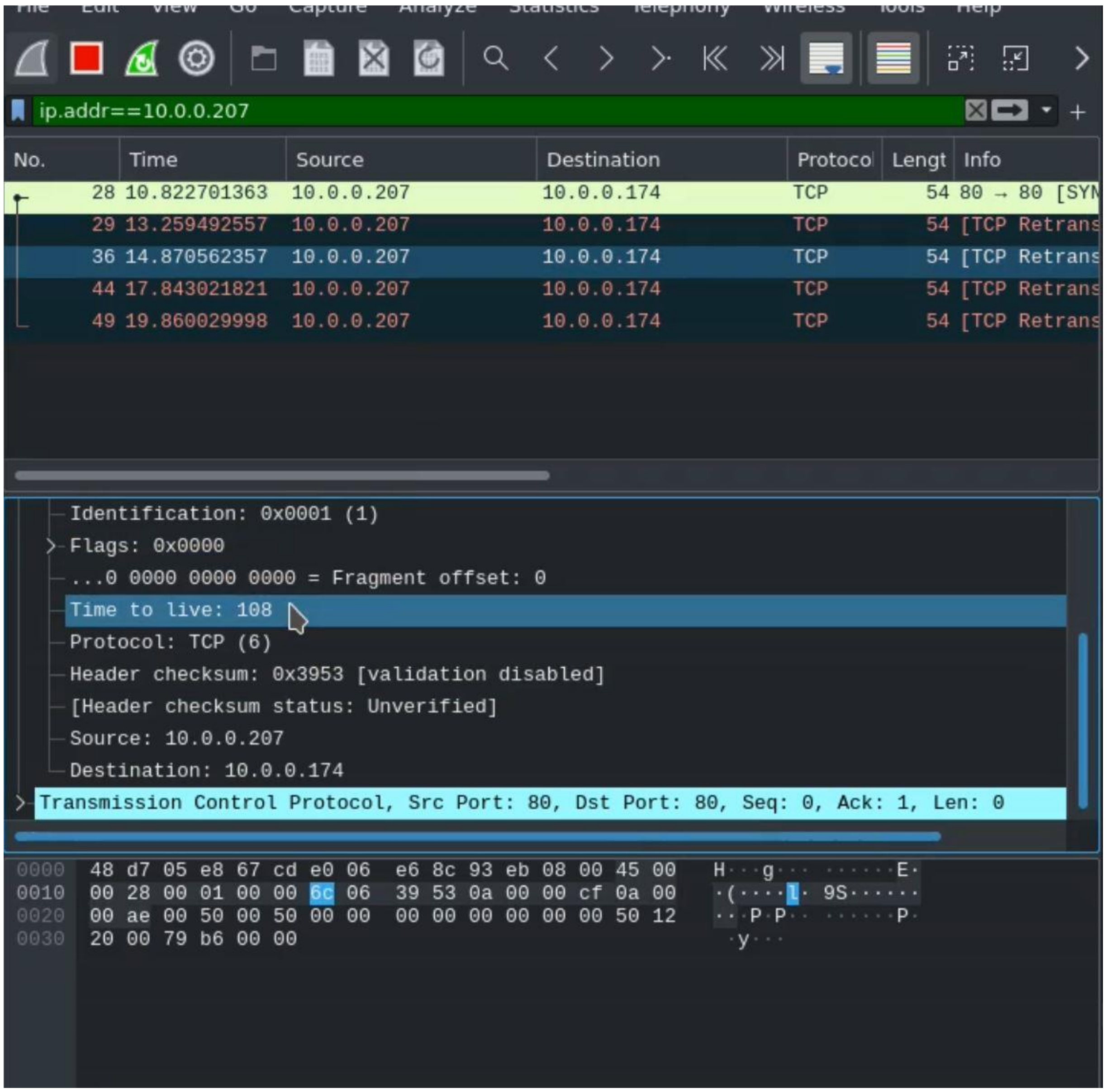
```
[root@localhost assignment1]# python3 cover_transfer.py -mode client -sIP 10.0.0.174 -s
Port 80 -dIP 10.0.0.207 -dPort 80
What' your message?hello
sending:  h
.
Sent 1 packets.
sending:  e
.
Sent 1 packets.
sending:  l
.
Sent 1 packets.
sending:  l
.
Sent 1 packets.
sending:  o
.
Sent 1 packets.
[root@localhost assignment1]# 
```

Above is the client machine sending the data to the server, the input buffer "hello" has been successfully read and sent.



```
[root@localhost assignment1]# python3 cover_transfer.py -mode server
Receiving packets
Receive:  h
Overall:h
Receive:  e
Overall:he
Receive:  l
Overall:hel
Receive:  l
Overall:hell
Receive:  o
Overall:hello
```

Above is the server machine receiving the packets sent by the client, and successfully forged the message back accordingly.


File Edit View Go Capture Analyze Statistics Telephony Wireless Tools Help

```
ip.addr==10.0.0.207

No.  Time          Source      Destination  Protocol  Length  Info
  28 10.822701363  10.0.0.207  10.0.0.174   TCP       54      80 → 80 [SYN
  29 13.259492557  10.0.0.207  10.0.0.174   TCP       54      [TCP Retrans
  36 14.870562357  10.0.0.207  10.0.0.174   TCP       54      [TCP Retrans
  44 17.843021821  10.0.0.207  10.0.0.174   TCP       54      [TCP Retrans
  49 19.860029998  10.0.0.207  10.0.0.174   TCP       54      [TCP Retrans

    Identification: 0x0001 (1)
  > Flags: 0x0000
    ...0 0000 0000 0000 = Fragment offset: 0
    Time to live: 108
    Protocol: TCP (6)
    Header checksum: 0x3953 [validation disabled]
    [Header checksum status: Unverified]
    Source: 10.0.0.207
    Destination: 10.0.0.174
> Transmission Control Protocol, Src Port: 80, Dst Port: 80, Seq: 0, Ack: 1, Len: 0

0000  48 d7 05 e8 67 cd e0 06  e6 8c 93 eb 08 00 45 00   H···g··· ······E·
0010  00 28 00 01 00 00 6c 06  39 53 0a 00 00 cf 0a 00   ·(····l· 9S······
0020  00 ae 00 50 00 50 00 00  00 00 00 00 00 00 50 12   ···P·P·· ······P·
0030  20 00 79 b6 00 00                                   ·y···
```

Above is the Wireshark capture on the server machine. As you can see, the TTL field is 108, which is the letter “l” in ASCII.

7. Conclusion

In conclusion, we think that the covert channel is not too challenging to implement, especially with the help from Scapy. Our implementation successfully transmits the data via the TTL field in the IP header, and the server is able to decode this and forge the original message back. The use of the SYN ACK flag as the filter can be changed as desired by the developers easily. Also, we have identified the problem in Craig Rowland's "covert_tcp.c", which is the function being overly complex, and we have rectified it by breaking up a big function into smaller functions.