Systems Security Report
Design and Implementation of Network Covert Channels

Alan Leong Kah Wah
Cyber Security Agency of Singapore

Jefnilham Bin Jamaludin
Cyber Security Agency of Singapore

Matthew Cheong Kin Weng
Cyber Security Agency of Singapore

Lee Ying Rong
Cyber Security Agency of Singapore

Ho Yoke Yin
Cyber Security Agency of Singapore

Abstract

Network covert channels have typically been a way for hidden information to be communicated by exploiting vulnerabilities in current network protocol. By researching on covert channels, we would gain deeper insight on developing and implementing covert channels that would in turn help to create a stronger network infrastructure against such attacks.

The aim of our project is to present a Proof of Concept by designing and implementing network covert channels using Transmission Control Protocol (TCP) with proposed data hiding scenario(s), to ensure the non-detectability of covert channels. In our paper, we would discuss potential vulnerabilities in the current network infrastructure that could be exploited to establish the communication channel and showcase the functionality of our proposed covert channel model.

1 Introduction

1.1 Background

Unconventional methods of communication outside of the system's protocol were coined by Butler W.Lampson to be called Covert Channels.[1] As these types of communications lie outside what is to be expected they are stealthier and more secure compared to mainstream communication channels.

There are two broad classification of covert channels: storage channels and timing channels. Communication via storage channel requires the sender to store hidden data within some storage location, subsequently the hidden data would be retrieved by the receiver. Alternatively, communication via timing channels require the sender to signal information by means of controlling observable computational readings that could be translated to information for the receiver.[2]

The project will focus on developing a covert channel using the TCP network.

Noteworthy research on covert channel done over TCP/IP network would be Covert Channel Analysis and Data Hiding in TCP/IP by Kamran Ahsan.[3]

The work focused on developing and analysing the various methods that could be used in establishing covert channels using TCP/IP protocol. They covered some of the flaws and improvements that would be critical for improvement in network security infrastructure. In addition, they have created some novelty ways of covert channel using techniques such as packet sorting and packet header manipulation.

Our paper aims to create alternatives to the methods provided and provide analysis on the effectiveness of the covet channel.

1.2 Objective

The purpose of this paper is to deepen the understanding of the existing TCP/IP protocol by explaining in detail the various critical aspects of the protocol that could be in turn used in developing our communication channel. We would make use of the new insight to create a covert channel built upon the TCP protocol.

We will do this specifically with storage channels within TCP, with out of order packets. If done correctly, we would have "innocent" network traffic which would be able to command an already infected machine with a payload.

1.3 Motivation

The motivation for this paper is to deepen our understanding in application and development of various covert channels. The deepened knowledge would expose possible limitations and restrictions which could be further studied and used in defending against malicious covert communication.

2 Internet Protocol and Transmission Control Protocol

2.1 IPv4

The first internet protocol standard was published in RFC 791.[4] The internet protocol is responsible for encapsulating TCP segments and contains the IP address that would be used to identify the sender and the receiver for these particular packets. In a network setting, routers would decapsulate packets to get the IP address of the packets. Depending on the routing table, the packets could be dropped or encapsulated to be sent to the next network medium.

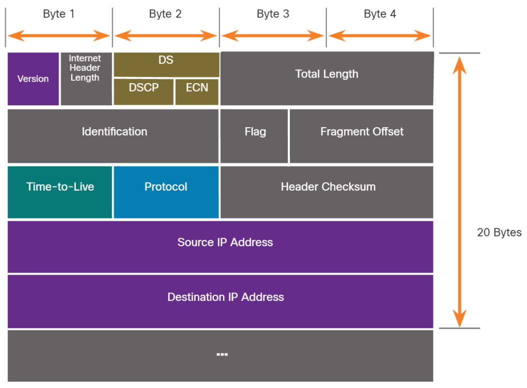


Figure 1: IPv4 header

2.2 TCP

TCP is the transport layer which works in complement with the IP protocol. The TCP provides reliability to communication by providing reliability to the packets transferred between applications across the network.[5] The application of TCP guarantees that the data transferred are error-less and in-sequence. Details of how this is achieved would be covered in the subsequent sub-chapters.

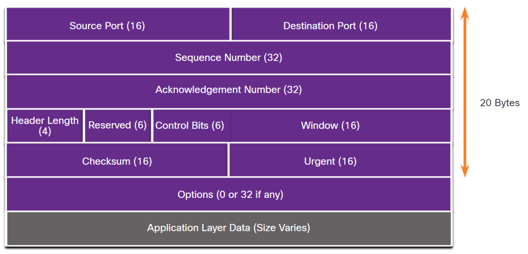


Figure 2: TCP header

2.3 TCP Reliability – Guaranteed and Ordered Delivery

2.3.1 Out-of-Order Delivery

In standard data transferred via TCP we could observe a common observation of "out-of -order". This phenomenon happens when during the transmission of a particular sequence of packets, the order of arrival is different from the order which it was sent out in. We observe that this particular observation is much more common for communication over the internet.

During data transfer, the data stream would be broken down into smaller packets. The packets would be sent off in sequence via routes that would be deemed most effective by each interim network medium to reach their destination. Upon receiving the packets, the end point would reassemble the packets back in the sequence that the data was supposed to be received in, completing the transfer of data. By the nature of TCP, we would be expecting data packets to travel in different routes resulting in packet loss and packet delays.

Hence, using this particular phenomenon as the spine of our covert channel development we would force packets to be out of order on purpose to signal a message via the sequence the packets arrive in. Other studies on subjects such as "Receive Index" and "Re-order judging Algorithm" will deepen our understanding on the concept of out-of-order delivery that would help with the development of the covert channel.

2.3.2 Receive Index (RI)

A received index is a value allocated to packets upon reaching their destination with regards to their sequence of arrival. The duplicated packets are not allocated received index and lost packets would result in received index not being assigned. If reordering is not needed, sequence number on arriving packets would be the same as the receive index allocated.

Generally, the receive index could be used to compute the time of an arriving packet, be it early or delayed. The following examples display the usage of receive index common settings for a set of 6 packets.

```
Example 1:
Arrived sequence: 2 1 4 5 3 6
Receive index: 1 2 3 4 5 6

Example 2:
Arrived sequence: 1 4 3 5 3 5
Receive index: 1 3 4 5 - -
```

In the first example, there is no loss or duplication in packets, the packets are assigned a receive index as to how they arrive. However, in the second example, the packet with sequence number 2 is lost in transmission; hence '2' is not assigned as an RI. Since Packet 3 and 5 are duplicates, the second copy arrived would not have an assigned RI.

The sequence of packets would be considered out-of-order if the receive index does not tally with the sequence number of the packets received. Duplicates, for which an RI is not defined, would be disregarded.[6]

2.3.3 Reorder-judging Algorithm

Generally, packets are considered out-of-order if the sequence number is smaller than the packet that was previously received by the receiver. However, it is fully possible for network duplication, network reordering or retransmission to cause the packet to be considered out-of-order. Although the mentioned 3 causes are different in nature, for simplicity the only cause we would discuss would be network reordering.

Network reordering is normally a result of 2 factors: parallelism within the router or route change within the network.[7] Figure 3 displays the reorder-judging algorithm, executed at the receiving end-point, which is based on TCP sequence number, IP ID and time lag.

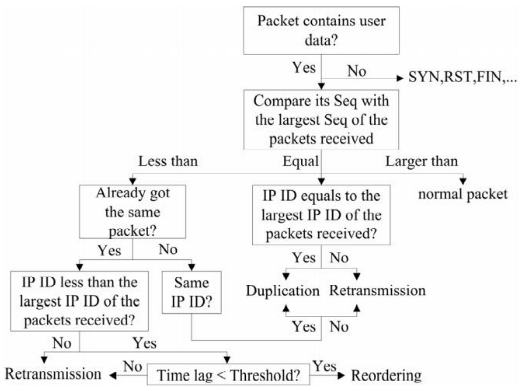

Packet contains user data?
Yes
No
SYN,RST,FIN,...
Compare its Seq with the largest Seq of the packets received
Less than
Equal
Larger than
normal packet
Already got the same packet?
Yes
No
IP ID equals to the largest IP ID of the packets received?
Yes
No
Duplication
Retransmission
IP ID less than the largest IP ID of the packets received?
Same IP ID?
Yes
No
No
Yes
Retransmission
No
Time lag < Threshold?
Yes
Reordering

Figure 3: Process of the reorder-judging algorithm

Since TCP IP ID resets to 0 when incrementing to 65535, depending on the various states of different routing paths, we would be able to expect a different arrangement of IPID. Due to the fast retransmission algorithm, the time delay of a retransmitted packet must be much larger than a reordered packet for the algorithm to work as intended.

Hence, it was set for a threshold of 300ms to the time lag to single out reordered packet from retransmitted packet with IPID wrapped.[7]

3 Design

3.1 Scenario

We will act as an Advanced Persistent Threat (APT) already possessing active connections to the victim's system. We will attempt communication with said victim while masquerading as legitimate traffic.

We are successful if we are able to have undetectable communication with a victim machine, while being able to perform actions on objectives.

3.2 Proposed Proof-of-Concept

We built a client-server application for the covert channel communication. To simulate the covert channel, a webserver is first set up for the client's entry.

TCP packets are sent when the client visits the website and performs various actions via Python, NFQueue and Scapy. NFQueue intercepts packets, shuffles, and delays them before transmitting the packets. Scapy allows the modification of packet fields.

If a packet is not needed for the covert channel, the packet would be ignored by the program. The receiver then reads each packet it receives, ideally looking out for signs that a packet has been modified by the sender, and piece together the hidden message.

3.3 Manipulation of TTL Field in IP Header

To assist in the interpretation of forced out-of-order TCP packets, we will be utilising the TTL field in the IP Header. We introduce a start and end identification to inform the server (receiver) that any packets in between could potentially be reordered.

With the help of the TTL field, the receiver can easily identify packets modified by the sender and interpret the covert message accordingly.

The TTL values utilised in the Proof-of-Concept are:

- Start and End: 50
- Packet that should come first: 63
- Packet that should come later: 65

3.4 Other settings/considerations

- Sender to specify preferred route – Fixed routing
- In the interest of time, we are setting up the receiver as an already compromised system

3.4.1 Why TTL? Why not reserved bits?

We wanted to make it hard to see any obvious outward tampering. Using reserved bits or any of the flag bits would immediately be noticeable by any IDS. Most IDS look at reserved bits because of common attacks like Christmas tree attacks. Additionally, tampering with the reserved bits has already been done before in a prior research paper and we chose TTL in an attempt to try something new.

3.4.2 Why TTL value of 50, 63 and 65?

There is no definite answer to why the TTL values were selected. However, our decision was to use a value that is closer/similar to the TTL of Linux systems, which is 64. As our main system used for testing was an Ubuntu machine, we decided on 63 and 65 as the TTL values for packets that would potentially be reordered.

3.4.3 Noise & wrong message received

Since we only have a local network proof of concept set up, it was hard to test for noise. That being said, our tolerance for noise will be low, and a single packet out of order or lost would break the message the receiver was meant to receive.

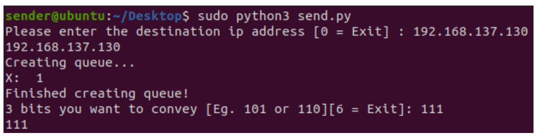
4 Proof of Concept

4.1 Covert Channels via Out-of-Order TCP Packets

We will now test the feasibility and detectability of our covert channel.

4.1.1 Sending the Covert Message

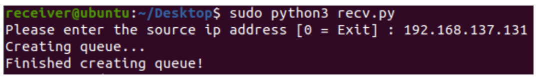
As previously mentioned, there is a client (sender) and a server (receiver). The sender uses a Python script (send.py) to create the NFQueue and message to send. The interface for the script is shown below.



```
sender@ubuntu:~/Desktop$ sudo python3 send.py
Please enter the destination ip address [0 = Exit] : 192.168.137.130
192.168.137.130
Creating queue...
X:  1
Finished creating queue!
3 bits you want to convey [Eg. 101 or 110][6 = Exit]: 111
111
```

Figure 4: Script interface for send.py

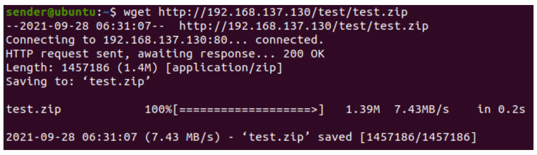
Here, we are going to send a message of "111" bits to the receiver at IP address 192.168.137.130. Before the communication takes place, we must set up the receiver script to actively listen for TCP packets originating from the sender.



```
receiver@ubuntu:~/Desktop$ sudo python3 recv.py
Please enter the source ip address [0 = Exit] : 192.168.137.131
Creating queue...
Finished creating queue!
```

Figure 5: Script interface for recv.py

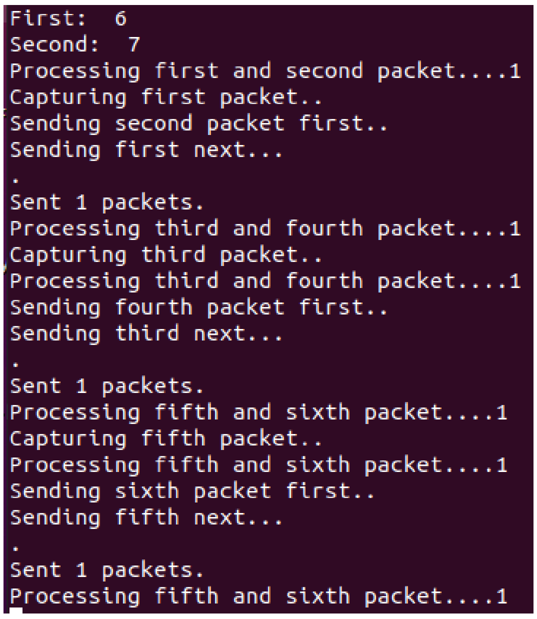
Now, we will create the communication channel with an action that involves the TCP protocol. We initiate a simple file download from the receiver using the wget command in a separate terminal, as seen below.



```
sender@ubuntu:~$ wget http://192.168.137.130/test/test.zip
--2021-09-28 06:31:07--  http://192.168.137.130/test/test.zip
Connecting to 192.168.137.130:80... connected.
HTTP request sent, awaiting response... 200 OK
Length: 1457186 (1.4M) [application/zip]
Saving to: ‘test.zip’

test.zip            100%[===================>]   1.39M  7.43MB/s    in 0.2s

2021-09-28 06:31:07 (7.43 MB/s) - ‘test.zip’ saved [1457186/1457186]
```

Figure 6: File download as legit TCP communication

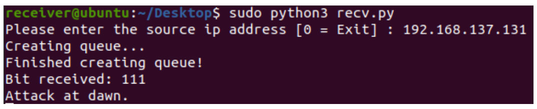
In the initial script terminal, we see the following output below that reorders the packets using NFQueue and Scapy. "1" signifies a reorder, "0" means no reorder.



```
First:  6
Second:  7
Processing first and second packet....1
Capturing first packet..
Sending second packet first..
Sending first next...
.
Sent 1 packets.
Processing third and fourth packet....1
Capturing third packet..
Processing third and fourth packet....1
Sending fourth packet first..
Sending third next...
.
Sent 1 packets.
Processing fifth and sixth packet....1
Capturing fifth packet..
Processing fifth and sixth packet....1
Sending sixth packet first..
Sending fifth next...
.
Sent 1 packets.
Processing fifth and sixth packet....1
```

Figure 7: Output for packet reordering

Once the communication is complete, the receiver's script is able to read the incoming packets and interpret the message accordingly to receive the bits of "111".



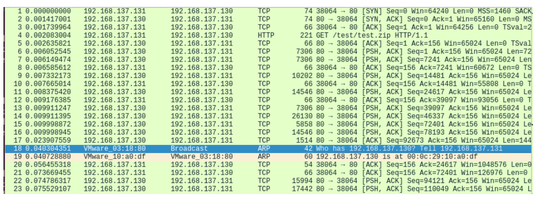
```
receiver@ubuntu:~/Desktop$ sudo python3 recv.py
Please enter the source ip address [0 = Exit] : 192.168.137.131
Creating queue...
Finished creating queue!
Bit received: 111
Attack at dawn.
```

Figure 8: Message corresponding to bit order received

In this scenario, the bits of "111" signifies the next course of action which is "Attack at dawn".

4.1.2 Detection of the Covert Channel

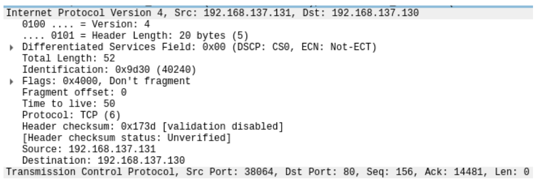
Using Wireshark to perform live network traffic capture, we were able to log down all the packets involved in the communication for the covert channel. Below is a screenshot of the network traffic at a glance. It does not explicitly indicate that there are any suspicious/malformed packets logged.



```
  1 0.000000000   192.168.137.131     192.168.137.130     TCP        74 38064 → 80 [SYN] Seq=0 Win=64240 Len=0 MSS=1460 SACK
  2 0.001417001   192.168.137.130     192.168.137.131     TCP        74 80 → 38064 [SYN, ACK] Seq=0 Ack=1 Win=65160 Len=0 MS
  3 0.001739964   192.168.137.131     192.168.137.130     TCP        66 38064 → 80 [ACK] Seq=1 Ack=1 Win=64256 Len=0 TSval=2
  4 0.002083004   192.168.137.131     192.168.137.130     HTTP      221 GET /test/test.zip HTTP/1.1
  5 0.002635821   192.168.137.130     192.168.137.131     TCP        66 80 → 38064 [ACK] Seq=1 Ack=156 Win=65024 Len=0 TSval
  6 0.006052545   192.168.137.130     192.168.137.131     TCP      7306 80 → 38064 [PSH, ACK] Seq=1 Ack=156 Win=65024 Len=72
  7 0.006149474   192.168.137.130     192.168.137.131     TCP      7306 80 → 38064 [PSH, ACK] Seq=7241 Ack=156 Win=65024 Len
  8 0.006585612   192.168.137.131     192.168.137.130     TCP        66 38064 → 80 [ACK] Seq=156 Ack=7241 Win=60672 Len=0 TS
  9 0.007332173   192.168.137.130     192.168.137.131     TCP     10202 80 → 38064 [PSH, ACK] Seq=14481 Ack=156 Win=65024 Le
 10 0.007665014   192.168.137.131     192.168.137.130     TCP        66 38064 → 80 [ACK] Seq=156 Ack=14481 Win=55808 Len=0 T
 11 0.008375420   192.168.137.130     192.168.137.131     TCP     14546 80 → 38064 [PSH, ACK] Seq=24617 Ack=156 Win=65024 Le
 12 0.009176385   192.168.137.131     192.168.137.130     TCP        66 38064 → 80 [ACK] Seq=156 Ack=39097 Win=93056 Len=0 T
 13 0.009911247   192.168.137.130     192.168.137.131     TCP      7306 80 → 38064 [PSH, ACK] Seq=39097 Ack=156 Win=65024 Le
 14 0.009911395   192.168.137.130     192.168.137.131     TCP     26130 80 → 38064 [PSH, ACK] Seq=46337 Ack=156 Win=65024 Le
 15 0.009998872   192.168.137.130     192.168.137.131     TCP      5858 80 → 38064 [PSH, ACK] Seq=72401 Ack=156 Win=65024 Le
 16 0.009998945   192.168.137.130     192.168.137.131     TCP     14546 80 → 38064 [PSH, ACK] Seq=78193 Ack=156 Win=65024 Le
 17 0.023907559   192.168.137.130     192.168.137.131     TCP      1514 80 → 38064 [ACK] Seq=92673 Ack=156 Win=65024 Len=144
 18 0.040304351   VMware_03:18:80     Broadcast           ARP        42 Who has 192.168.137.130? Tell 192.168.137.131
 19 0.040728880   VMware_10:a0:df     VMware_03:18:80     ARP        60 192.168.137.130 is at 00:0c:29:10:a0:df
 20 0.056455318   192.168.137.131     192.168.137.130     TCP        54 38064 → 80 [ACK] Seq=156 Ack=24617 Win=1048576 Len=0
 21 0.073669455   192.168.137.131     192.168.137.130     TCP        66 38064 → 80 [ACK] Seq=156 Ack=72401 Win=126976 Len=0
 22 0.074786317   192.168.137.130     192.168.137.131     TCP     15994 80 → 38064 [PSH, ACK] Seq=94121 Ack=156 Win=65024 Le
 23 0.075529107   192.168.137.130     192.168.137.131     TCP     17442 80 → 38064 [PSH, ACK] Seq=110049 Ack=156 Win=65024 L
```

Figure 9: Wireshark capture of network traffic

Diving deeper into the pcap analysis, we are able to discover the packets modified by our Python script (send.py).

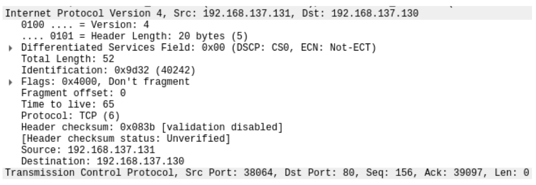


```
Internet Protocol Version 4, Src: 192.168.137.131, Dst: 192.168.137.130
    0100 .... = Version: 4
    .... 0101 = Header Length: 20 bytes (5)
  ▸ Differentiated Services Field: 0x00 (DSCP: CS0, ECN: Not-ECT)
    Total Length: 52
    Identification: 0x9d30 (40240)
  ▸ Flags: 0x4000, Don't fragment
    Fragment offset: 0
    Time to live: 50
    Protocol: TCP (6)
    Header checksum: 0x173d [validation disabled]
    [Header checksum status: Unverified]
    Source: 192.168.137.131
    Destination: 192.168.137.130
Transmission Control Protocol, Src Port: 38064, Dst Port: 80, Seq: 156, Ack: 14481, Len: 0
```

Figure 10: Analysing packets modified by send.py: First modified packet

The above screenshot is the first packet modified by our script. The TTL field is set to 50 to inform the receiver that this is the starting point for any potential packet reorder. Next, we search for TTL values of 63 (packet that should come first) and 65 (packet that should come after the packet with a TTL value of 63).

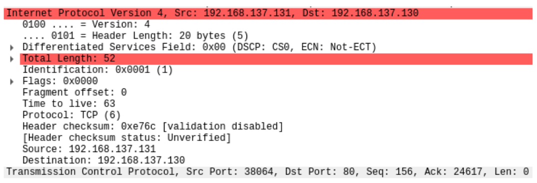
Below, we have our next modified packet with a TTL value of 65, meaning that the next modified packet in the list should contain a TTL value of 63, and an acknowledgement number that is smaller than the one seen.



```
Internet Protocol Version 4, Src: 192.168.137.131, Dst: 192.168.137.130
    0100 .... = Version: 4
    .... 0101 = Header Length: 20 bytes (5)
  ▸ Differentiated Services Field: 0x00 (DSCP: CS0, ECN: Not-ECT)
    Total Length: 52
    Identification: 0x9d32 (40242)
  ▸ Flags: 0x4000, Don't fragment
    Fragment offset: 0
    Time to live: 65
    Protocol: TCP (6)
    Header checksum: 0x083b [validation disabled]
    [Header checksum status: Unverified]
    Source: 192.168.137.131
    Destination: 192.168.137.130
Transmission Control Protocol, Src Port: 38064, Dst Port: 80, Seq: 156, Ack: 39097, Len: 0
```

Figure 11: Next (second) packet modified by send.py

The next modified packet's information can be seen below.



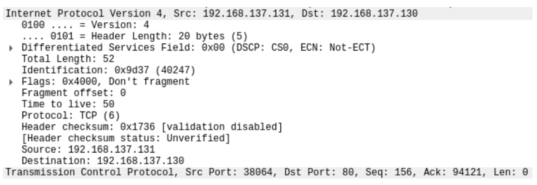
```
Internet Protocol Version 4, Src: 192.168.137.131, Dst: 192.168.137.130
    0100 .... = Version: 4
    .... 0101 = Header Length: 20 bytes (5)
  ▸ Differentiated Services Field: 0x00 (DSCP: CS0, ECN: Not-ECT)
  ▸ Total Length: 52
    Identification: 0x0001 (1)
  ▸ Flags: 0x0000
    Fragment offset: 0
    Time to live: 63
    Protocol: TCP (6)
    Header checksum: 0xe76c [validation disabled]
    [Header checksum status: Unverified]
    Source: 192.168.137.131
    Destination: 192.168.137.130
Transmission Control Protocol, Src Port: 38064, Dst Port: 80, Seq: 156, Ack: 24617, Len: 0
```

Figure 12: Next (third) packet modified by send.py

As mentioned previously, we see a TTL value of 63, and an acknowledgement number of 24617 which is smaller than the acknowledgement number of 39097 of the previously modified packet. This means that the packets were forcefully reordered.

Although the packet information has its Total Length field highlighted in red, it is not immediately obvious to the analyst when reviewing logs at a glance.

Once all modified packets with TTL values of 63 and 65 are done transmitting, we end off the communication to the receiver by sending the last modified packet with a TTL value of 50 (similar to the start).



```
Internet Protocol Version 4, Src: 192.168.137.131, Dst: 192.168.137.130
    0100 .... = Version: 4
    .... 0101 = Header Length: 20 bytes (5)
  ▸ Differentiated Services Field: 0x00 (DSCP: CS0, ECN: Not-ECT)
    Total Length: 52
    Identification: 0x9d37 (40247)
  ▸ Flags: 0x4000, Don't fragment
    Fragment offset: 0
    Time to live: 50
    Protocol: TCP (6)
    Header checksum: 0x1736 [validation disabled]
    [Header checksum status: Unverified]
    Source: 192.168.137.131
    Destination: 192.168.137.130
Transmission Control Protocol, Src Port: 38064, Dst Port: 80, Seq: 156, Ack: 94121, Len: 0
```

Figure 13: Last packet TTL = 50 modified by send.py

4.1.3 Summary/Learning Points

This demonstration successfully proves that forceful reordering of packets is not detectable and explicitly flagged by common network traffic analysis tools such as Wireshark.

Additionally, even if the communication channel was to be discovered, it is impossible to determine what information is being sent, given that the send.py Python script only reveals how the communication is executed and the possible encoded message sent. The decoding of the message can only be done on the receiver's side.

4.2 Covert Channels via Covert_TCP

Covert_TCP is another proof of concept for exploitation of covert channels using the TCP/IP protocol suite devised by Craig Rowland.[8]

The covert_TCP program manipulates the TCP/IP header of the data packets to send a file one byte at a time from any host to a destination. It can act like a server as well as a client and can be used to hide the data transmitted inside an IP header.

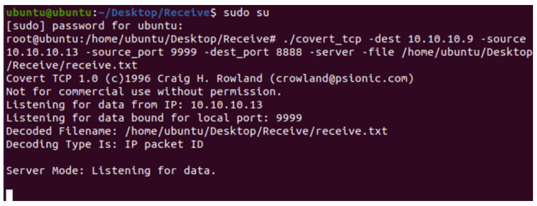
4.2.1 Create a covert channel to send a text document from a client (sender) to a server (receiver)

Both sender and receiver run the covert_tcp program, covert_tcp.c, that uses raw sockets to construct forged packets and encapsulate data from a filename given on the command line. The file itself can contain text or binary data as necessary. The message we are sending is “Secret Message”.

On the receiver with IP address 10.10.10.9, We run the program to start a listener to listen for packets from the sender using the following command:

```
./covert_tcp -dest 10.10.10.9 -source
    ↪ 10.10.10.13 -source_port 9999 -
    ↪ dest_port 8888 -server -file /
    ↪ home/ubuntu/Desktop/Receive/
    ↪ receive.txt
```

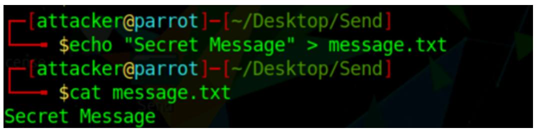
It will receive the data and save to the destination file given on the command line.



```
ubuntu@ubuntu:~/Desktop/Receive$ sudo su
[sudo] password for ubuntu:
root@ubuntu:/home/ubuntu/Desktop/Receive# ./covert_tcp -dest 10.10.10.9 -source
10.10.10.13 -source_port 9999 -dest_port 8888 -server -file /home/ubuntu/Desktop
/Receive/receive.txt
Covert TCP 1.0 (c)1996 Craig H. Rowland (crowland@psionic.com)
Not for commercial use without permission.
Listening for data from IP: 10.10.10.13
Listening for data bound for local port: 9999
Decoded Filename: /home/ubuntu/Desktop/Receive/receive.txt
Decoding Type Is: IP packet ID

Server Mode: Listening for data.
```

Figure 14: Start a listener

On the sender with IP address 10.10.10.13, create a message.txt file containing the string “Secret Message” at `/home/attacker/Desktop` ↪ `/Send` folder.



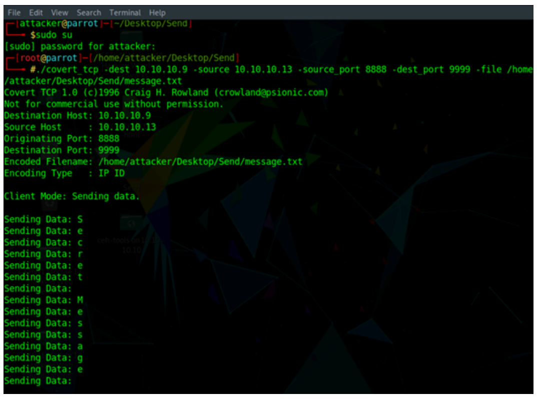
```
┌─[attacker@parrot]─[~/Desktop/Send]
└──╼ $echo "Secret Message" > message.txt
┌─[attacker@parrot]─[~/Desktop/Send]
└──╼ $cat message.txt
Secret Message
```

Figure 15: Create message.txt

Now, we will start sending the contents of message.txt file over tcp using the following command.

```
./covert_tcp -dest 10.10.10.9 -source
    ↪ 10.10.10.13 -source_port 8888 -
    ↪ dest_port 9999 -file /home/
    ↪ attacker/Desktop/Send/message.txt
```

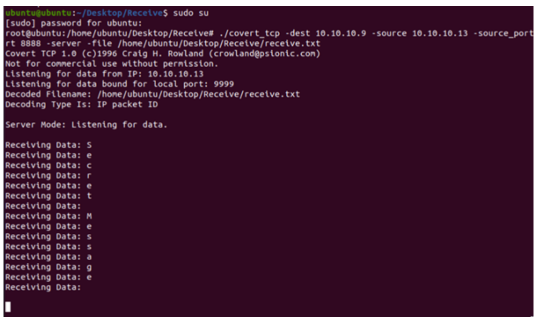
The covert_tcp program starts sending the string one character at a time, as shown below.



```
┌─[attacker@parrot]─[~/Desktop/Send]
└──╼ $sudo su
[sudo] password for attacker:
┌─[root@parrot]─[/home/attacker/Desktop/Send]
└──╼ #./covert_tcp -dest 10.10.10.9 -source 10.10.10.13 -source_port 8888 -dest_port 9999 -file /home
/attacker/Desktop/Send/message.txt
Covert TCP 1.0 (c)1996 Craig H. Rowland (crowland@psionic.com)
Not for commercial use without permission.
Destination Host: 10.10.10.9
Source Host     : 10.10.10.13
Originating Port: 8888
Destination Port: 9999
Encoded Filename: /home/attacker/Desktop/Send/message.txt
Encoding Type   : IP ID

Client Mode: Sending data.

Sending Data: S
Sending Data: e
Sending Data: c
Sending Data: r
Sending Data: e
Sending Data: t
Sending Data:
Sending Data: M
Sending Data: e
Sending Data: s
Sending Data: s
Sending Data: a
Sending Data: g
Sending Data: e
Sending Data:
```

Figure 16: Send one character at a time

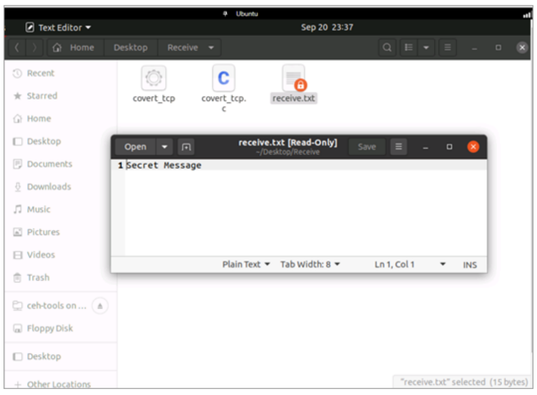
On the receiver, the message is being received, one character at a time.



```
ubuntu@ubuntu:~/Desktop/Receive$ sudo su
[sudo] password for ubuntu:
root@ubuntu:/home/ubuntu/Desktop/Receive# ./covert_tcp -dest 10.10.10.9 -source 10.10.10.13 -source_port
rt 8888 -server -file /home/ubuntu/Desktop/Receive/receive.txt
Covert TCP 1.0 (c)1996 Craig H. Rowland (crowland@psionic.com)
Not for commercial use without permission.
Listening for data from IP: 10.10.10.13
Listening for data bound for local port: 9999
Decoded Filename: /home/ubuntu/Desktop/Receive/receive.txt
Decoding Type Is: IP packet ID

Server Mode: Listening for data.

Receiving Data: S
Receiving Data: e
Receiving Data: c
Receiving Data: r
Receiving Data: e
Receiving Data: t
Receiving Data:
Receiving Data: M
Receiving Data: e
Receiving Data: s
Receiving Data: s
Receiving Data: a
Receiving Data: g
Receiving Data: e
Receiving Data:
```

Figure 17: Receive one character at a time

Navigate to `/home/ubuntu/Desktop/Receive` folder and double-click the receive.txt file to view its contents. It shows the full message saved in the file.



receive.txt [Read-Only]
~/Desktop/Receive

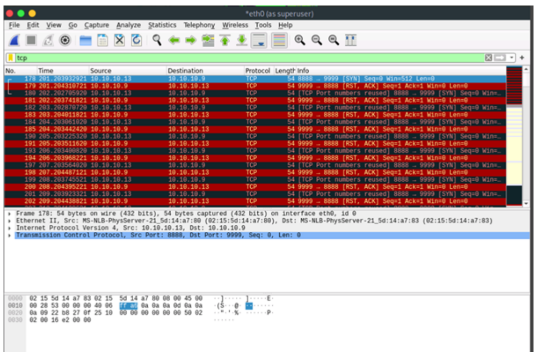
```
1 Secret Message
```

"receive.txt" selected (15 bytes)

Figure 18: receive.txt

4.2.2 Detection of the Covert Channel

Using Wireshark to perform packet capture, we logged all the packets involved in the communication for the Covert_TCP covert channel. Below is a screenshot of the network traffic with a display filter, “tcp”, showing many suspicious/-malformed packets logged.

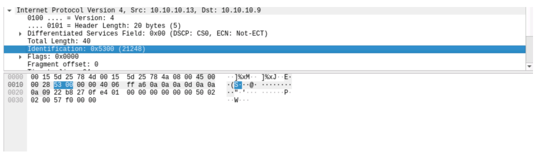


```
*eth0 (as superuser)
File  Edit  View  Go  Capture  Analyze  Statistics  Telephony  Wireless  Tools  Help
tcp
No.   Time           Source        Destination   Protocol Length Info
  178 201.203932921 10.10.10.13   10.10.10.9     TCP       54 8888 → 9999 [SYN] Seq=0 Win=512 Len=0
  179 201.204310721 10.10.10.9    10.10.10.13    TCP       54 9999 → 8888 [RST, ACK] Seq=1 Ack=1 Win=0 Len=0
  180 202.202705920 10.10.10.13   10.10.10.9     TCP       54 [TCP Port numbers reused] 8888 → 9999 [SYN] Seq=0 Win=…
  181 202.203741821 10.10.10.9    10.10.10.13    TCP       54 9999 → 8888 [RST, ACK] Seq=1 Ack=1 Win=0 Len=0
  182 203.202870720 10.10.10.13   10.10.10.9     TCP       54 [TCP Port numbers reused] 8888 → 9999 [SYN] Seq=0 Win=…
  183 203.204011821 10.10.10.9    10.10.10.13    TCP       54 9999 → 8888 [RST, ACK] Seq=1 Ack=1 Win=0 Len=0
  184 204.203061020 10.10.10.13   10.10.10.9     TCP       54 [TCP Port numbers reused] 8888 → 9999 [SYN] Seq=0 Win=…
  185 204.203442420 10.10.10.9    10.10.10.13    TCP       54 9999 → 8888 [RST, ACK] Seq=1 Ack=1 Win=0 Len=0
  190 205.203225320 10.10.10.13   10.10.10.9     TCP       54 [TCP Port numbers reused] 8888 → 9999 [SYN] Seq=0 Win=…
  191 205.203511620 10.10.10.9    10.10.10.13    TCP       54 9999 → 8888 [RST, ACK] Seq=1 Ack=1 Win=0 Len=0
  193 206.203400820 10.10.10.13   10.10.10.9     TCP       54 [TCP Port numbers reused] 8888 → 9999 [SYN] Seq=0 Win=…
  194 206.203968221 10.10.10.9    10.10.10.13    TCP       54 9999 → 8888 [RST, ACK] Seq=1 Ack=1 Win=0 Len=0
  197 207.203564020 10.10.10.13   10.10.10.9     TCP       54 [TCP Port numbers reused] 8888 → 9999 [SYN] Seq=0 Win=…
  198 207.204487121 10.10.10.9    10.10.10.13    TCP       54 9999 → 8888 [RST, ACK] Seq=1 Ack=1 Win=0 Len=0
  199 208.203745521 10.10.10.13   10.10.10.9     TCP       54 [TCP Port numbers reused] 8888 → 9999 [SYN] Seq=0 Win=…
  200 208.204395221 10.10.10.9    10.10.10.13    TCP       54 9999 → 8888 [RST, ACK] Seq=1 Ack=1 Win=0 Len=0
  201 209.203923321 10.10.10.13   10.10.10.9     TCP       54 [TCP Port numbers reused] 8888 → 9999 [SYN] Seq=0 Win=…
  202 209.204438821 10.10.10.9    10.10.10.13    TCP       54 9999 → 8888 [RST, ACK] Seq=1 Ack=1 Win=0 Len=0

▸ Frame 178: 54 bytes on wire (432 bits), 54 bytes captured (432 bits) on interface eth0, id 0
▸ Ethernet II, Src: MS-NLB-PhysServer-21_5d:14:a7:80 (02:15:5d:14:a7:80), Dst: MS-NLB-PhysServer-21_5d:14:a7:83 (02:15:5d:14:a7:83)
▸ Internet Protocol Version 4, Src: 10.10.10.13, Dst: 10.10.10.9
▸ Transmission Control Protocol, Src Port: 8888, Dst Port: 9999, Seq: 0, Len: 0

0000  02 15 5d 14 a7 83 02 15  5d 14 a7 80 08 00 45 00   ··]·····]·····E·
0010  00 28 53 00 00 00 40 06  ff ad 0a 0a 0a 0d 0a 0a   ·(S···@·········
0020  0a 09 22 b8 27 0f 25 10  00 00 00 00 00 00 50 02   ··"·'·%·······P·
0030  02 00 16 e2 00 00                                  ······
```

Figure 19: Wireshark capture of network traffic

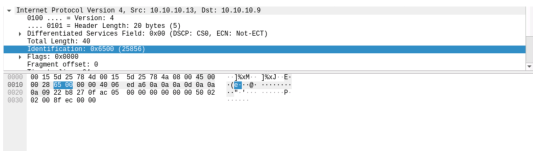
Upon analysis, we find each character of the message string being sent in individual packets over the network. The covert_tcp program utilizes the identification field of the IP protocol and replaces it, one character at a time, with the characters of the string to send the message.

Below are sample packets showing characters "S", "e", "c", "r", "e" and "t" in the IP identification field respectively, for the string "Secret".



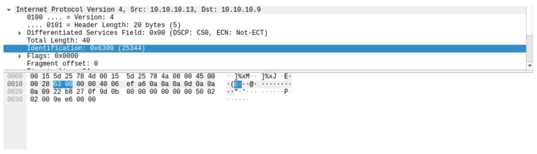
```
▾ Internet Protocol Version 4, Src: 10.10.10.13, Dst: 10.10.10.9
    0100 .... = Version: 4
    .... 0101 = Header Length: 20 bytes (5)
  ▸ Differentiated Services Field: 0x00 (DSCP: CS0, ECN: Not-ECT)
    Total Length: 40
    Identification: 0x5300 (21248)
  ▸ Flags: 0x0000
    Fragment offset: 0

0000  00 15 5d 25 78 4d 00 15  5d 25 78 4a 08 00 45 00   ··]%xM··]%xJ··E·
0010  00 28 53 00 00 00 40 06  ff a6 0a 0a 0a 0d 0a 0a   ·(S···@·········
0020  0a 09 22 b8 27 0f e4 01  00 00 00 00 00 00 50 02   ··"·'·········P·
0030  02 00 57 f0 00 00                                  ··W···
```

Figure 20: Character "S"



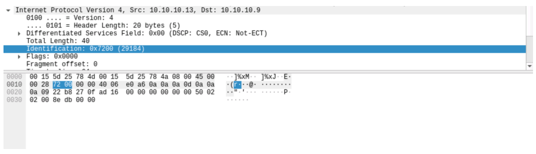
```
▾ Internet Protocol Version 4, Src: 10.10.10.13, Dst: 10.10.10.9
    0100 .... = Version: 4
    .... 0101 = Header Length: 20 bytes (5)
  ▸ Differentiated Services Field: 0x00 (DSCP: CS0, ECN: Not-ECT)
    Total Length: 40
    Identification: 0x6500 (25856)
  ▸ Flags: 0x0000
    Fragment offset: 0

0000  00 15 5d 25 78 4d 00 15  5d 25 78 4a 08 00 45 00   ··]%xM··]%xJ··E·
0010  00 28 65 00 00 00 40 06  ed a6 0a 0a 0a 0d 0a 0a   ·(e···@·········
0020  0a 09 22 b8 27 0f ac 05  00 00 00 00 00 00 50 02   ··"·'·········P·
0030  02 00 8f ec 00 00                                  ······
```

Figure 21: Character "e"



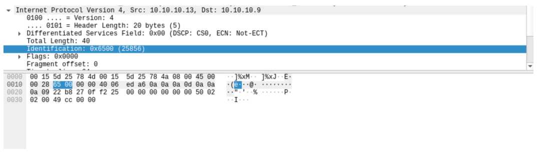
```
▾ Internet Protocol Version 4, Src: 10.10.10.13, Dst: 10.10.10.9
    0100 .... = Version: 4
    .... 0101 = Header Length: 20 bytes (5)
  ▸ Differentiated Services Field: 0x00 (DSCP: CS0, ECN: Not-ECT)
    Total Length: 40
    Identification: 0x6300 (25344)
  ▸ Flags: 0x0000
    Fragment offset: 0

0000  00 15 5d 25 78 4d 00 15  5d 25 78 4a 08 00 45 00   ··]%xM··]%xJ··E·
0010  00 28 63 00 00 00 40 06  ef a6 0a 0a 0a 0d 0a 0a   ·(c···@·········
0020  0a 09 22 b8 27 0f 9d 0b  00 00 00 00 00 00 50 02   ··"·'·········P·
0030  02 00 9e e6 00 00                                  ······
```

Figure 22: Character "c"



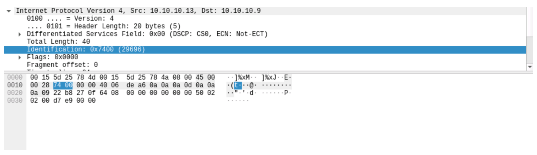
```
▾ Internet Protocol Version 4, Src: 10.10.10.13, Dst: 10.10.10.9
    0100 .... = Version: 4
    .... 0101 = Header Length: 20 bytes (5)
  ▸ Differentiated Services Field: 0x00 (DSCP: CS0, ECN: Not-ECT)
    Total Length: 40
    Identification: 0x7200 (29184)
  ▸ Flags: 0x0000
    Fragment offset: 0

0000  00 15 5d 25 78 4d 00 15  5d 25 78 4a 08 00 45 00   ··]%xM··]%xJ··E·
0010  00 28 72 00 00 00 40 06  e0 a6 0a 0a 0a 0d 0a 0a   ·(r···@·········
0020  0a 09 22 b8 27 0f ad 16  00 00 00 00 00 00 50 02   ··"·'·········P·
0030  02 00 8e db 00 00                                  ······
```

Figure 23: Character "r"



```
▾ Internet Protocol Version 4, Src: 10.10.10.13, Dst: 10.10.10.9
    0100 .... = Version: 4
    .... 0101 = Header Length: 20 bytes (5)
  ▸ Differentiated Services Field: 0x00 (DSCP: CS0, ECN: Not-ECT)
    Total Length: 40
    Identification: 0x6500 (25856)
  ▸ Flags: 0x0000
    Fragment offset: 0

0000  00 15 5d 25 78 4d 00 15  5d 25 78 4a 08 00 45 00   ··]%xM··]%xJ··E·
0010  00 28 65 00 00 00 40 06  ed a6 0a 0a 0a 0d 0a 0a   ·(e···@·········
0020  0a 09 22 b8 27 0f f2 25  00 00 00 00 00 00 50 02   ··"·'··%······P·
0030  02 00 49 cc 00 00                                  ··I···
```

Figure 24: Character "e"



```
▾ Internet Protocol Version 4, Src: 10.10.10.13, Dst: 10.10.10.9
    0100 .... = Version: 4
    .... 0101 = Header Length: 20 bytes (5)
  ▸ Differentiated Services Field: 0x00 (DSCP: CS0, ECN: Not-ECT)
    Total Length: 40
    Identification: 0x7400 (29696)
  ▸ Flags: 0x0000
    Fragment offset: 0

0000  00 15 5d 25 78 4d 00 15  5d 25 78 4a 08 00 45 00   ··]%xM··]%xJ··E·
0010  00 28 74 00 00 00 40 06  de a6 0a 0a 0a 0d 0a 0a   ·(t···@·········
0020  0a 09 22 b8 27 0f 64 08  00 00 00 00 00 00 50 02   ··"·'··d······P·
0030  02 00 d7 e9 00 00                                  ······
```

Figure 25: Character "t"

4.2.3 Summary/Learning Points

This demonstration of a covert channel using Covert_TCP utilizes the IP identification field. Using common network traffic analysis tools such as Wireshark, it is relatively easy to spot suspicious/malformed packets. Moreover, the program is very slow in transmitting data (about one packet per second). The transfer rate is limited to one packet per second to ensure packets do not arrive out of sync.

It requires proper decoding at the remote end and is painstakingly slow as each bit of information needs to be sent one packet at a time. Because we are subjugating the TCP/IP protocol for a purpose not designed, the normal reliability modes are non-existent and in essence functions much like UDP.

5 Evaluation and Findings

5.1 Comparison

We can see that Covert_TCP can be improved on as it is easily detectable on wireshark and the message hidden within the packets can be found. In comparison it can hold more data compared to our out-of-order method, but since Covert_TCP is so detectable, even without any specialized wireshark filters, we can't really call it covert.

5.2 Limitations

5.2.1 Out-of-order Packets

A big limitation of this method is that within the script we had to manually increase the amount of packets with more if else statements. This made it tedious to increase the size of the data we could send.

Another limitation is that due to TTL being the starting signal for the program, it CANNOT be a default value, as our receiver would not be able to distinguish it from normal traffic. If the receiving network were to block any odd TTL values other than default this would immediately hit a wall.

5.2.2 Covert_TCP

As we can see from the proof of concept, a big limitation to covert channels is sizing and timing. As the message size becomes longer, it becomes harder and harder to send more data without being discovered or suspicious.

Certain applications like Wireshark also automatically flag packets when using Covert_TCP, making sustained communication difficult, however it is still hard to figure out what is going on unless one can correctly guess a covert channel attack.

5.2.3 Both Methods

The final and biggest limitation is that both methods require an already compromised listener to be able to receive messages. This means that while covert channels can maintain a harder to detect connection, the initial compromise might still raise alarms.

5.3 Future Enhancements

5.3.1 IPv6

Right now this project is not compatible with IPv6, however we could add IPv6 compatibility. It would require a bit of overhaul, as IPv6 headers are different from IPv4 headers.

5.3.2 Command and Control

Our receiving script currently only prints whether it was successful or not. We could extend this functionality to run specified scripts that have been placed prior or run a specific file if receiving a certain order of packets.

5.3.3 Obfuscation

Another enhancement that can be performed is obfuscation. As the messages are already hard to decipher, we will instead be obfuscating the callback IP, further reducing traceability.

5.3.4 Noise

As mentioned earlier, we have no real tolerance for noise. An enhancement or fix for this would be to have redundancies in place to handle packet loss or if noise corrupts the message. This would still be hard to implement, as the amount of bits we have is so small, it would be hard to use any for redundancies or checksums.

6 Conclusion

Not a lot of research and analysis has gone into covert channels. However, as is the norm in cyber security, it is quite possible to discover more ways to exploit the current system.

With our proof of concept in out-of-order packets, it can be used as another tool to disguise command and control commands to look like regular traffic, concealing the involvement with the victim machine.

References

[1] A note on the confinement problem. https://www.microsoft.com/en-us/research/wp-content/uploads/2016/02/acrobat-24.pdf. Last Accessed: 2021-09-20.

[2] Trends and challenges in network covert channels countermeasures. https://www.mdpi.com/2076-3417/11/4/1641/pdf. Last Accessed: 2021-09-20.

[3] Covert channel analysis and data hiding in tcp/ip. http://www.security-science.com/pdf/covert-channel-analysis-and-data-hiding-in-tcp-ip.pdf. Last Accessed: 2021-09-20.

[4] Internet protocol. https://datatracker.ietf.org/doc/html/rfc791. Last Accessed: 2021-09-20.

[5] Transmission control protocol. https://www.rfc-editor.org/rfc/pdfrfc/rfc793.txt.pdf. Last Accessed: 2021-09-20.

[6] Improved packet reordering metrics. https://datatracker.ietf.org/doc/html/rfc5236. Last Accessed: 2021-09-20.

[7] A study of internet packet reordering. https://doi.org/10.1007/978-3-540-25978-7_36. Last Accessed: 2021-09-20.

[8] Covert channels in the tcp/ip protocol suite. https://firstmonday.org/ojs/index.php/fm/article/view/528/449. Last Accessed: 2021-09-20.