COVERT CHANNEL

By:

Jeffrey Sasaki

British Columbia Institute of Technology

COMP 8505 – Assignment 1

Aman Abdulla

April 27, 2015

Contents

Introduction

Craig Rowland’s implementation of the covert channel utilizes the IP layer’s Identification field which is a 16 bit field which is capable of holding 2 ASCII characters per packet. Additionally, Rowland also mentioned in his documentation that the TCP initial sequence number field and TCP Acknowledge number field are also viable options for hiding data for covert channel.

The problem he mentioned is that the packets sent are not reliable, due to the fact that he does not ACK the packets for retries.

The covert channel I implemented uses the source port field while enabling the “E” (ECN / Explicit Congestion Notification) flag bit to denote that the packet is a covert message.

Requirements

- Technique for embedding covert data into the headers must be one that is not covered by any of the techniques in the paper.
- Only use the TCP, UDP, or IP headers for this exercise.
- Show all the data supporting the success (or lack thereof) of your data embedding scheme.
- Critique and analyze Rowland’s code. Then identify one or more weaknesses in the code and show how you would rectify those. Then using the base code, work with it and modify it to suit a method of sending data covertly other than what is already being done in the code.

Implementation

The program was written in python with the help of scapy, a packet crafting API. There are two programs:

client.py – the covert message sender

server.py – the covert message receiver

The client program crafts a packet in a way that the "E" flag (ECN) is set to denote that the packet is a covert message and the message itself is stored in the source port.

The E flag bit is the choice of flagging a covert message for two important reasons:

1. It is rarely used in normal TCP/IP connection.
2. It requires strict compliance on ECN processing between two communicating machines[1]

The ports are redundant for this implementation, since we are not establishing a three way handshake, rather, we are expecting a RST and ACK back from the server. The message itself was implemented in the source port and will not alter the behavior of the program if it was implemented in the destination port. The destination port in this implementation is completely random.

Usage:

Scapy must be pre-installed prior to running the program. Scapy can be found at http://www.secdev.org/projects/scapy/. Simply unzip the zip file and run:

```
python setup.py install
```

The programs can be executed using the following commands:

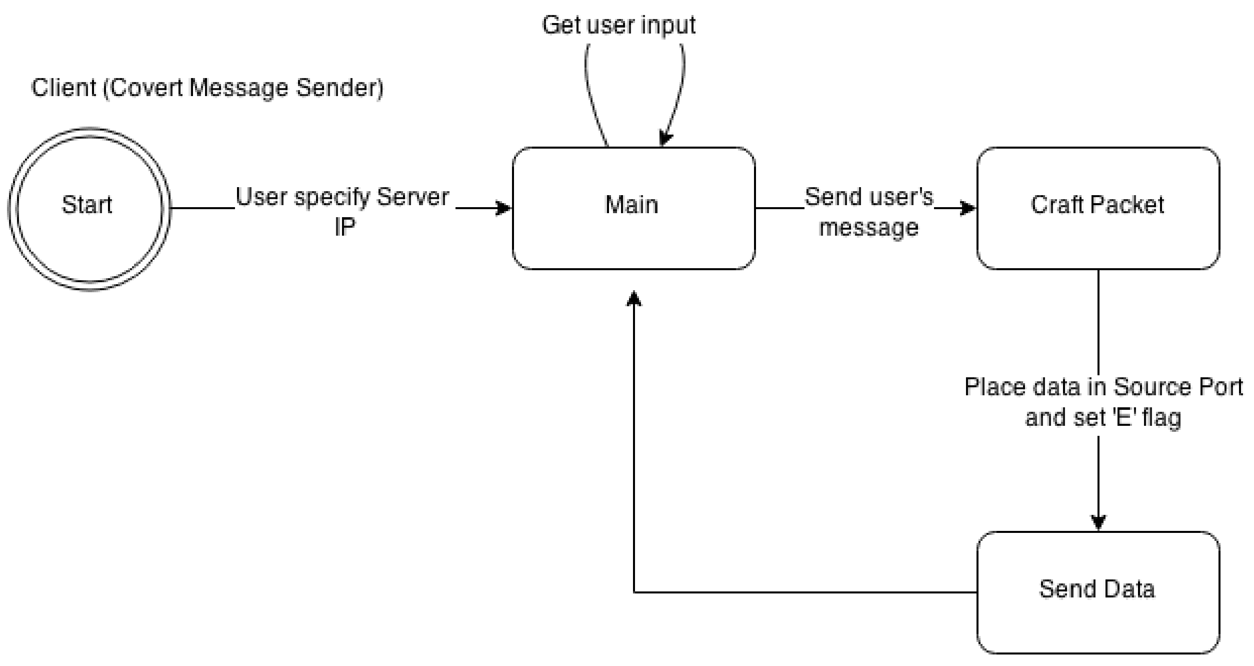
[1] http://tools.ietf.org/html/rfc6040#section-5.2

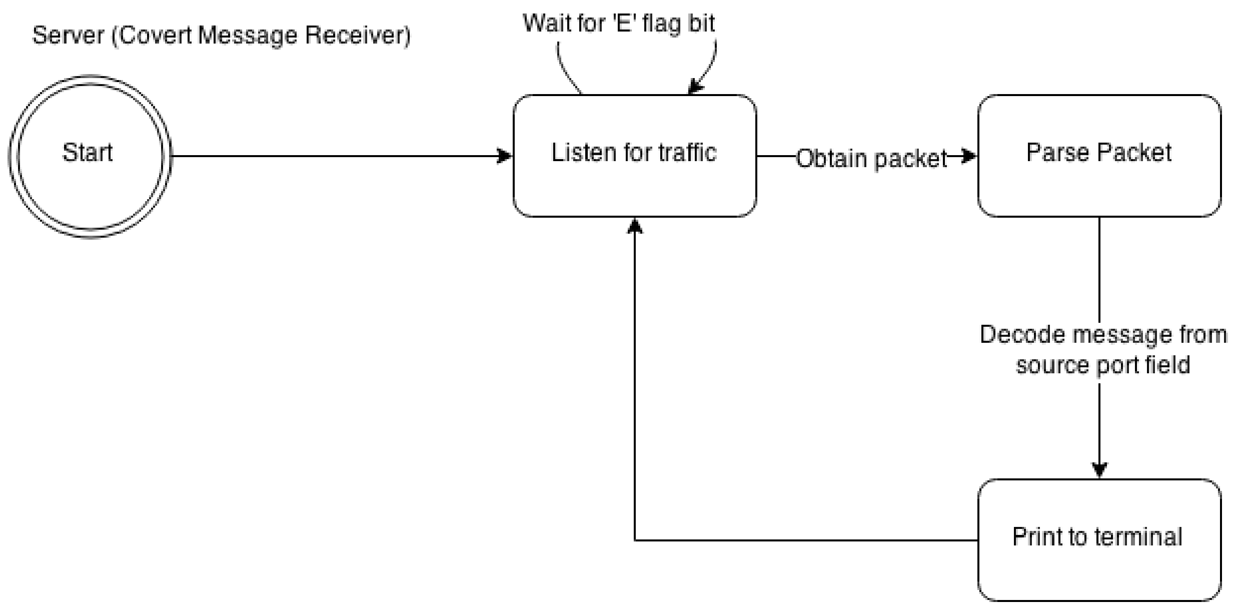
```
$ python client.py <host_ip>
```

```
$ python server.py
```

Note: the user must be a root user to use the program.

The following is a diagram to show the flow of the two programs:

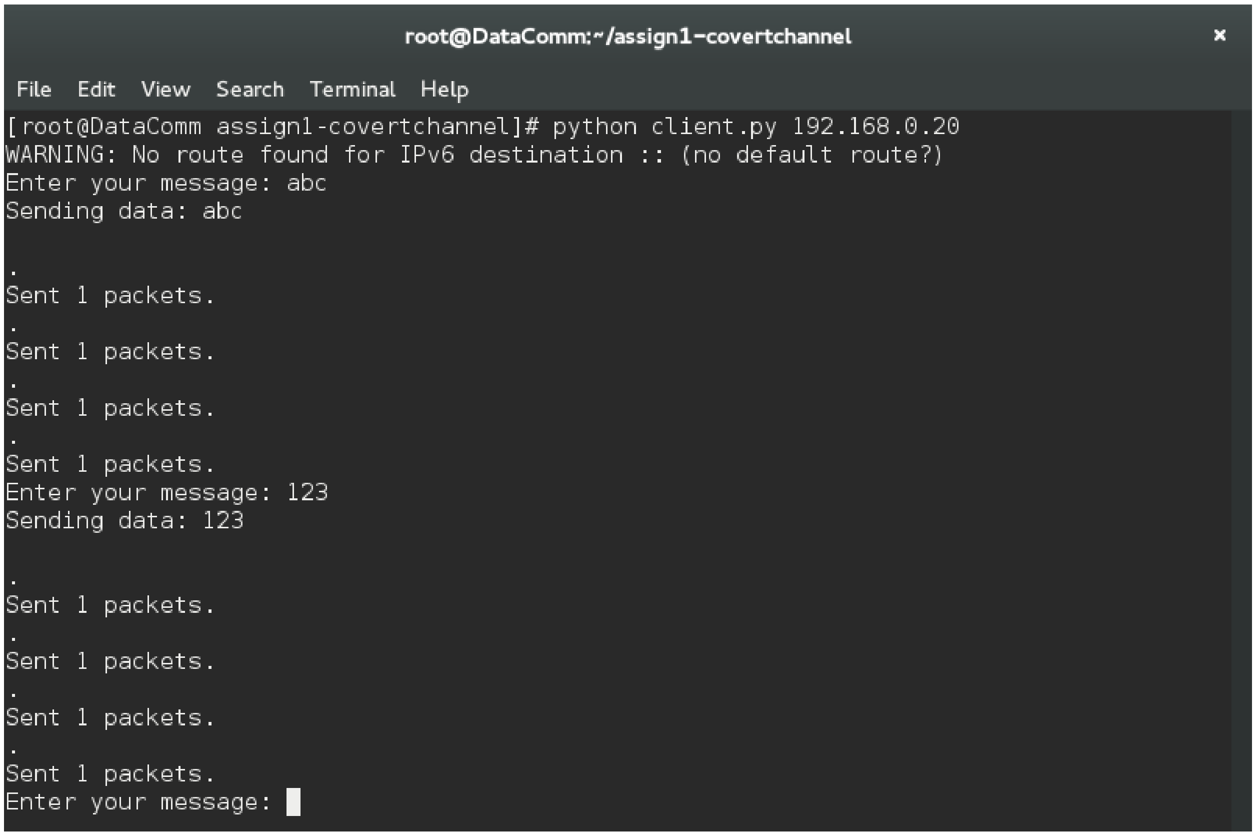




Testing

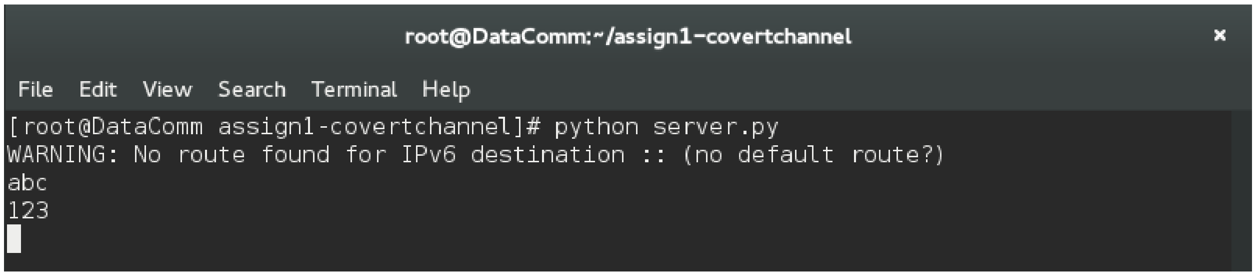
The easiest way to identify that the program itself is to input a pattern ("abc" and "123"), such that the values of the characters will be sequential.

Below is the screenshot of the client program:


root@DataComm:~/assign1-covertchannel
File Edit View Search Terminal Help

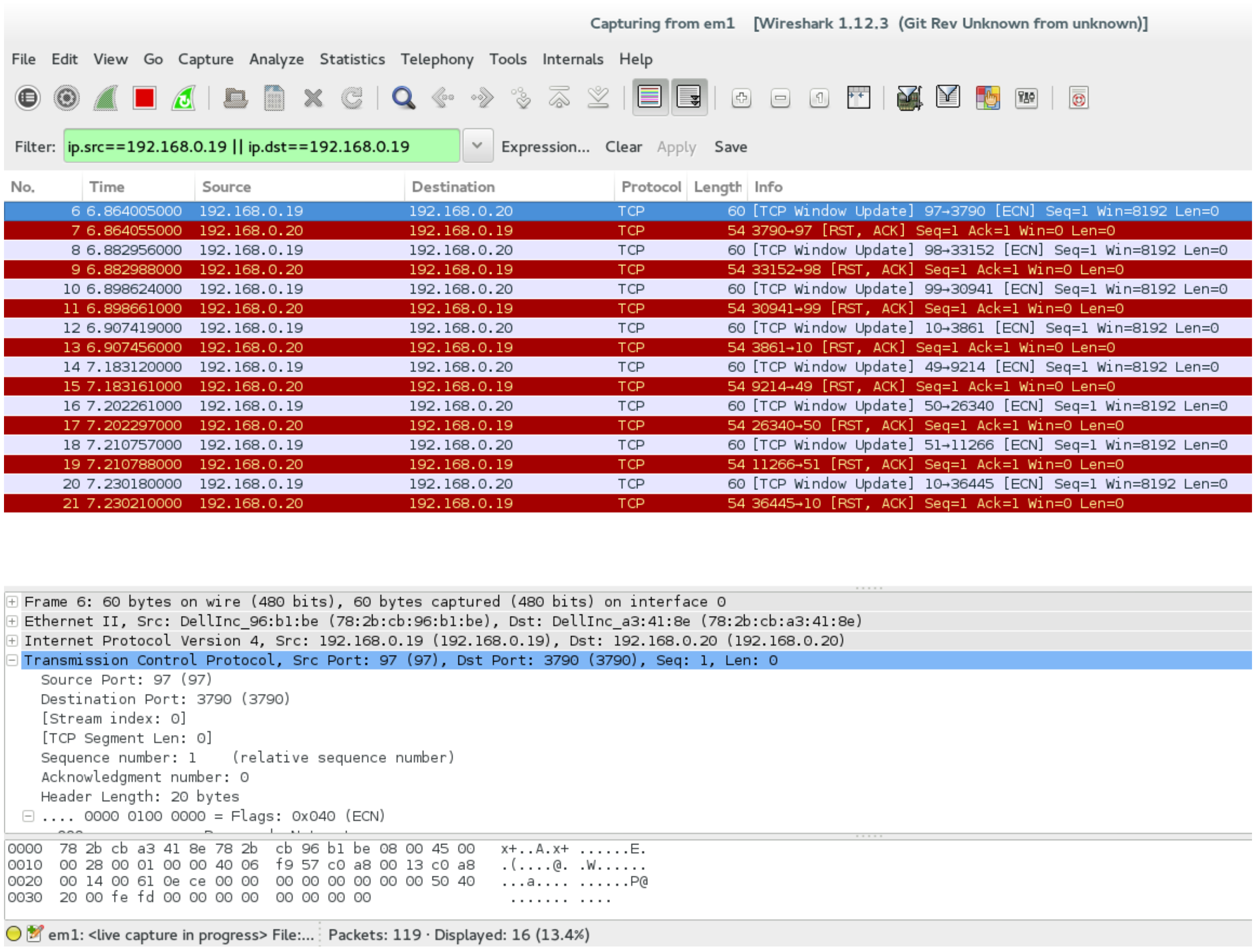
```
[root@DataComm assign1-covertchannel]# python client.py 192.168.0.20
WARNING: No route found for IPv6 destination :: (no default route?)
Enter your message: abc
Sending data: abc

.
Sent 1 packets.
.
Sent 1 packets.
.
Sent 1 packets.
.
Sent 1 packets.
Enter your message: 123
Sending data: 123

.
Sent 1 packets.
.
Sent 1 packets.
.
Sent 1 packets.
.
Sent 1 packets.
Enter your message: 
```

Below is the screenshot of the server program:


root@DataComm:~/assign1-covertchannel
File Edit View Search Terminal Help

```
[root@DataComm assign1-covertchannel]# python server.py
WARNING: No route found for IPv6 destination :: (no default route?)
abc
123
```

The Wireshark screenshot below shows the source port sequentially increasing (97, 98, 99 & 49, 50, 51) for “abc” and “123” respectively. The 10 following each message denotes a new line character. The screenshot also indicate the ECN flag being set.


Capturing from em1   [Wireshark 1.12.3 (Git Rev Unknown from unknown)]
File Edit View Go Capture Analyze Statistics Telephony Tools Internals Help
Filter: ip.src==192.168.0.19 || ip.dst==192.168.0.19 Expression... Clear Apply Save

```
No.  Time         Source        Destination   Protocol Length Info
   6 6.864005000  192.168.0.19  192.168.0.20  TCP      60 [TCP Window Update] 97→3790 [ECN] Seq=1 Win=8192 Len=0
   7 6.864055000  192.168.0.20  192.168.0.19  TCP      54 3790→97 [RST, ACK] Seq=1 Ack=1 Win=0 Len=0
   8 6.882956000  192.168.0.19  192.168.0.20  TCP      60 [TCP Window Update] 98→33152 [ECN] Seq=1 Win=8192 Len=0
   9 6.882988000  192.168.0.20  192.168.0.19  TCP      54 33152→98 [RST, ACK] Seq=1 Ack=1 Win=0 Len=0
  10 6.898624000  192.168.0.19  192.168.0.20  TCP      60 [TCP Window Update] 99→30941 [ECN] Seq=1 Win=8192 Len=0
  11 6.898661000  192.168.0.20  192.168.0.19  TCP      54 30941→99 [RST, ACK] Seq=1 Ack=1 Win=0 Len=0
  12 6.907419000  192.168.0.19  192.168.0.20  TCP      60 [TCP Window Update] 10→3861 [ECN] Seq=1 Win=8192 Len=0
  13 6.907456000  192.168.0.20  192.168.0.19  TCP      54 3861→10 [RST, ACK] Seq=1 Ack=1 Win=0 Len=0
  14 7.183120000  192.168.0.19  192.168.0.20  TCP      60 [TCP Window Update] 49→9214 [ECN] Seq=1 Win=8192 Len=0
  15 7.183161000  192.168.0.20  192.168.0.19  TCP      54 9214→49 [RST, ACK] Seq=1 Ack=1 Win=0 Len=0
  16 7.202261000  192.168.0.19  192.168.0.20  TCP      60 [TCP Window Update] 50→26340 [ECN] Seq=1 Win=8192 Len=0
  17 7.202297000  192.168.0.20  192.168.0.19  TCP      54 26340→50 [RST, ACK] Seq=1 Ack=1 Win=0 Len=0
  18 7.210757000  192.168.0.19  192.168.0.20  TCP      60 [TCP Window Update] 51→11266 [ECN] Seq=1 Win=8192 Len=0
  19 7.210788000  192.168.0.20  192.168.0.19  TCP      54 11266→51 [RST, ACK] Seq=1 Ack=1 Win=0 Len=0
  20 7.230180000  192.168.0.19  192.168.0.20  TCP      60 [TCP Window Update] 10→36445 [ECN] Seq=1 Win=8192 Len=0
  21 7.230210000  192.168.0.20  192.168.0.19  TCP      54 36445→10 [RST, ACK] Seq=1 Ack=1 Win=0 Len=0
```

```
⊞ Frame 6: 60 bytes on wire (480 bits), 60 bytes captured (480 bits) on interface 0
⊞ Ethernet II, Src: DellInc_96:b1:be (78:2b:cb:96:b1:be), Dst: DellInc_a3:41:8e (78:2b:cb:a3:41:8e)
⊞ Internet Protocol Version 4, Src: 192.168.0.19 (192.168.0.19), Dst: 192.168.0.20 (192.168.0.20)
⊟ Transmission Control Protocol, Src Port: 97 (97), Dst Port: 3790 (3790), Seq: 1, Len: 0
    Source Port: 97 (97)
    Destination Port: 3790 (3790)
    [Stream index: 0]
    [TCP Segment Len: 0]
    Sequence number: 1    (relative sequence number)
    Acknowledgment number: 0
    Header Length: 20 bytes
  ⊟ .... 0000 0100 0000 = Flags: 0x040 (ECN)
```

```
0000  78 2b cb a3 41 8e 78 2b  cb 96 b1 be 08 00 45 00   x+..A.x+ ......E.
0010  00 28 00 01 00 00 40 06  f9 57 c0 a8 00 13 c0 a8   .(....@. .W......
0020  00 14 00 61 0e ce 00 00  00 00 00 00 00 00 50 40   ...a.... ......P@
0030  20 00 fe fd 00 00 00 00  00 00 00 00                ....... ....
```

em1: <live capture in progress> File:...   Packets: 119 · Displayed: 16 (13.4%)

Conclusion

Rowland and I have implemented a convert channel sending one character per packet; however, it is important to note that there are other fields in the TCP and IP header that are capable of sending 32 bits of data, such as the Source IP. Aside from the ones Rowland mentioned in his document, there are multiple fields and protocols that are capable of establishing a covert channel:

- ICMP type and code
- UDP source port
- TCP Reserved field
- TCP source port
- TCP options field

Like Rowland's implementation, this implementation suffers heavily on reliability and negated the "bounce" feature presented in Rowland's implementation.

Future changes of this program should feature a bounce implementation, as well as reliability. In terms of reliability, using another header field, such as the Type of Service field, can be used to denote the packet number and attempt a packet retransmission request if the packets did not arrive in order.