Introduction to Covert Channels & Communications

David Tran

A00801942

COMP 8505 7D

British Columbia Institute of Technology

Aman Abdulla

Monday, September 22 2014

# 

# Table of Contents

# 

# Introduction

The purpose of this assignment is to introduce the users to covert channel communication. We will base many of our observations using Craig Rowlands’ “covert\_tcp.c” work. His work will help be the foundation in observing potential improvements on his application, as well as identifying weaknesses in his application, in his methodologies, or both. Finally, the bulk of this assignment will include the implementation, observation, and testing of our suggested improvements. This will not only build on Rowlands’ work but also broaden our understanding of covert channels.

# Critique on Craig Rowlands’ Work

To begin, let us recap Rowlands’ three methods to using covert channels:

## Method 1: IP Identification Field

Rowlands’ implementation regarding the IP Identification Field (IP ID) requires that we split our data into 1-byte values, and then embed it within the field in hexadecimal, followed by multiplying that hexadecimal value by a predetermined value (such as 256), and then sending it on its way to its destination in random intervals. Finally, on the receiving server, we would parse these values in the IP ID by dividing them by 256. At this point, we would just reverse the process to get our 1-byte value. Eventually, the server would piece together the data collected, resulting in our original data.

### Shortcomings of Method 1

As Rowlands’ stated in his paper, this method is easily falliable to packet filtering at the firewall and the modification of the actual “forged” packet through network address translation, resulting in data loss.

## Method 2: TCP Initial Sequence Number Field

The TCP Initial Sequence Number uses the same implementation as Method 1, except that the implementation occurs in the TCP protocol; specifically, within the Initial Sequence number field.

### Shortcomings of Method 2

While the values in the IP ID field of the IP header can be trivial, The TCP initial sequence number field can become suspicious if there was a vigilant network administrator. By running a packet sniffing application like Wireshark, monitoring legitimate packets between two establishing machines will reveal that the relative sequence number is incremental. If the administrator watches the packets used for Method 2, it will not reveal a relative sequence number in order, thus raising suspicion.

## Method 3: TCP Acknowledge Sequence Number Field “Bounce”

The “Bounce” method uses a tertiary server called the “Bounce Server” in order to allow the covert channel to bypass potentially problematic firewall issues. It also hides the sender by spoofing its IP to be a legitimate server. This makes it that much harder to detect the sender when the bounce server is quite busy.

### Shortcomings of Method 3

This method is dependent on the tertiary server being “trusted” if the target is under heavy security. Otherwise, packet filtering at the target’s network may drop all covert communications. However, by doing some reconnaissance or using some “well-known” Internet websites, the workaround for this shortcoming is quite feasible.

# Suggestions for Improvement

Besides the shortcomings of the individual methods that Rowlands had suggested in his article for covert communications as well as the additional comments provided by Rowlands in his article, there are other all-around concerns that may trigger suspicion from the network administrator. Let us consider the following features of Rowlands’ application:

## The Need for Root Privileges

Root privileges are required to create these custom packets by using raw socket programming. This makes the use of the program to be problematic when communicating over covert channels. Furthermore, it does not appear to have a workaround either. In turn, the only way to “solve” this issue is to make sure that the user (the one doing the covert communicating) has some root access.

## Utilizing a Single Method at a Time

To use the current version of Rowlands’ application, the user needs to specify one method of covert communications. However, it is not unfeasible to utilize more than one field per header per packet at a time or to utilize “cross-field” headers. For example, in the TCP header, it is not wishful thinking to use the Sequence Number as well as, per se, the Urgent Header. Also, it can be feasible to use the IP ID field in the IP Header, in conjunction with a field in the TCP header, such as the Sequence Number. This can make the process of covert communication to be faster, to have redundancy, or both.

## Sending Packets in One-Second Intervals

An alert network administrator may see that packets coming from the same source with the same time interval. Although network traffic can be congested, let us assume that the administrator is seasoned in their role. To mitigate the feeling of suspicion, Rowlands’ application could be extended further by randomizing the time the sender sends out these forged packets. Of course it would increase the total time to communicate over covert channels but the key concept here is to be as patient as possible.

# Proposed Implementation

For the purpose of this assignment, the deliverables will be built on-top of Rowlands’ application and will be used as the base of our code. The code will include two new methods for covert communication as well as any other improvements as time permits. We shall be implementing the following methods:

* The TCP Urgent Pointer & Tagging the URG Flag
* Communication through UDP Source Port
* Simple “Encryption”

# Design Work - Pseudo Code

## The TCP Urgent Pointer & Tagging the URG Flag

void main ( ... )

{

// to create our packet

…

if ( urg var is true )

{

set urg flag to true

set urg pointer to our char

}

…

// to read our packet

…

if ( urg var is true )

{

read from the urg pointer

output from the urg pointer to file

flush output

}

}

## Communication through UDP Source Port

void main ( … )

{

// to create our packet

…

if ( udp var is true )

{

create udp struct

create pseudo udp header struct

populate structure with proper values

udp.source = our char value

open socket

send udp packet

}

// to listen and receive our packet

…

if ( udp var is true )

{

listen on specified udp port

parse value in UDP source port

}

}

## Simple “Encryption”

int encrypt( int raw\_char )

{

initialize scrambled variable

scrambled variable = (raw\_char)(scramble formula)

return scrambled variable

}

int decrypt( int scrambled\_char )

{

initialize unscrambled variable

unscrambled variable = (scrambled\_char)

(inverse scramble formula)

return unscrambled variable

}

// example of usage

void forgepacket ( … )

{

…

// encrypting...

if ( urg var is true )

{

if ( scramble var is true )

{

set urg flag to true

set urg pointer = encrypt( our char )

}

}

…

// decrypting

if ( urg var is true )

{

if ( scramble var is true )

{

decrypt( read from urg pointer )

decrypt( output from the urg

pointer to file )

flush output

}

}

}

# Tools & Equipment

## Hardware

|  |  |  |
| --- | --- | --- |
| * 8GB RAM | * Intel i5 Quad Core | * 500GB HDD |
| * Client Host | * Server Host | * “Bounce” Server (when necessary) |

## Software

|  |  |  |
| --- | --- | --- |
| * Fedora Linux 20 64-bit | * C Programming | * Wireshark |
| * Gedit | * Terminal Console |  |

# Testing Cases

## Names & Aliases

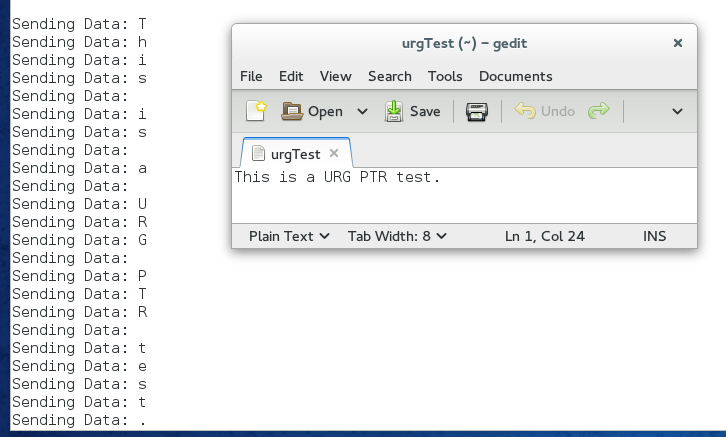
|  |  |
| --- | --- |
| **IP Address of Machine** | **Alias** |
| 192.168.0.13 | Client |
| 192.168.0.14 | Server |
| 192.168.0.24 | “Bounce” Server |

## The TCP Urgent Pointer & Tagging the URG Flag

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Case #** | **Test Case** | **Tools Used** | **Expected Outcome** | **Results** |
| 1 | Client is able to read the specified input file, char by char, and sends the packet. | Wireshark, Terminal,  Gedit | Terminal shows the packet being read and it is the same as the input file specified | PASSED. See results for details. |
| 2 | Client is able to send packets with the Urgent Pointer set and the URG flag tagged | Wireshark | Packets are confirmed with Urgent Pointer set, and values equal to ASCII table equivalent | PASSED. See results for details. |
| 3 | Server is able to receive packets with the Urgent Pointer set and the URG flag tagged | Wireshark | Packets are confirmed with Urgent Pointer set, and values equal to ASCII table equivalent | PASSED. See results for details. |
| 4 | Server is able to store the char values from the Urgent Pointer to output file specified | Wireshark, Terminal,  Gedit | Terminal shows the packet being parsed and the output file is the same as the parsed values, as well as being exactly the same as the client’s input file. | PASSED. See results for details. |

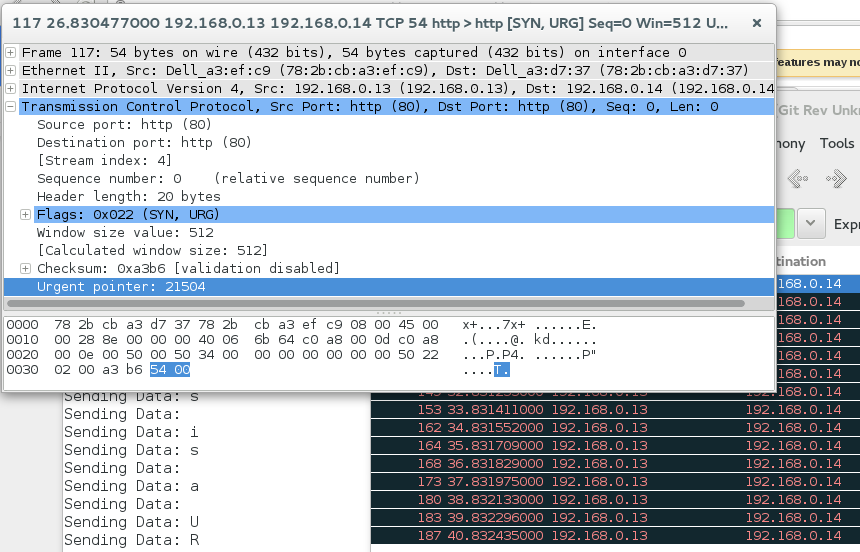
### Evidence - The TCP Urgent Pointer & Tagging the URG Flag

#### Test Case 1: Client is able to read the specified input file, char by char, and sends the packet



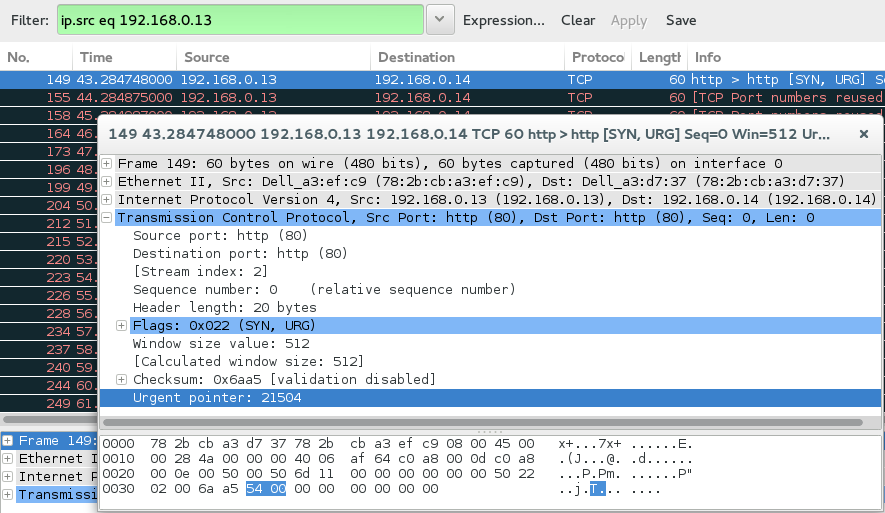
Gedit shows urgTest file; Terminal shows, char by char, values being sent

Test Case 2: Client is able to send packets with the Urgent Pointer set and the URG flag tagged



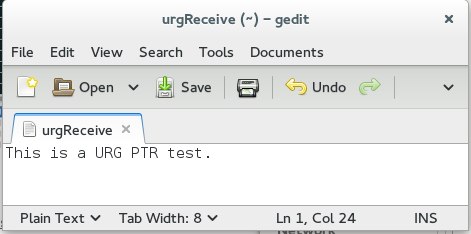
Wireshark datagram details show Urgent Pointer value; It is ASCII Table values of “T” (54)

#### Test Case 3: Server is able to receive packets with the Urgent Pointer set and the URG flag tagged



Wireshark packet capture on the Server side shows the same Urgent Pointer value, same ASCII code and thus, same char as we would expect.

#### Test Case 4: Server is able to store the char values from the Urgent Pointer to output file specified

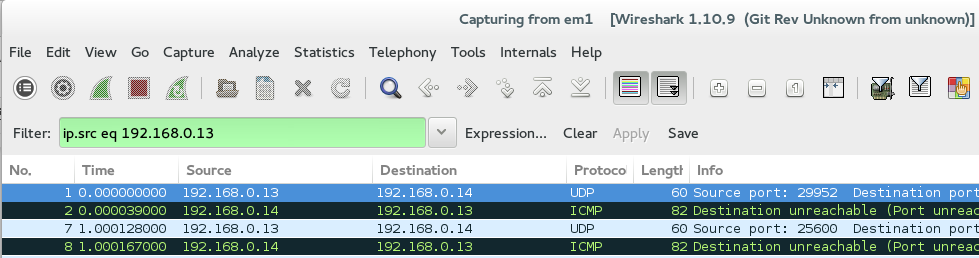


Gedit view of the urgReceive, the output file specified; recall in Test Case 1: message is exactly the same

## Communication through UDP Source Port

We had initially attempted to use the UDP header as a means for covert communications. However, our program ran into errors and although compilation of the program at the time was successful and had no errors, the Server was unable to parse the UDP packets.

What resulted was the packet capture on the Server had received the UDP packet with the source port configured to be our data placeholder. The Server then responded with an ICMP message with a Destination (Port) Unreachable.



On the client side, we were able to see the packets going through using Wireshark. We were also able to visually see the file being read, char by char. Therefore, it might be due to the way we were forging our UDP packet.

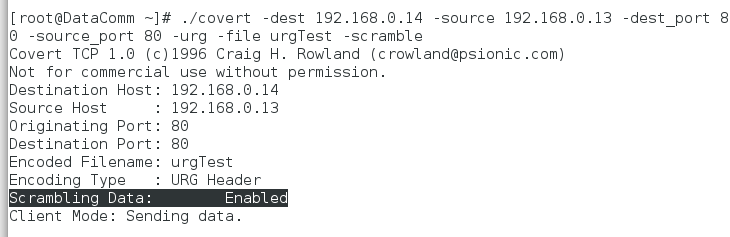
We would have liked to have the inclusion of UDP on this assignment. However, due to time constraints and a lack of comprehension as to why this error occurs forced us to omit this section of intended features.

## Simple “Encryption”

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Case #** | **Test Case** | **Tools Used** | **Expected Outcome** | **Results** |
| 1 | Client is able to enable the scrambling of data | Terminal, Wireshark | Args has “-scramble”; initializing application shows “Data Scrambling: Enabled”; char values do not match ASCII table values | PASSED. See results for details. |
| 2 | Client is able to omit the scrambling of data | Terminal, Wireshark | No “-scramble” arg; initializing app. shows “Data Scrambling: Disabled”; char values match ASCII table values | PASSED. See results for details. |
| 3 | Data scrambling works on IP ID encoding | Terminal,  Wireshark | Packet capture shows a different value than an expected ASCII value | PASSED. See results for details. |
| 4 | Server is able to unscramble data in IP ID encoding | Terminal,  Wireshark,  Gedit | Packet capture shows a different value than an expected ASCII value; Terminal outputs a latin character, char by char; File output is exactly the same as the client | PASSED. See results for details. |
| 5 | Data scrambling works on TCP SEQ encoding | Terminal,  Wireshark | Packet capture shows a different value than an expected ASCII value | PASSED. See results for details. |
| 6 | Server is able to unscramble data in TCP SEQ encoding | Terminal,  Wireshark,  Gedit | Packet capture shows a different value than an expected ASCII value; Terminal outputs a latin character, char by char; File output is exactly the same as the client | PASSED. See results for details. |
| 7 | Data scrambling works with “Bounce” server | Terminal,  Wireshark | Packet capture shows a different value than an expected ASCII value | PASSED. See results for details. |
| 8 | Server is able to unscramble data from “Bounce” server | Terminal,  Wireshark,  Gedit | Packet capture shows a different value than an expected ASCII value; Terminal outputs a latin character, char by char; File output is exactly the same as the client | PASSED. See results for details. |
| 9 | Data scrambling works on Urgent Pointer encoding | Terminal,  Wireshark | Packet capture shows a different value than an expected ASCII value | PASSED. See results for details. |
| 10 | Server is able to unscramble data in Urgent Pointer encoding | Terminal,  Wireshark,  Gedit | Packet capture shows a different value than an expected ASCII value; Terminal outputs a latin character, char by char; File output is exactly the same as the client | PASSED. See results for details. |

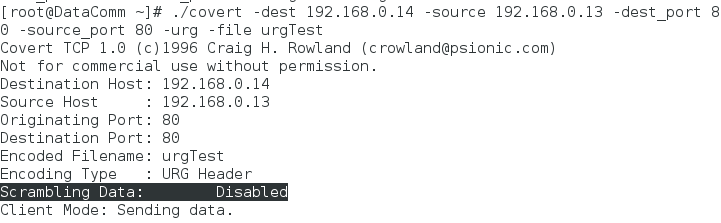
### Evidence - Simple “Encryption”

#### Test Case 1: Client is able to enable the scrambling of data



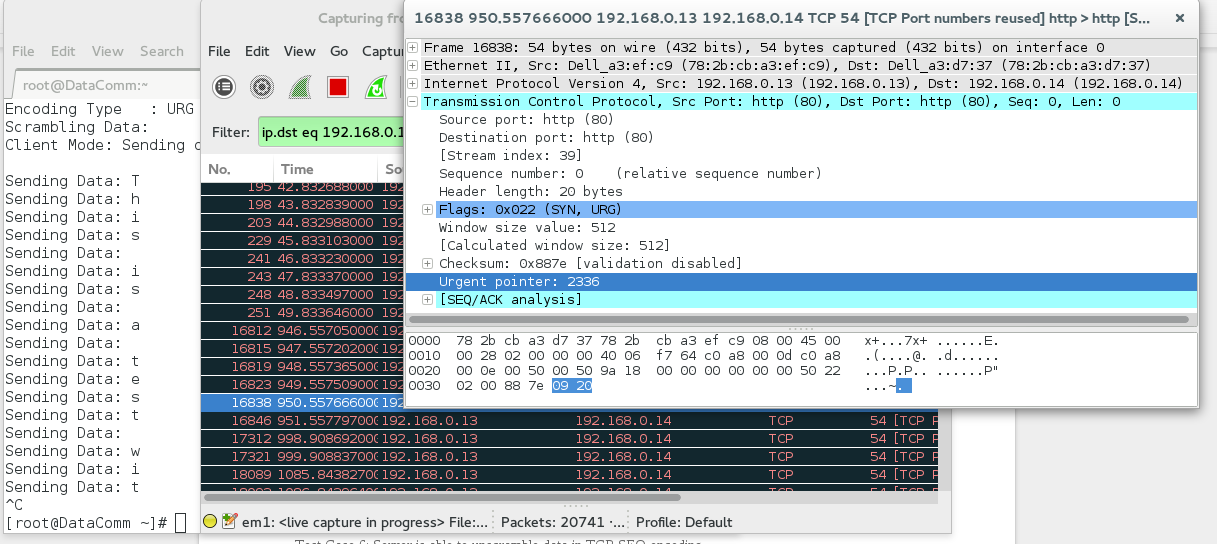
Note the highlighted part; User has enabled data to be scrambled.

#### Test Case 2: Client is able to omit the scrambling of data



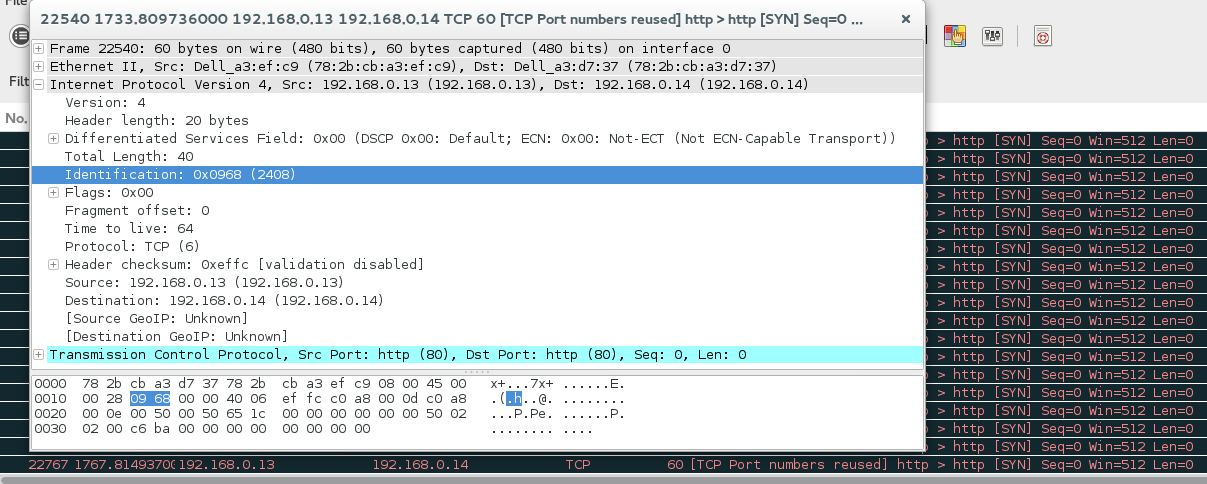
Note the highlighted part; User is able to omit the data scrambling

#### Test Case 3: Data scrambling works on IP ID encoding

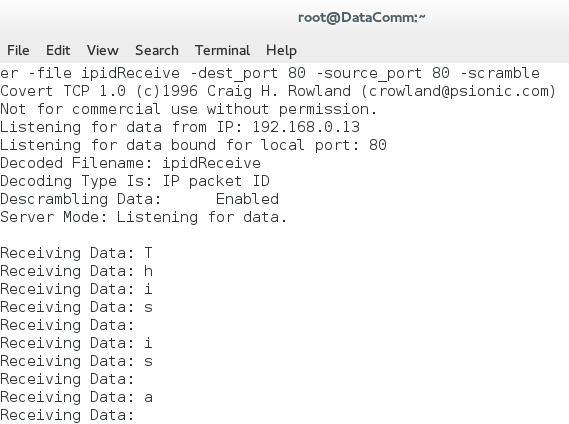


Terminal outputs readable ASCII; Wireshark3 packet capture shows a different value than ASCII

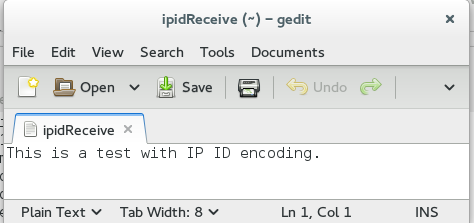
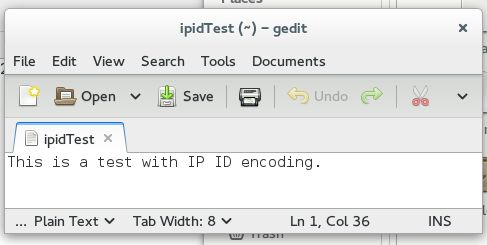
#### Test Case 4: Server is able to unscramble data in IP ID encoding



Value is same as client; Packets are received safely

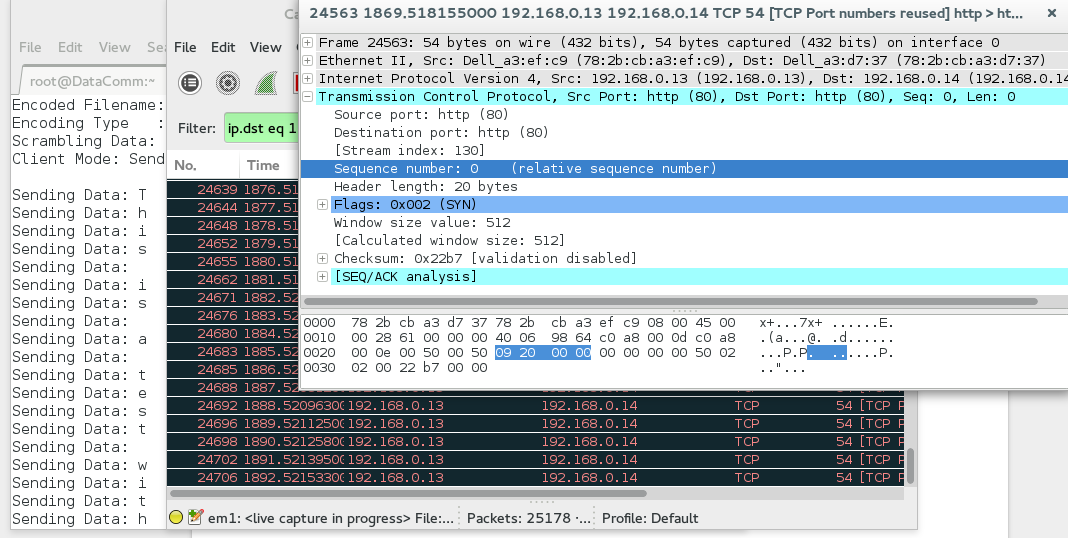


Terminal outputs readable ASCII characters



Client and Server (respectively) with their input and output files; They are the same

#### Test Case 5: Data scrambling works on TCP SEQ encoding

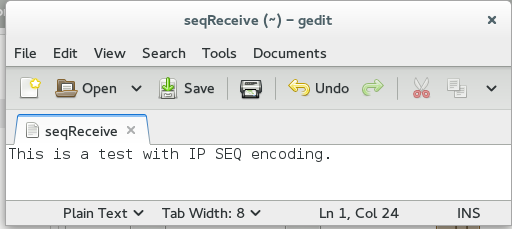
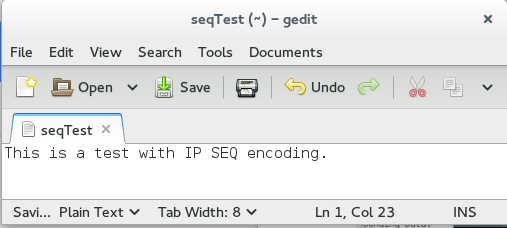


We expect to see the same as Test Case 3; It is.

#### Test Case 6: Server is able to unscramble data in TCP SEQ encoding

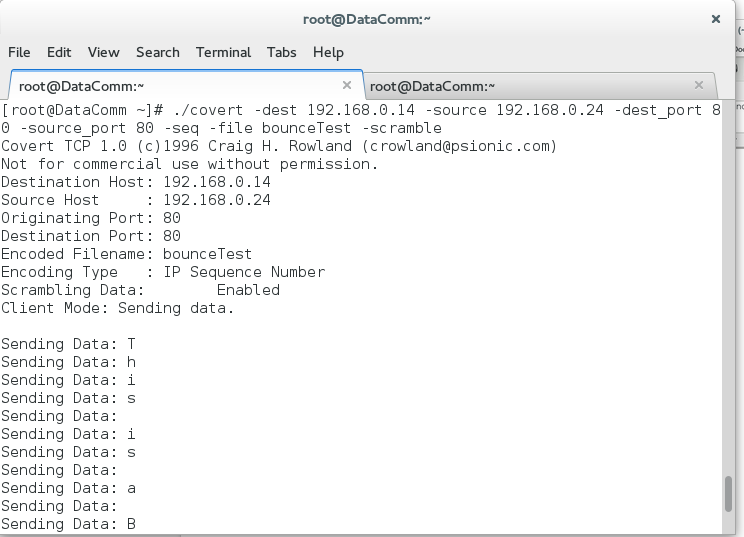


Terminal on Server outputs readable ASCII characters

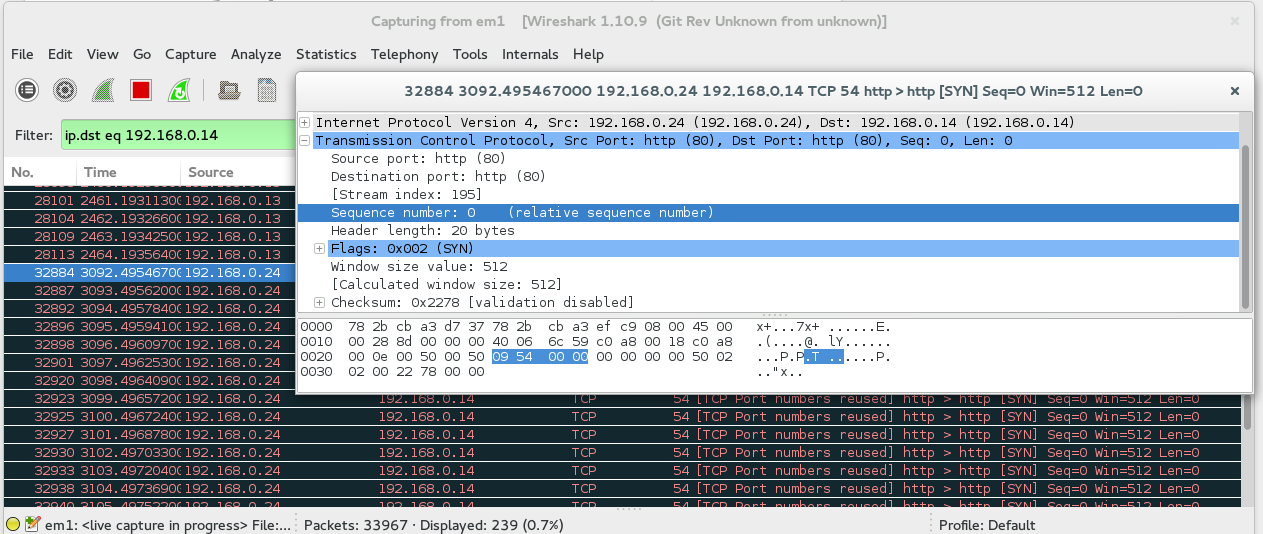


Client and Server (respectively) with their input and output files

#### Test Case 7: Data scrambling works with “Bounce” server

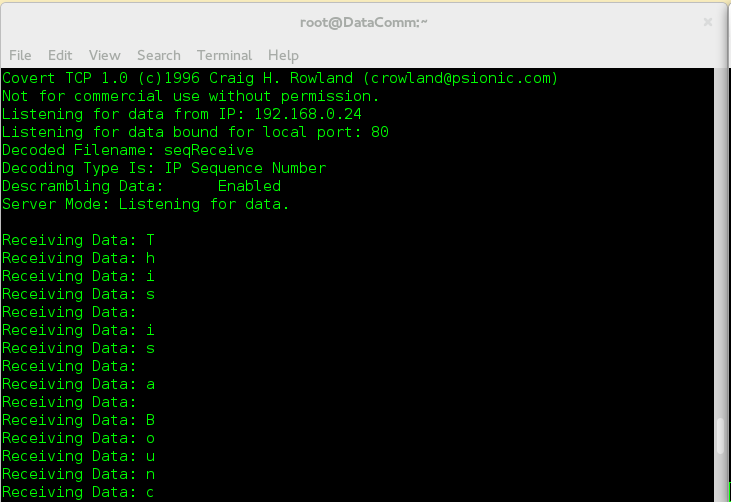


This is the Client spoofing as the Bounce server. Terminal shows readable ASCII characters

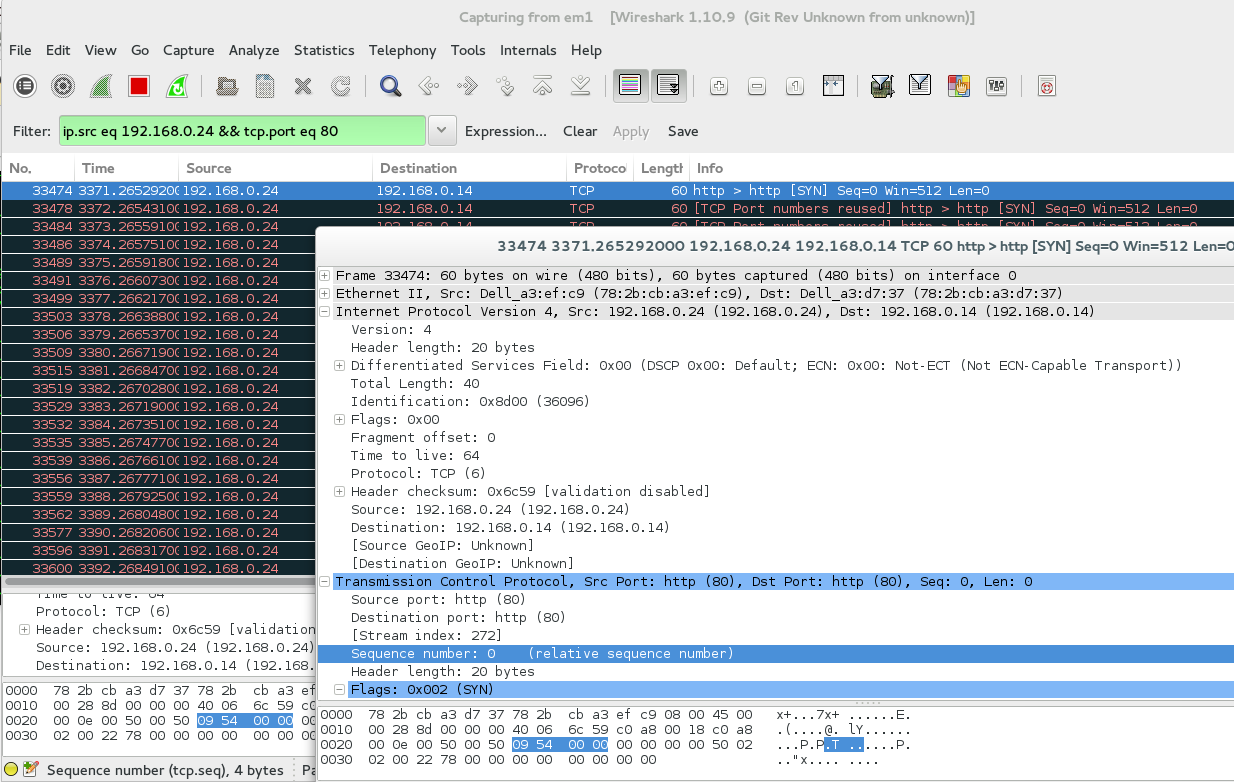


Wireshark capture from Client shows that the characters have been scrambled; It also shows a source IP of our spoof server.

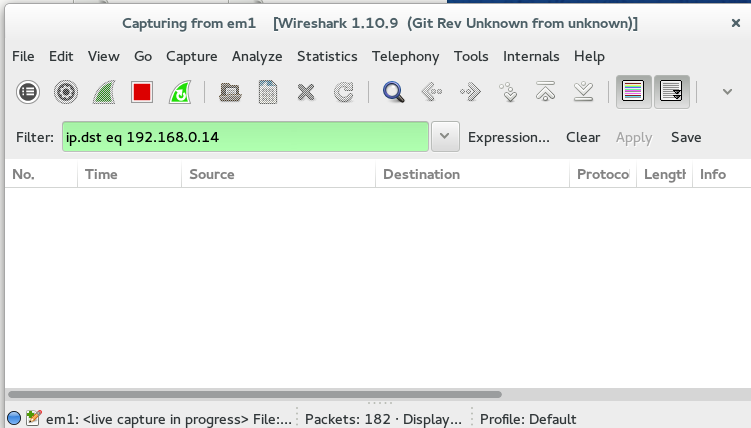
#### Test Case 8: Server is able to unscramble data from “Bounce” server



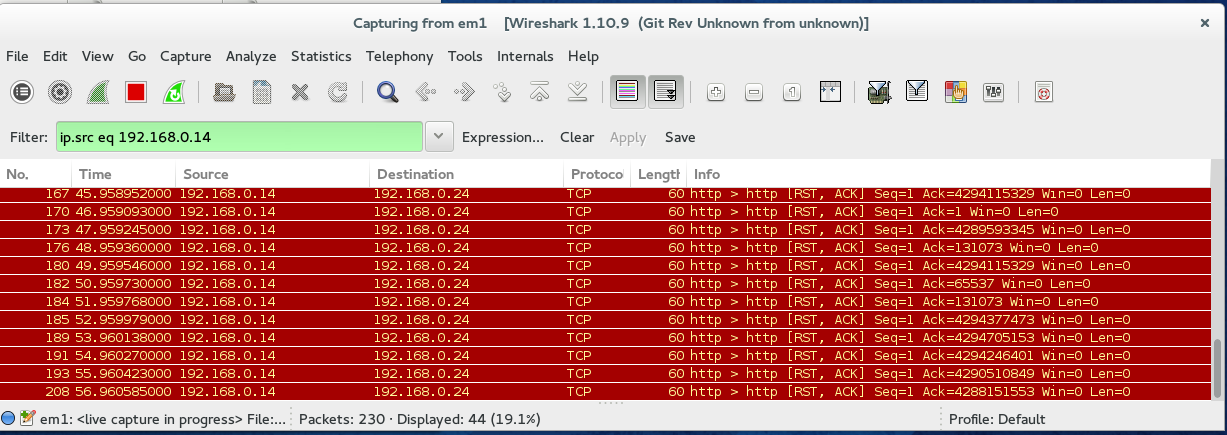
This is our Server; it is receiving data from our spoof server



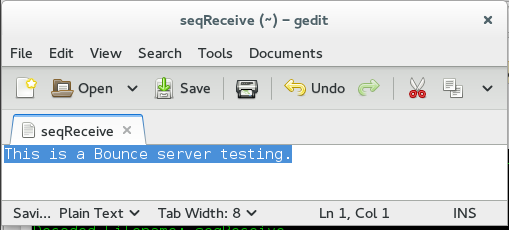
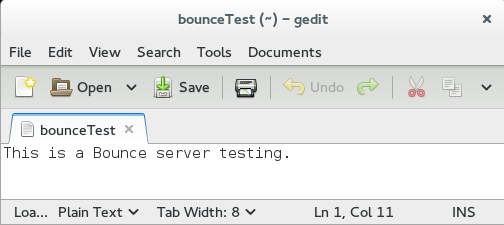
Wireshark on our Server shows the datagram to be from our spoof server; Sequence number the same as our Client



However, on our spoof server, there are no records of it sending the Server data!!

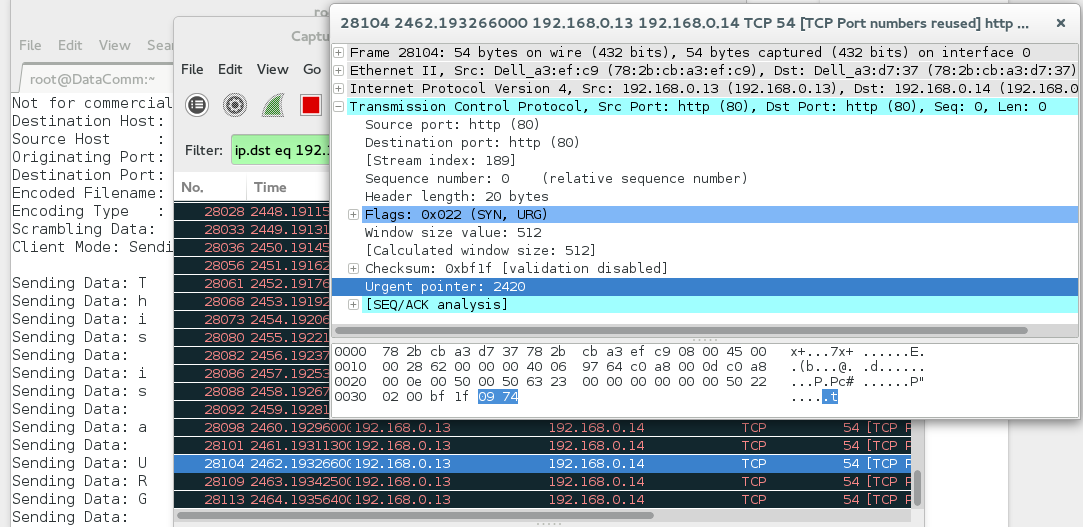


But we do see our Server trying to communicate back with our spoof server here. Interesting



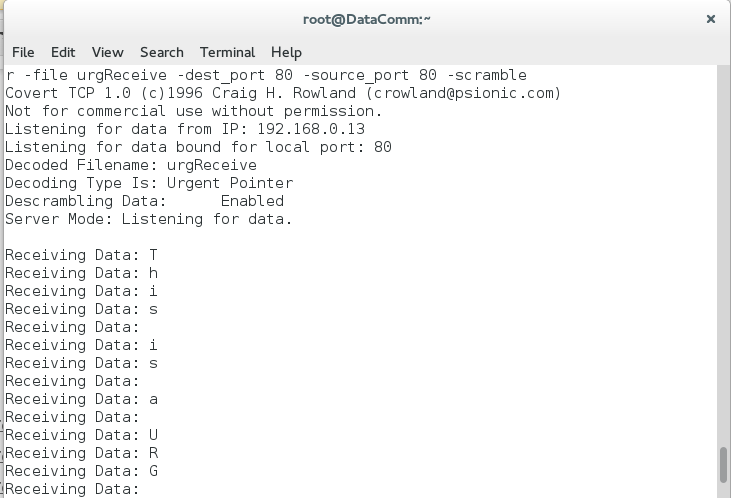
Lastly, a comparison between Client and Server. They are the same

#### Test Case 9: Data scrambling works on Urgent Pointer encoding

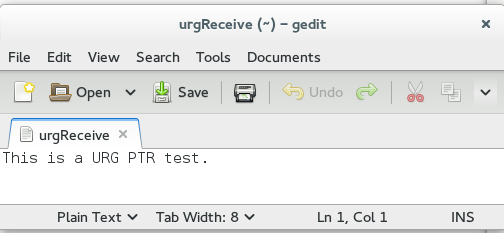
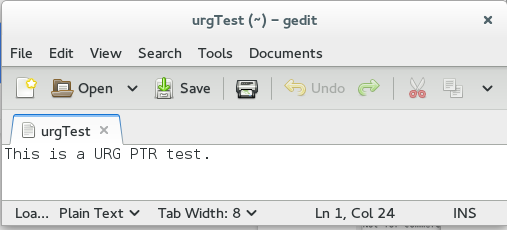


We expect the same as Test Cases 3 and 5; the only difference is Urgent Pointer encoding.

#### Test Case 10: Server is able to unscramble data in Urgent Pointer encoding



Our Server is receiving data as normal, through the Urgent Pointer.



And again, the comparison between Client and Server.

# Conclusion

Although Rowlands’ application was a simple proof of concept, it clearly illustrates the potential for the use of covert channels. The greatest lesson we can take away from this assignment is the further understanding of how to conceal ourselves. In doing so, it allows us to be more sufficiently aware of such communications in the future, as well as, allowing ourselves to communicate covertly if needed in cases of dire emergency.

# Appendix

## Content On-Disk

* Introduction to Covert Channels and Communications
* covert.c