Covert Channels: The Pros and Cons

Tyler Jackson
Data and Network Security
SMU Spring 2015

Table of Contents

List of Figures

Abstract

Covert channels are an important topic for any security professional to at least be familiar with. In order to know how to detect, or prevent unauthorized transmission of confidential data an administrator must know how the attack works. This paper dives into the background for a few different types of attacks including manipulating IP headers and TCP vulnerabilities. Covert channels are non-trivial to set up, and it can be difficult to create a successful covert channel without understanding the protocols that make up the transmitted packets. Even with a successful implementation, it is still difficult to transmit large amounts of data without being detected.

What are Covert Channels?

Covert Channels in the simplest form are a means of transmitting information undetected on a medium. It is a secret form of communication that sits on top of an existing communication channel that violates its security policies. The existence of the channel is hidden altogether enabling a transfer of information undetectable by the user.

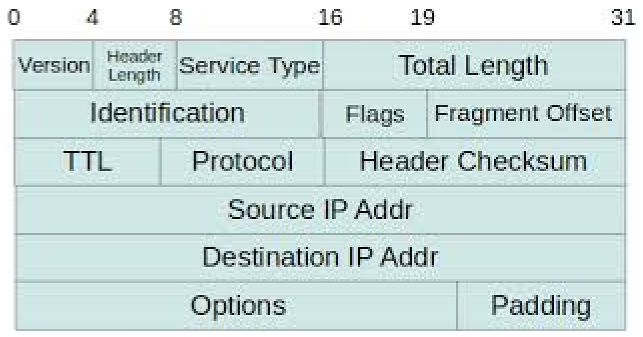
Types of Covert Channels

There are numerous ways to transmit information undetected. Over history the medium in which information could be secretly transmitted has changed. Rulers used to shave their messengers' heads, tattoo a secret message onto their scalp, and then let their hair grow back out to cover up the hidden message. Then when the messenger was delivering one message, the recipient could shave the messenger's head and get a second secret message. As you can imagine this was a rather inefficient process.
Over the years the process and techniques have improved.

Exploiting Header Fields: Background

Some of the more modern techniques utilize properties of TCP/IP headers in network packets. In order to fully understand how these exploits work it is important to take a closer look into how these packets are configured.

Figure 1: IP Header



We will give a brief description of what each of these fields are for. The first four bits are for the version of IP that is being used (IPv4, IPv6, etc.). The next four (bits 5-8) are for the length of the header. The next 8 bits include the differentiated services bits that handle various traffic conditioning, the type of service bits, and the precedence bits that determine the level of service that the packets will receive. The total length bits specify the total length of the packet including the data and headers. The identification, flags, and fragment offset bits are used to fragment and reassemble the packets if the data is too large to fit into one packet. The TTL field refers to the Time to Live field which specifies how many routers a packet can pass through before being dropped. It starts at this value and decrements with each router it passes through. The protocol field specifies the encapsulating protocol. There are numerous options for this field, but some of the more popular ones that we will examine include TCP and UDP. The header checksum is used to validate the integrity of a packet. It is calculated based on the IP Header and helps determine if there are any errors in the headers that were introduced during transmission. The source IP address specifies the address of the original sending machine, and the Destination IP address specifies the address of the final destination machine. Lastly are the options bits that specify other miscellaneous configurations, and the padding bits to make sure the ip header ends on a multiple of 32 bits.

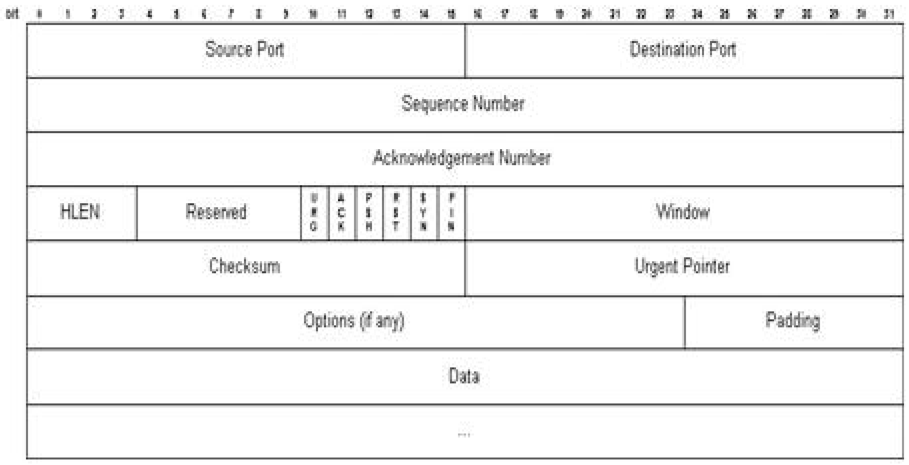
Exploiting the IP Identification Field

The IP identification field as mentioned above is used for fragmentation and reassembly of packets. Upon a receiver getting the packets it will use the id field to determine the order the packets should be reassembled. In the case that only a small amount of data is being transmitted the packet won’t be fragmented. The identification field will therefore not be used for anything. This opens up 16 bits to transmit secret information, which is enough to transmit 2 letters using their ascii equivalent.

Exploiting the TCP Handshake: Background

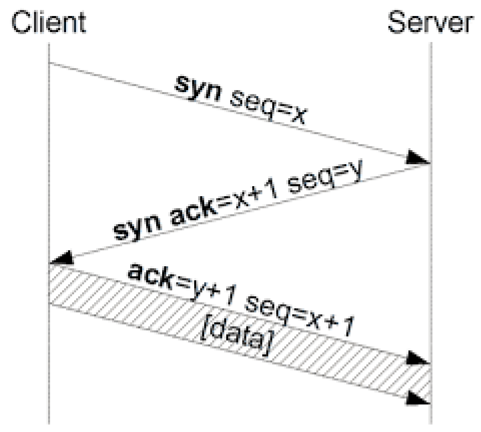
Another type of exploit comes from a vulnerability inherent in the way TCP is configured. TCP is a connection-oriented protocol in which a three way handshake is used to establish a connection. We will go over how a TCP packet is configured as well as how a TCP handshake works in order to understand these next 2 exploits.

Figure 2: TCP Header



The source port refers to the port the client is using, and the destination port refers to the port the server is listening on. The sequence number is a 32 bit field used to allow a client to confirm that their packet was received. The acknowledgment number is sent in response to a SYN packet and is the preceding sequence number incremented by 1. We will discuss these 2 fields more during the 3-way handshake description. The next 4 bits are the HLEN bits or the Data offset bits. They specify the length of the TCP header which is important in determining where the data begins and the header ends. The Reserved bits are 6 bits that are sent as 0 and reserved for future use. The next 6 bits are the Control bits and are set to tell the receiver the type of action to take. For example there is an ACK bit which is set for an Acknowledgment packet and a SYN bit which is set for a Synchronize packet. The window is used to determine the amount of data the receiver can respond with. The sender specifies how much data it is willing to accept in this field, and then the receiver knows that it can send up to that amount. The checksum field, similarly to in an IP packet, is used to protect against any sort of errors in the header fields that may have occurred during transmission. It is calculated using the entire TCP packet. The urgent pointer is used if the Urgent bit (one of the control bits) is set. This pointer holds the sequence number of the last urgent packet sent. The option field can vary in length and specifies other miscellaneous optional data that can be included. The padding bits are used to extend the option bits so that they are a multiple of 32 bits. The data bits is the actual payload that's being sent.

Figure 3: TCP 3-Way Handshake



The TCP 3-way handshake is how TCP establishes a connection. Recall that TCP is a connection-oriented protocol. The handshake starts with the client sending a SYN packet which has an initial sequence number stored in the sequence field. The server then responds with a SYN ACK packet. This packet has a second sequence number that the server created. The ACK field is filled with the Client's sequence number plus 1. This way the client knows that the server is responding to the initial SYN packet. The Client then sends the initial sequence number +1 (that it just received from the server) and an ACK packet which is the server's sequence number +1. The data can also be sent with this third exchange as the connection has been established.

Exploiting the TCP Handshake: The Initial Sequence Number

The initial sequence number is used to help the client and server synchronize their sequence numbers. The actual number is unimportant. This enables an attacker to be able to use this 32 bit field to transmit data undetected by the user. Instead of generating a random sequence number to be used initially, secret data can be sent as long as it is 32 bits or less.

Exploiting TCP: Forged Source Headers

This covert channel allows a user to send information to a machine while hiding who it is coming from. If I am machine A and I want to send information to C I can do it through a “bounce” server B. When a client makes a connection request to a server using TCP it sends a SYN packet as mentioned above. In this SYN packet it provides its IP address and port as the source port and source IP address. The server then responds with its SYN ACK packet to the provided source port and source IP address. However, if in the initial SYN packet I provide machine Cs source IP and port instead of my own then this SYN ACK will get sent to that machine instead of my own. If I combine this with the previous attack, then I can send information to Machine C but make it look like it came through Machine B. This can also be useful if trying to navigate through a firewall that only allows packets from trusted servers. The packet will appear as if it came from Machine B.

Difficulties of Covert Channels

There are many difficulties in implementing a covert channel. Intrinsic in any covert channel is its ability to go undetected. In order to do this, usually only small amounts of data can be sent at a time over the channel. The ratio of clear text data sent to covert data sent is usually very large which makes it a lot more difficult to send any significant amount of information over the channel. One way to improve this is to compress the data you send before sending it over the covert channel. This will enable you to send more data.

A second difficulty with using a covert channel is getting it set up. An attacker has to somehow have access to a machine in order to set up the packets to be sent as described above. This could be through some sort of malware that adjusts the packets before they are sent.

Another difficulty is preventing the secret information from being detected. Assuming everything is set up perfectly and the secret information is being sent, there is still a chance that someone could notice the covert channel. In order to do this you could encode the data so that it is less readily detected. The data need not be encrypted, as that can often extend the length of the data, but encoding will achieve the desired purpose. Let's say a system administrator is monitoring the traffic going out over a network using wireshark. If data is hidden in some of the header fields then as long as it isn't in plain text or very basically encoded (simple ASCII conversion) then it will most likely go undetected. There are little tricks that will also make it harder for a network administrator to detect the channel like spreading the covert message over multiple packets. You could simply shuffle the bits using an algorithm across a handful of messages so that any one message just appears to be random information. It would require an administrator to look at large groups of packets together to make any sense of the secret information.

When are They Used?

Covert channels are used in any situation in which someone wants to piggyback on an existing communication channel in order to send secret information. As mentioned earlier, there is the example of a King sending a message tattooed on a messenger's head. Another example would be 2 prisoners communicating to each other. A warden might allow them to send messages to each other in solitary, but he would censor these messages and read through them to make sure there is nothing suspicious being discussed. However, if they could send secret messages about how to escape then that would be a covert channel. In this situation, remaining undetected would be very important for their channel.

Another example of a covert channel is with an attacker who infiltrated a system. It is one thing to get access to a classified system, but it is an entirely different operation to get information from that system out. If an infiltrator got access to a system, and also had a way to secretly get the information out of that system then it could do this for years without the user being aware.

There is also a possibility that the attacker could be from within the company. For example, if I work for a big oil company, and I come across records that show proof of an illegal oil rigging operation then I might want to smuggle that information out. Sometimes it isn't as simple as looking at the information and emailing it to someone. I may want to slowly leak the information out over a normal looking channel. In this case I would have access to my machine and could therefore alter the header fields and inject my hidden information, and then send the packets as if they were normal packets.

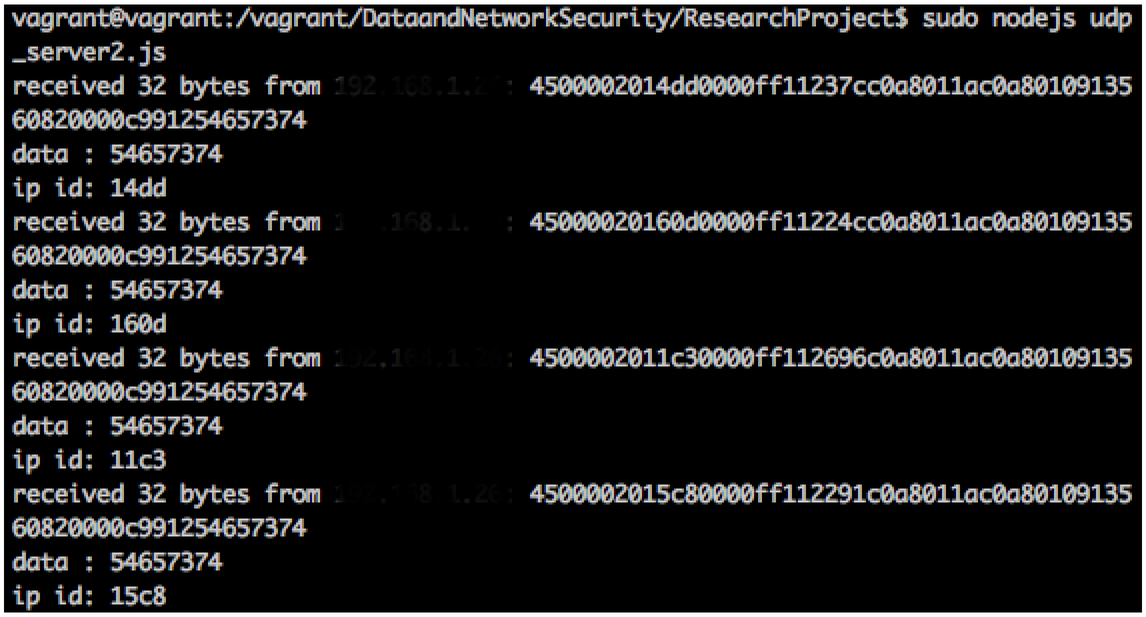
Exploiting the IP Identification Field: A Walkthrough

As mentioned above, the IP header has some fields that are not always used. The Identification field for example is not used if the packet is not fragmented. For my research I set up a covert channel using a UDP/IP Client-Server connection. Unlike TCP, UDP is connectionless. This simplifies things from a networking perspective, because there is no acknowledgement necessary. This makes setting up a covert channel easier. For example, if I were a spy in North Korea and was trying to smuggle information out then I could try to broadcast information out of Korea undetected without having to worry about a response. UDP uses IP, so we can still use the IP header fields to transmit covert data.

In this channel, I used a UDP client written in c and a UDP server written in node js. It is relatively simple to create a udp client-server setup. There are lots of examples of these already written in c, c++, python, javascript, etc. However, in order to be able to access the header fields for IP or UDP you need to use a raw-socket. Raw-sockets are more complicated as there are lots of ways in which your system or the network may reject your packet. With raw-socket you build out all of the IP header fields and all of the UDP header fields. The client program I mostly took from previous code written, and I altered the IPID field to have 2 letters of my secret message with each packet. The message I transmitted was "SAVE EGU|", with EGU| being the President ("Eagle"). In order to do this I had to transmit 4 packets, with "SA","VE","EG", and "U|" in packet 1,2,3 and 4 respectively. I converted these to hex and then they were converted again by my client, adding another layer between the plaintext and the actual transmitted IP

ID. I then received the packets with my node server and printed out the data in hex for the covert channel, as well as the actual packet data.

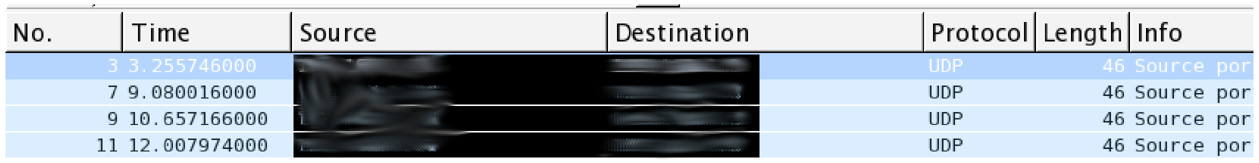
Figure 4: Received UDP Packets



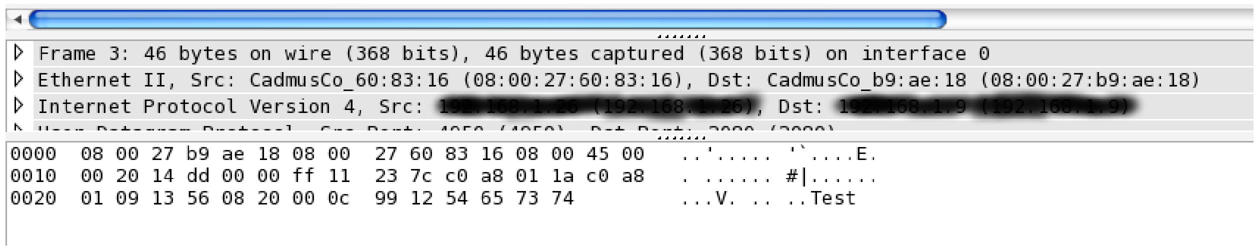
```
vagrant@vagrant:/vagrant/DataandNetworkSecurity/ResearchProject$ sudo nodejs udp
_server2.js
received 32 bytes from [redacted]: 4500002014dd0000ff11237cc0a8011ac0a80109135
60820000c991254657374
data : 54657374
ip id: 14dd
received 32 bytes from [redacted]: 45000020160d0000ff11224cc0a8011ac0a80109135
60820000c991254657374
data : 54657374
ip id: 160d
received 32 bytes from [redacted]: 4500002011c30000ff112696c0a8011ac0a80109135
60820000c991254657374
data : 54657374
ip id: 11c3
received 32 bytes from [redacted]: 4500002015c80000ff112291c0a8011ac0a80109135
60820000c991254657374
data : 54657374
ip id: 15c8
```

Above shows the received packet. You can see all of packet received as well as the data, and IP ID printed separately. I used a Kali Machine to transmit my data and an Ubuntu machine to receive it, all on a bridged network using Virtualbox. This way they had their own separate IP addresses. I also captured the packets using wireshark so that you can see that the network registered them as UDP packets and that they followed protocol. For the naked eye, they look like completely normal packets.

Figure 5: Captured UDP Packets



```
No.   Time          Source       Destination   Protocol Length Info
    3 3.255746000   [redacted]                 UDP          46 Source por
    7 9.080016000                              UDP          46 Source por
    9 10.657166000                             UDP          46 Source por
   11 12.007974000                             UDP          46 Source por
```



```
▷ Frame 3: 46 bytes on wire (368 bits), 46 bytes captured (368 bits) on interface 0
▷ Ethernet II, Src: CadmusCo_60:83:16 (08:00:27:60:83:16), Dst: CadmusCo_b9:ae:18 (08:00:27:b9:ae:18)
▷ Internet Protocol Version 4, Src: [redacted], Dst: [redacted]

0000  08 00 27 b9 ae 18 08 00  27 60 83 16 08 00 45 00   ..'..... '`....E.
0010  00 20 14 dd 00 00 ff 11  23 7c c0 a8 01 1a c0 a8   . ...... #|......
0020  01 09 13 56 08 20 00 0c  99 12 54 65 73 74         ...V. .. ..Test
```

From the above wireshark packets you can see the payload is Test, which is decoded and displayed in plain text by Wireshark. You can also see the IP Id field which I set manually to hold our covert data. Since we are looking at the first packet, the data I sent was 14dd. This comes from taking “S” and “A” and their hex representation (5341) and passing that to our client. Our client then encodes it in hex again which is 14dd. As you can see with this extra layer of conversion there is little to no chance someone would notice this with the naked eye.

As mentioned above, the biggest issue with transmitting data this way is how little data you are able to transmit with each packet. Also, if a machine was transmitting a lot of data at once then the IPID field would be needed for the server to reassemble the packets.

Conclusions

Transmitting Data over covert channels is non trivial. There are several things that have to be configured correctly to not raise suspicion, regarding how packets are formed, and what protocols your network uses. That being said, if a covert channel is in place, and hasn't been detected it is highly likely that it won't be. It is difficult to transmit data undetected with any reasonably acceptable packet size, but it is difficult to detect any data being transferred covertly when the secret data is sent in small bursts. This creates a trade off between the amount of data an attacker sends secretly, and the covertness of the channel. As the amount of data sent in secrecy increases, so does the chance of getting caught.

Appendix

UDP Client - udp4.c

This udp client code was copied and pasted from P.D. Buchan with the IP ID field, the IP address fields, and the ports adjusted.

```
/*  Copyright (C) 2011-2015  P.D. Buchan (pdbuchan@yahoo.com)

    This program is free software: you can redistribute it and/or modify
    it under the terms of the GNU General Public License as published by
    the Free Software Foundation, either version 3 of the License, or
    (at your option) any later version.

    This program is distributed in the hope that it will be useful,
    but WITHOUT ANY WARRANTY; without even the implied warranty of
    MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
    GNU General Public License for more details.

    You should have received a copy of the GNU General Public License
    along with this program.  If not, see <http://www.gnu.org/licenses/>.
*/

// Send an IPv4 UDP packet via raw socket.
// Stack fills out layer 2 (data link) information (MAC addresses) for us.
// Includes some UDP data.

#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>           // close()
#include <string.h>           // strcpy, memset(), and memcpy()

#include <netdb.h>            // struct addrinfo
#include <sys/types.h>        // needed for socket(), uint8_t, uint16_t,
uint32_t
#include <sys/socket.h>       // needed for socket()
#include <netinet/in.h>       // IPPROTO_RAW, IPPROTO_IP, IPPROTO_UDP,
INET_ADDRSTRLEN
#include <netinet/ip.h>       // struct ip and IP_MAXPACKET (which is 65535)
#include <netinet/udp.h>      // struct udphdr
#include <arpa/inet.h>        // inet_pton() and inet_ntop()
#include <sys/ioctl.h>        // macro ioctl is defined
#include <bits/ioctls.h>      // defines values for argument "request" of
ioctl.
#include <net/if.h>           // struct ifreq

#include <errno.h>            // errno, perror()

// Define some constants.
#define IP4_HDRLEN 20         // IPv4 header length
#define UDP_HDRLEN  8         // UDP header length, excludes data

// Function prototypes
uint16_t checksum (uint16_t *, int);
uint16_t udp4_checksum (struct ip, struct udphdr, uint8_t *, int);
char *allocate_strmem (int);
uint8_t *allocate_ustrmem (int);
int *allocate_intmem (int);

int
main (int argc, char **argv)
{
  int status, datalen, sd, *ip_flags;
  const int on = 1;
  char *interface, *target, *src_ip, *dst_ip;
  struct ip iphdr;
  struct udphdr udphdr;
  uint8_t *data, *packet;
  struct addrinfo hints, *res;
  struct sockaddr_in *ipv4, sin;
  struct ifreq ifr;
  void *tmp;
  // Allocate memory for various arrays.
  data = allocate_ustrmem (IP_MAXPACKET);
  packet = allocate_ustrmem (IP_MAXPACKET);
  interface = allocate_strmem (40);
  target = allocate_strmem (40);
  src_ip = allocate_strmem (INET_ADDRSTRLEN);
  dst_ip = allocate_strmem (INET_ADDRSTRLEN);
  ip_flags = allocate_intmem (4);

  // Interface to send packet through.
  strcpy (interface, "eth0");

  // Submit request for a socket descriptor to look up interface.
  if ((sd = socket (AF_INET, SOCK_RAW, IPPROTO_RAW)) < 0) {
    perror ("socket() failed to get socket descriptor for using ioctl() ");
    exit (EXIT_FAILURE);
  }

  // Use ioctl() to look up interface index which we will use to
  // bind socket descriptor sd to specified interface with setsockopt() since
  // none of the other arguments of sendto() specify which interface to use.
  memset (&ifr, 0, sizeof (ifr));
  snprintf (ifr.ifr_name, sizeof (ifr.ifr_name), "%s", interface);
  if (ioctl (sd, SIOCGIFINDEX, &ifr) < 0) {
    perror ("ioctl() failed to find interface ");
    return (EXIT_FAILURE);
  }
  close (sd);
  printf ("Index for interface %s is %i\n", interface, ifr.ifr_ifindex);

  // Source IPv4 address: you need to fill this out
  strcpy (src_ip, "111.11.11.11");

  // Destination URL or IPv4 address: you need to fill this out
  //
  strcpy (target, "111.11.1.1");

  // Fill out hints for getaddrinfo().
  memset (&hints, 0, sizeof (struct addrinfo));
  hints.ai_family = AF_INET;
  hints.ai_socktype = SOCK_STREAM;
  hints.ai_flags = hints.ai_flags | AI_CANONNAME;

  // Resolve target using getaddrinfo().
  if ((status = getaddrinfo (target, NULL, &hints, &res)) != 0) {
    fprintf (stderr, "getaddrinfo() failed: %s\n", gai_strerror (status));
    exit (EXIT_FAILURE);
  }
  ipv4 = (struct sockaddr_in *) res->ai_addr;
  tmp = &(ipv4->sin_addr);
  if (inet_ntop (AF_INET, tmp, dst_ip, INET_ADDRSTRLEN) == NULL) {
    status = errno;
    fprintf (stderr, "inet_ntop() failed.\nError message: %s", strerror
(status));
    exit (EXIT_FAILURE);
  }
  freeaddrinfo (res);

  // UDP data
  datalen = 4;
  data[0] = 'T';
  data[1] = 'e';
  data[2] = 's';
  data[3] = 't';

  // IPv4 header

  // IPv4 header length (4 bits): Number of 32-bit words in header = 5
  iphdr.ip_hl = IP4_HDRLEN / sizeof (uint32_t);

  // Internet Protocol version (4 bits): IPv4
  iphdr.ip_v = 4;

  // Type of service (8 bits)
  iphdr.ip_tos = 0;

  // Total length of datagram (16 bits): IP header + UDP header + datalen
  iphdr.ip_len = htons (IP4_HDRLEN + UDP_HDRLEN + datalen);

  // ID sequence number (16 bits): unused, since single datagram
  iphdr.ip_id = htons (5341);

  // Flags, and Fragmentation offset (3, 13 bits): 0 since single datagram

  // Zero (1 bit)
  ip_flags[0] = 0;

  // Do not fragment flag (1 bit)
  ip_flags[1] = 0;

  // More fragments following flag (1 bit)
  ip_flags[2] = 0;

  // Fragmentation offset (13 bits)
  ip_flags[3] = 0;
  iphdr.ip_off = htons ((ip_flags[0] << 15)
                      + (ip_flags[1] << 14)
                      + (ip_flags[2] << 13)
                      +  ip_flags[3]);

  // Time-to-Live (8 bits): default to maximum value
  iphdr.ip_ttl = 255;

  // Transport layer protocol (8 bits): 17 for UDP
  iphdr.ip_p = IPPROTO_UDP;

  // Source IPv4 address (32 bits)
  if ((status = inet_pton (AF_INET, src_ip, &(iphdr.ip_src))) != 1) {
    fprintf (stderr, "inet_pton() failed.\nError message: %s", strerror
(status));
    exit (EXIT_FAILURE);
  }

  // Destination IPv4 address (32 bits)
  if ((status = inet_pton (AF_INET, dst_ip, &(iphdr.ip_dst))) != 1) {
    fprintf (stderr, "inet_pton() failed.\nError message: %s", strerror
(status));
    exit (EXIT_FAILURE);
  }

  // IPv4 header checksum (16 bits): set to 0 when calculating checksum
  iphdr.ip_sum = 0;
  iphdr.ip_sum = checksum ((uint16_t *) &iphdr, IP4_HDRLEN);

  // UDP header

  // Source port number (16 bits): pick a number
  udphdr.source = htons (4950);

  // Destination port number (16 bits): pick a number
  udphdr.dest = htons (2080);

  // Length of UDP datagram (16 bits): UDP header + UDP data
  udphdr.len = htons (UDP_HDRLEN + datalen);

  // UDP checksum (16 bits)
  udphdr.check = udp4_checksum (iphdr, udphdr, data, datalen);

  // Prepare packet.

  // First part is an IPv4 header.
  memcpy (packet, &iphdr, IP4_HDRLEN * sizeof (uint8_t));
  // Next part of packet is upper layer protocol header.
  memcpy ((packet + IP4_HDRLEN), &udphdr, UDP_HDRLEN * sizeof (uint8_t));

  // Finally, add the UDP data.
  memcpy (packet + IP4_HDRLEN + UDP_HDRLEN, data, datalen * sizeof (uint8_t));

  // The kernel is going to prepare layer 2 information (ethernet frame header)
for us.
  // For that, we need to specify a destination for the kernel in order for it
  // to decide where to send the raw datagram. We fill in a struct in_addr with
  // the desired destination IP address, and pass this structure to the
sendto() function.
  memset (&sin, 0, sizeof (struct sockaddr_in));
  sin.sin_family = AF_INET;
  sin.sin_addr.s_addr = iphdr.ip_dst.s_addr;

  // Submit request for a raw socket descriptor.
  if ((sd = socket (AF_INET, SOCK_RAW, IPPROTO_RAW)) < 0) {
    perror ("socket() failed ");
    exit (EXIT_FAILURE);
  }

  // Set flag so socket expects us to provide IPv4 header.
  if (setsockopt (sd, IPPROTO_IP, IP_HDRINCL, &on, sizeof (on)) < 0) {
    perror ("setsockopt() failed to set IP_HDRINCL ");
    exit (EXIT_FAILURE);
  }

  // Bind socket to interface index.
  if (setsockopt (sd, SOL_SOCKET, SO_BINDTODEVICE, &ifr, sizeof (ifr)) < 0) {
    perror ("setsockopt() failed to bind to interface ");
    exit (EXIT_FAILURE);
  }

  // Send packet.
  if (sendto (sd, packet, IP4_HDRLEN + UDP_HDRLEN + datalen, 0, (struct
sockaddr *) &sin, sizeof (struct sockaddr)) < 0)  {
    perror ("sendto() failed ");
    exit (EXIT_FAILURE);
  }

  // Close socket descriptor.
  close (sd);

  // Free allocated memory.
  free (data);
  free (packet);
  free (interface);
  free (target);
  free (src_ip);
  free (dst_ip);
  free (ip_flags);

  return (EXIT_SUCCESS);
}

// Computing the internet checksum (RFC 1071).
// Note that the internet checksum does not preclude collisions.
uint16_t
checksum (uint16_t *addr, int len)
{
  int count = len;
  register uint32_t sum = 0;
  uint16_t answer = 0;

  // Sum up 2-byte values until none or only one byte left.
  while (count > 1) {
    sum += *(addr++);
    count -= 2;
  }

  // Add left-over byte, if any.
  if (count > 0) {
    sum += *(uint8_t *) addr;
  }

  // Fold 32-bit sum into 16 bits; we lose information by doing this,
  // increasing the chances of a collision.
  // sum = (lower 16 bits) + (upper 16 bits shifted right 16 bits)
  while (sum >> 16) {
    sum = (sum & 0xffff) + (sum >> 16);
  }

  // Checksum is one's compliment of sum.
  answer = ~sum;

  return (answer);
}

// Build IPv4 UDP pseudo-header and call checksum function.
uint16_t
udp4_checksum (struct ip iphdr, struct udphdr udphdr, uint8_t *payload, int
payloadlen)
{
  char buf[IP_MAXPACKET];
  char *ptr;
  int chksumlen = 0;
  int i;

  ptr = &buf[0];  // ptr points to beginning of buffer buf

  // Copy source IP address into buf (32 bits)
  memcpy (ptr, &iphdr.ip_src.s_addr, sizeof (iphdr.ip_src.s_addr));
  ptr += sizeof (iphdr.ip_src.s_addr);
  chksumlen += sizeof (iphdr.ip_src.s_addr);

  // Copy destination IP address into buf (32 bits)
  memcpy (ptr, &iphdr.ip_dst.s_addr, sizeof (iphdr.ip_dst.s_addr));
  ptr += sizeof (iphdr.ip_dst.s_addr);
  chksumlen += sizeof (iphdr.ip_dst.s_addr);

  // Copy zero field to buf (8 bits)
  *ptr = 0; ptr++;
  chksumlen += 1;

  // Copy transport layer protocol to buf (8 bits)
  memcpy (ptr, &iphdr.ip_p, sizeof (iphdr.ip_p));
  ptr += sizeof (iphdr.ip_p);
  chksumlen += sizeof (iphdr.ip_p);

  // Copy UDP length to buf (16 bits)
  memcpy (ptr, &udphdr.len, sizeof (udphdr.len));
  ptr += sizeof (udphdr.len);
  chksumlen += sizeof (udphdr.len);

  // Copy UDP source port to buf (16 bits)
  memcpy (ptr, &udphdr.source, sizeof (udphdr.source));
  ptr += sizeof (udphdr.source);
  chksumlen += sizeof (udphdr.source);

  // Copy UDP destination port to buf (16 bits)
  memcpy (ptr, &udphdr.dest, sizeof (udphdr.dest));
  ptr += sizeof (udphdr.dest);
  chksumlen += sizeof (udphdr.dest);

  // Copy UDP length again to buf (16 bits)
  memcpy (ptr, &udphdr.len, sizeof (udphdr.len));
  ptr += sizeof (udphdr.len);
  chksumlen += sizeof (udphdr.len);

  // Copy UDP checksum to buf (16 bits)
  // Zero, since we don't know it yet
  *ptr = 0; ptr++;
  *ptr = 0; ptr++;
  chksumlen += 2;

  // Copy payload to buf
  memcpy (ptr, payload, payloadlen);
  ptr += payloadlen;
  chksumlen += payloadlen;

  // Pad to the next 16-bit boundary
  for (i=0; i<payloadlen%2; i++, ptr++) {
    *ptr = 0;
    ptr++;
    chksumlen++;
  }

  return checksum ((uint16_t *) buf, chksumlen);
}

// Allocate memory for an array of chars.
char *
allocate_strmem (int len)
{
  void *tmp;

  if (len <= 0) {
    fprintf (stderr, "ERROR: Cannot allocate memory because len = %i in
allocate_strmem().\n", len);
    exit (EXIT_FAILURE);
  }

  tmp = (char *) malloc (len * sizeof (char));
  if (tmp != NULL) {
    memset (tmp, 0, len * sizeof (char));
    return (tmp);
  } else {
    fprintf (stderr, "ERROR: Cannot allocate memory for array
allocate_strmem().\n");
    exit (EXIT_FAILURE);
  }
}

// Allocate memory for an array of unsigned chars.
uint8_t *
allocate_ustrmem (int len)
{
  void *tmp;

  if (len <= 0) {
    fprintf (stderr, "ERROR: Cannot allocate memory because len = %i in
allocate_ustrmem().\n", len);
    exit (EXIT_FAILURE);
  }

  tmp = (uint8_t *) malloc (len * sizeof (uint8_t));
  if (tmp != NULL) {
    memset (tmp, 0, len * sizeof (uint8_t));
    return (tmp);
  } else {
    fprintf (stderr, "ERROR: Cannot allocate memory for array
allocate_ustrmem().\n");
    exit (EXIT_FAILURE);
  }
}

// Allocate memory for an array of ints.
int *
allocate_intmem (int len)
{
  void *tmp;

  if (len <= 0) {
    fprintf (stderr, "ERROR: Cannot allocate memory because len = %i in
allocate_intmem().\n", len);
    exit (EXIT_FAILURE);
  }

  tmp = (int *) malloc (len * sizeof (int));
  if (tmp != NULL) {
    memset (tmp, 0, len * sizeof (int));
    return (tmp);
  } else {
    fprintf (stderr, "ERROR: Cannot allocate memory for array
allocate_intmem().\n");
    exit (EXIT_FAILURE);
  }
}
```

UDP Server - udp_node_server.js

This next program is the server written in node js. It uses raw-sockets to accept a udp packet and print the payload, and the IPID field specifically, to the screen.

```
var PORT = 2080;
var HOST = '111.11.11.1';

//var dgram = require('dgram');
//var server = dgram.createSocket('udp4');
var raw = require ("raw-socket");
var server = raw.createSocket({protocol: raw.Protocol.UDP});

server.on("message",function(buffer,source){
      console.log("received " + buffer.length + " bytes from " + source + ": "
+ buffer.toString("hex"));
      var buffString = buffer.toString("hex");
      console.log("data : " + buffString.slice(56,buffString.length));
      console.log("ip id: " + buffString.slice(8,12));
});
```

Bibliography

"2.5. TCP Headers." *TCP Headers*. N.p., n.d. Web. 03 May 2015.

"A Discussion of Covert Channels and Steganography." *SANS*. SANS Institute InfoSec Reading Room, n.d. Web.

"Covert Channels in the TCP/IP Protocol Suite | Rowland | First Monday."*Covert Channels in the TCP/IP Protocol Suite | Rowland | First Monday*. N.p., n.d. Web. 03 May 2015.

"Explanation of the Three-Way Handshake via TCP/IP." *Explanation of the Three-Way Handshake via TCP/IP*. N.p., n.d. Web. 03 May 2015.

"InetDaemon.Com." *TCP 3-Way Handshake (SYN,SYN-ACK,ACK)*. N.p., n.d. Web. 03 May 2015.

"Internet Addressing and Routing First Step." *IP Header Format*. N.p., 27 Mar. 2014. Web. 03 May 2015.

"Intrusion Detection FAQ: What Is Covert Channel and What Are Some Examples?" *SANS:*. N.p., n.d. Web. 03 May 2015.

"IP, Internet Protocol." *IP, Internet Protocol*. Network Sorcery, n.d. Web. 03 May 2015.

Nichols, K. "Definition of the Differentiated Services Field (DS Field) in the IPv4 and IPv6 Headers." Cisco, n.d. Web. 03 May 2015.

"The TCP/IP Guide - TCP Message (Segment) Format." *The TCP/IP Guide - TCP Message (Segment) Format*. N.p., n.d. Web. 03 May 2015.

"What Is Covert Channel? - Definition from Techopedia." *Techopedias*. N.p., n.d. Web. 03 May 2015.