Adam Bates

300223031

Ethernet Packet Sniffer

Introduction

An Ethernet Packet Sniffer is a program used to capture packets being sent over a network. It is a very useful networking tool that can be used in debugging networks, testing application communication and used for such questionable activities like monitoring or spying on user traffic and reverse engineering proprietary protocols.

This report documents the procedures in Lab 1 in which an Ethernet packet sniffer program was developed using the C programming language. The requirements for the program were given as follows:

- The program should print out packets in a suitable format.
- The program must capture and identify at least the following
    - IPv4 traffic
        - TCP
        - UDP
        - ICMP
        - unknown
    - IPv6 traffic
        - IPv6 Extension headers
        - TCP
        - UDP
        - ICMPv6
        - unknown
    - other types of Ethernet traffic

As promiscuous mode was unavailable on the ECS Arch Linux machines, tcpdump was used to capture the packet data, capturing data was therefore excluded from the requirements listed.

The scope of this document is to provide in depth details on how the requirements were achieved and show understanding of internet packets and protocols.

Adam Bates

300223031

Design

The Ethernet packet sniffer made use of the libpcap framework which is a system independent interface to capture and identify packets. This includes all the C header files which contained the packet structures, header type values and the offline packet processing loop method. The packet processing method pulls out one Ethernet packet for each loop iteration so that it is possible to process what each packet contains.

The requirements stated that the internet protocol/network layer packet had to be determined and then if possible the transport layer protocol.

The IEEE 802.3 standard for Ethernet states that an Ethernet header should conform to the design in figure 1.

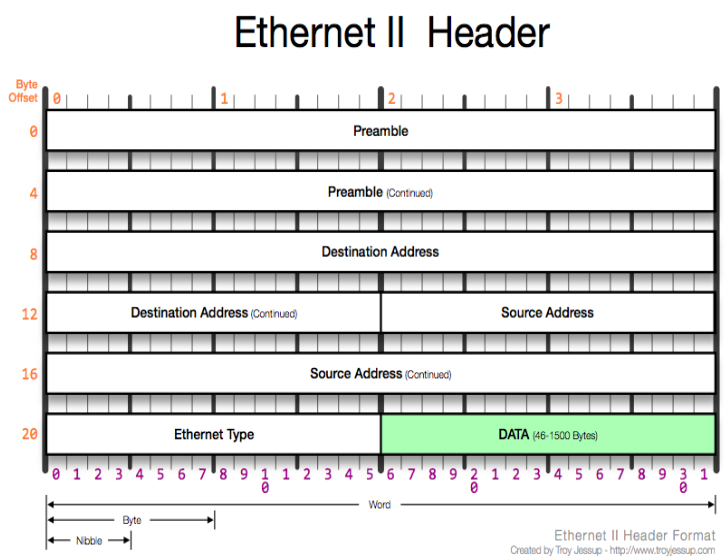


Figure 1: Ethernet II Header

As shown in this diagram it is possible to see that an Ethernet header contains an Ethernet type which is a 16 bit/2 byte value. To get this value we just have to reference it from the libpcap Ethernet packet structure. Once the Ethernet type value is retrieved, it is converted from network byte order to host byte order and the type is compared to the values for IPv4, IPv6 and the Address Resolution Protocol (ARP). All other values are being treated as unknown for this implementation.

Adam Bates

300223031

IPv4

If it has been determined that the header after the Ethernet header is an IPv4 header, the C structure must be retrieved. It is done by typecasting the location of the packet location plus the size of the Ethernet header to get the correct place in memory where the IP header is stored.

The IPv4 header shown in figure 2 as described in the Internet Engineering Task Force (IETF) publication RFC 791.

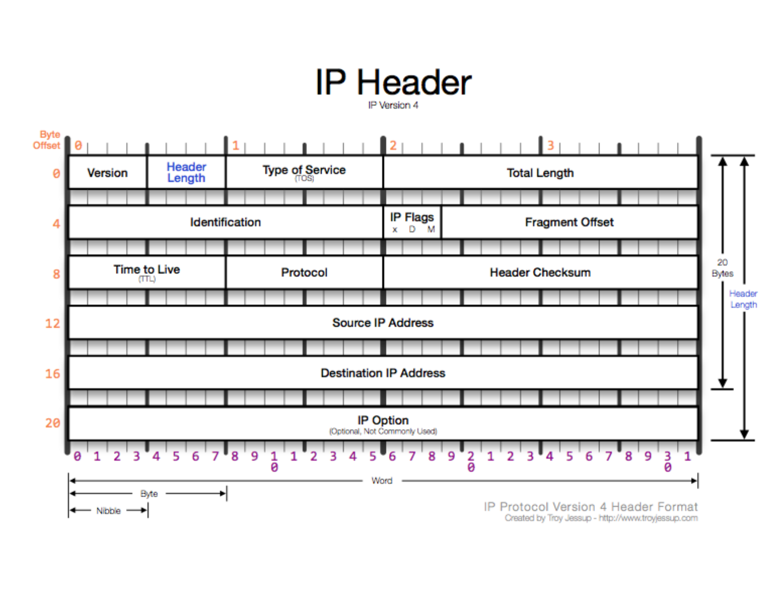


Figure 2: IPv4 Header

As seen in figure 2, the IPv4 header contains some useful information that the program could potentially print out. The packet sniffer implemented however only printed the source and destination address. When getting the source and destination address from the IPv4 header the C function inet_ntop had to be used to convert the 32 bit addresses into dotted decimal human readable format.

The next layer we have to find is the transport layer protocol. In IPv4 the header contains a value that defines what the transport layer protocol is. In the C code, the protocol can just be referenced from the libpcap IPv4 header structure. The protocol value is then compared to the values for TCP, UDP and ICMP which will determine what the next header will be.

Adam Bates

300223031

TCP & UDP

If the next value found to be TCP then the TCP C structure must be retrieved. If it is UDP then the UDP structure is retrieved. This process is similar to the getting the IP layer protocol except we also add the size of the last header.

The TCP header shown in figure 3 as described in the IETF publication RFC 793 and the UDP header in figure 4 as described in the IETF publication RFC 786.

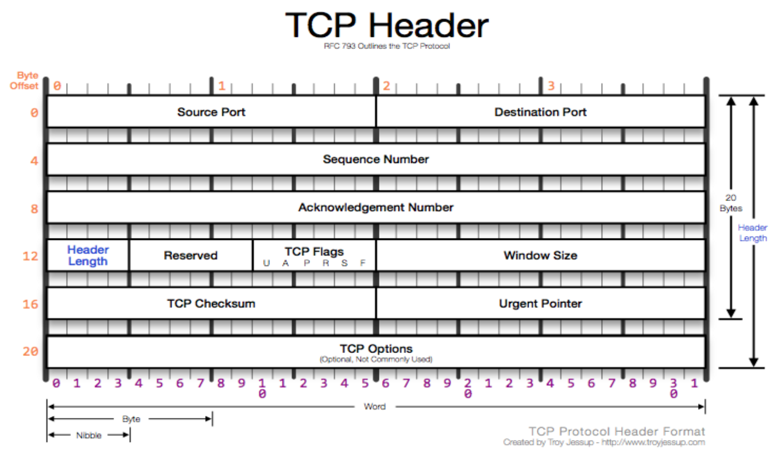


Figure 3: TCP Header

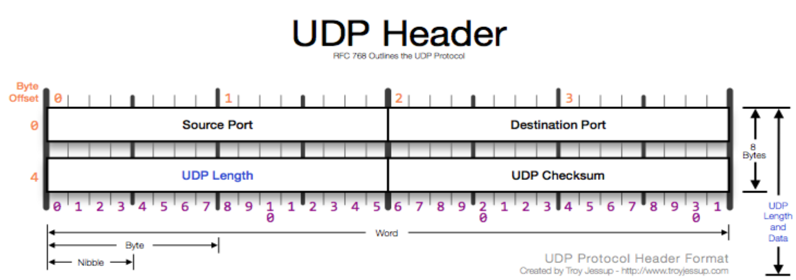


Figure 4: UDP Header

The source and destination ports, checksum, UDP and TCP payload were printed out. These values apart from the payload had to be converted to host byte order as explained in IPv4. To print the payload, the TCP and UDP header size had to be calculated so all data contained in the rest of the packet was considered to be payload. The data was casted to an unsigned char and the pointer passed to the PrintData function referenced from binarytides.com. This function converts any unreadable characters to human readable format and prints the data.

ICMP

The ICMP header shown in figure 5 as described in the IETF publication RFC 792.

The ICMP process is very similar to the TCP and UDP process except that it doesn’t contain as much information. As we can see from the ICMP header in figure 5 it we can get the type, code and checksum from the packet. We just get the values from the C libpcap structure and convert the values to host byte order and get the payload as everything after the header.

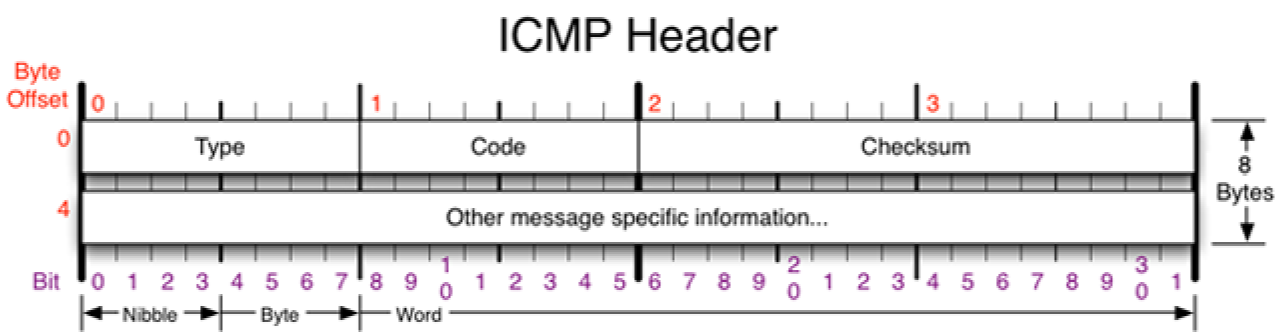


Figure 5: ICMP Header

Adam Bates

300223031

IPv6

Once we have determine that the header after the Ethernet header is an IPv6 header we can extract the information we need and determine the transport layer protocol contained inside.

The IPv6 header shown in figure 6 as described in the IETF publication RFC 2460.

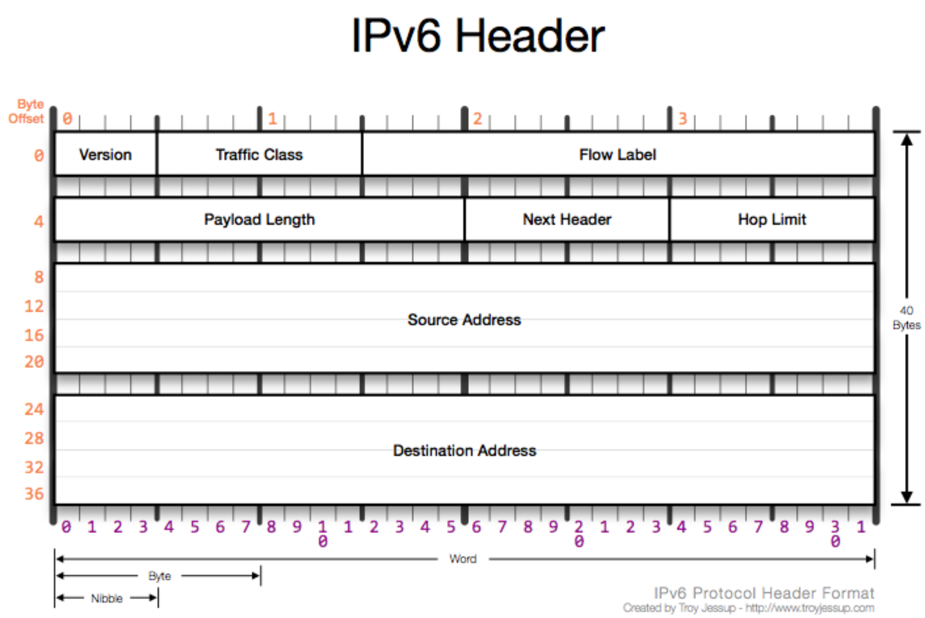


Figure 6: IPv6 Header

From looking at this header we can determine allot of information. In the application, the source and destination addresses were printed but the C function inet_ntop had to be used to convert the 128 bit addresses to a human readable hexadecimal format. IPv6 differs from IPv4 as it doesn't have to have a transport protocol straight after it. IPv6 can have any amount of extension headers after it, where the first ones type is given in the Next Header field shown in figure 6. Every extension header after this can have a next header field until the transport layer protocol is found.

A recursive function was used to determine each next header as the extension headers were recursed through. The function called IPV6_HANDLER would take the packet pointer, the point where we are up to in the packet (size of all the headers

so far), and the next header as arguments along with a string array to store values for printing tidily later on in the program. On the first function call the next header value from the IPv6 header would be passed as an argument so the function would determine what the next header is by matching the next headers value with the value for the next extension header or transport layer protocol.
Once determined, if the value is an extension header, the C structure of the header would be retrieved like previous headers and the next header value, size of all previous headers including the size of this extension header, packet and string will be passed to the same function for a recursive call.

If the next header value is a transport layer protocol then the same situation applies as the IPv4 transport layer protocols.

ARP

If the header after the Ethernet header is found to be an ARP header then we retrieve the ARP header structure in C. From the header in figure 7 which is defined by RFC 826 we can determine the opcode that tells us the operation of the ARP packet. Next the hardware type is used to then determine the Internet protocol type. In the program Ethernet 10/100Mbps was the hardware type implemented and then IPv4 was the Internet protocol implemented under it. Since we know the internet protocol is IPv4 the source and destination can now be printed as an IPv4 address.

<table>
<tr><td colspan="2">Hardware type</td><td>Protocol type</td></tr>
<tr><td>HW addr lth</td><td>P addr lth</td><td>Opcode</td></tr>
<tr><td colspan="3">Source hardware address</td></tr>
<tr><td colspan="3">Source protocol address</td></tr>
<tr><td colspan="3">Destination hardware address</td></tr>
<tr><td colspan="3">Destination protocol address</td></tr>
</table>

ARP message

Figure 7: ARP Header

Adam Bates

300223031

Apparatus and Procedures

The equipment and tools need to replicate this experiment is as follows:

- A computer running an operating system that supports:
  - libpcap or windows version called winpcap.
  - C compiler like GCC or MinGW for windows
  - Wireshark or Equivalent.
  - TCPDUMP or Equivalent.
- Experience in C programming and an understanding of Internet layers.

Results and Discussion

The results were verified by checking packets from the programs output with Wireshark a pre-existing packet sniffer application. All information printed matched the Wireshark output. This means that the Ethernet packet sniffer was functioning correctly. The results of the testing are displayed in Appendix 2.

Conclusions and recommendations.

The Ethernet packet sniffer was a success as it passed all the tests and met all the requirements of the lab. It shows understanding of how the libpcap library and Internet protocols work. Future work could contain more work on implementing other protocols and building on the ARP implementation as currently it only detects Ethernet IPv4 packets.

References

TCPDUMP & LIBPCAP (http://www.tcpdump.org)
  TCPDUMP and LIBPCAP are used under a 3-clause BSD license.

ARP IETF RFC - http://tools.ietf.org/html/rfc826

TCP IETF RFC - http://tools.ietf.org/html/rfc793

UDP IETF RFC - http://tools.ietf.org/html/rfc786

ETHERNET IEEE 802.3 - http://standards.ieee.org/about/get/802/802.3.html

IPV6 IETF RFC – http://tools.ietf.org/html/rfc2460

ICMPv6 IETF RFC - http://tools.ietf.org/html/rfc4443

ICMP IETF RFC - http://tools.ietf.org/html/rfc792

Figures from - http://www.troyjessup.com/headers/ provided as a public resource.

Appendices

Appendix A – Source Code

```
/* NWEN302 LAB 1
 * Name: Adam Bates
 * Usercode: batesadam
 * Student ID: 300223031
 */

// Libraries/Header Files to include
#include </usr/include/netinet/ip.h>
#include </usr/include/netinet/ip6.h>
#include </usr/include/pcap/pcap.h>
#include <stdio.h>
#include <stdlib.h>
#include <errno.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <arpa/inet.h>
#include <netinet/if_ether.h>
#include <net/ethernet.h>
#include <netinet/ether.h>
#include <netinet/tcp.h>
#include <netinet/udp.h>
#include <netinet/ip_icmp.h>
#include <netinet/icmp6.h>
#include <stdbool.h>
#include <string.h>

/* Function Prototypes */
void packetHandler(u_char *, const struct pcap_pkthdr*, const u_char*);
void IPV6_HANDLER(int, int, const u_char*, char*);
void Handle_TCP (const u_char*, int*);
void Handle_UDP (const u_char*, int*);
void PrintData (const u_char *, int Size);
void Handle_ARP(const u_char*, int*);
void printIPV4header(char*, char*);
void Handle_ICMPV6(const u_char*, int*);
void printIPV6Header();




/* Global Variables */
int packet_counter = 0, p = 0;
bool ARP_bool = false, IPV4_bool = false, IPV6_bool =false, ICMP_bool =
false;
bool TCP_bool = false; UDP_bool = false; UNKNOWN_bool = false;
UNKNWONPROTO_bool = false;

/* Declear IPv6 Source and Destination Address Variables */
char sourceIp6[INET6_ADDRSTRLEN];
char destIp6[INET6_ADDRSTRLEN];

int main(int argc, char *argv[]) {
```

```
    /* Filename of PCAP file as the first argument of the program
       if it is not included then inform the user and terminate program */
    if(argc == 1){
        printf("Please include pcap file as first argument\n");
        return;
    }

    const char* filename = argv[1];

    int i;
    for(i=2; i < argc; i++){
        if(strcasecmp("ARP", argv[i]) == 0)   // strcasecmp ignores case when
comparing strings
            ARP_bool = true;
        else if(strcasecmp("TCP", argv[i]) == 0)
            TCP_bool = true;
        else if(strcasecmp("UDP", argv[i]) == 0)
            UDP_bool = true;
        else if(strcasecmp("IPV4", argv[i]) == 0)
            IPV4_bool = true;
        else if(strcasecmp("IPV6", argv[i]) == 0)
            IPV6_bool = true;
        else if(strcasecmp("ICMP", argv[i]) == 0)
            ICMP_bool = true;
        else if(strcasecmp("UNKNOWN", argv[i]) == 0)
            UNKNOWN_bool = true;
    }

    if(argc == 2){  /* If there are no filters then enable all */
        ARP_bool = true; IPV4_bool = true; IPV6_bool = true; UNKNOWN_bool =
true;
    }

    if((IPV4_bool == true || IPV6_bool == true) && TCP_bool == false &&
UDP_bool == false && ICMP_bool == false && UNKNWONPROTO_bool == false){
        TCP_bool = true; UDP_bool = true; ICMP_bool = true; UNKNWONPROTO_bool
= true;
    }

    // Select a protocol filter
    char errbuf[PCAP_ERRBUF_SIZE];

    // Open capture file
    pcap_t *descr;
    descr = pcap_open_offline(filename, errbuf);
    if(descr == NULL){
        printf("Name of file was: %s\n", filename);
        printf("Error, : %s\n", errbuf);
    }

    // start packet processing loop, just like live capture
    if (pcap_loop(descr, 0, packetHandler, NULL) < 0) {
        //cout << "pcap_loop() failed: " << pcap_geterr(descr);
        printf("pcap_loop() failed\n");
        return 1;
    }
```

```
    else{
        printf("Loop not failed\n");
    }

    return 1;
}

void packetHandler(u_char *userData, const struct pcap_pkthdr* pkthdr, const
u_char* packet) {

    /* Link Layer - Declear Ethernet Header */
    const struct ether_header* ethernet_header;

    /* Declear IPv4 Headers */
    const struct ip* ip_header;
    const struct udphdr* udp_header;
    const struct icmphdr* icmp_header;

    /* Declear Source and Destination IPv4 Address Variables */
    char sourceIp[INET_ADDRSTRLEN];
    char destIp[INET_ADDRSTRLEN];

    /* Declear IPv6 Headers */
    const struct ip6_hdr* ip6_header;

    p = pkthdr->len;

    packet_counter++;  // Increment the packet counter

    /* Initialise Ethernet Structure */
    ethernet_header = (struct ether_header*)packet;

    int size = 0;
    size+=sizeof(struct ether_header);

    switch(ntohs(ethernet_header->ether_type)){
    case ETHERTYPE_IP:   // IPV4 Header

        if(IPV4_bool == false) return;

        ip_header = (struct ip*)(packet + size);
        inet_ntop(AF_INET, &(ip_header->ip_src), sourceIp, INET_ADDRSTRLEN);
        inet_ntop(AF_INET, &(ip_header->ip_dst), destIp, INET_ADDRSTRLEN);

        size+=sizeof(struct ip);

        u_char *data;
        int dataLength = 0;

        switch(ip_header->ip_p){
        case IPPROTO_TCP:  // Transmission Control Protocol (TCP)
            if(TCP_bool == false) return;
            printIPV4header(sourceIp, destIp);
            Handle_TCP(packet, &size);
            break;
        case IPPROTO_UDP:
            if(UDP_bool == false) return;
```

```
            printIPV4header(sourceIp, destIp);
            Handle_UDP(packet, &size);
          break;
        case IPPROTO_ICMP:  // Internet Control essgae Protocol (ICMP)

          if(ICMP_bool == false) return;
          printIPV4header(sourceIp, destIp);
          printf(" Protocol: ICMP\n");

          icmp_header = (struct icmphdr*)(packet + sizeof(struct
ether_header) +   sizeof(struct ip));

          u_int type = icmp_header->type;

          if(type == 11){
            printf(" TTL Expired\n");
          }
          else if(type == ICMP_ECHOREPLY){
            printf(" ICMP Echo Reply\n");
          }

          data = (u_char*)(packet + sizeof(struct ether_header) +
sizeof(struct ip) + sizeof(struct icmphdr));
          dataLength = pkthdr->len - (sizeof(struct ether_header) +
sizeof(struct ip) + sizeof(struct icmphdr));

          printf(" Code: %d\n", (unsigned int)(icmp_header->code));
          printf(" Checksum: %d\n", ntohs(icmp_header->checksum));
          printf(" Payload(%d bytes):\n", dataLength);

          PrintData(data, dataLength);

          break;
        default: // Unknown IPV4 Protocol
            if(UNKNWONPROTO_bool == false) return;
               printf(" Protocol: Unknown\n");
            break;
        }
        break;
        case ETHERTYPE_IPV6:  // IPV6

            if(IPV6_bool == false) return;

            ip6_header = (struct ip6_hdr*)(packet + size);

            inet_ntop(AF_INET6, &(ip6_header->ip6_src), sourceIp6,
INET6_ADDRSTRLEN);
            inet_ntop(AF_INET6, &(ip6_header->ip6_dst), destIp6,
INET6_ADDRSTRLEN);

            int nexthdr = ip6_header->ip6_nxt;

            size+=sizeof(struct ip6_hdr);

            char string[100] = " ";

            IPV6_HANDLER(nexthdr, size, packet, string);
```

Adam Bates

300223031

```
            break;
        case ETHERTYPE_ARP:   // ARP
            if(ARP_bool == false) return;
            Handle_ARP(packet, &size);
            break;
        default:
            if(UNKNOWN_bool == false) return;
            printf(" ETHER_TYPE: Unknown\n");
            break;
    }

}


/* Handle IPV6 Headers */
void IPV6_HANDLER(int hrd, int size, const u_char* packet, char* string){

    switch(hrd){
    case IPPROTO_ROUTING:  /* Routing Header */
        strcat(string, "ROUTING, ");
        struct ip6_rthdr* header = (struct ip6_rthdr*)(packet + size);
        size+=sizeof(struct ip6_rthdr);
        IPV6_HANDLER(header->ip6r_nxt, size, packet, string);
        break;
    case IPPROTO_HOPOPTS:  /* Hop-by-Hop options */
        strcat(string, "HOP-BY_HOP, ");
        struct ip6_hbh* header_hop = (struct ip6_hbh*)(packet + size);
        size+=sizeof(struct ip6_hbh);
        IPV6_HANDLER(header_hop->ip6h_nxt, size, packet, string);
        break;
    case IPPROTO_FRAGMENT: /* Fragmentation header(FRAGMENT) */
        strcat(string, "FRAGMENTATION, ");
        struct ip6_frag* header_frag = (struct ip6_frag*)(packet + size);
        size+=sizeof(struct ip6_frag);
        IPV6_HANDLER(header_frag->ip6f_nxt, size, packet, string);
        break;
    case IPPROTO_DSTOPTS:  /* Destination options(DSTOPTS) */
        strcat(string, "Destination options, ");
        struct ip6_dest* header_dest = (struct ip6_dest*)(packet + size);
        size+=sizeof(struct ip6_dest);
        IPV6_HANDLER(header_dest->ip6d_nxt, size, packet, string);
        break;
    case IPPROTO_TCP:      /* TCP PROTOCOL */
        if(TCP_bool == false) return;
        printIPV6Header();
        printf("%s\n", string);
        Handle_TCP (packet, &size);
        break;
    case IPPROTO_UDP:      /* UDP PROTOCOL */
        if(UDP_bool == false) return;
        printIPV6Header();
        printf("%s\n", string);
        Handle_UDP (packet, &size);
        break;
    case IPPROTO_ICMPV6:     /* ICMP6*/
        if(ICMP_bool == false) return;
```

```
        printIPV6Header();
        printf("%s\n", string);
        Handle_ICMPV6(packet, &size);
        break;
    default:
        if(UNKNWONPROTO_bool == false) return;
        printIPV6Header();
        printf("Unknown header(%d),", hrd);  /* Unknown Header */
        break;
    }
}

void printIPV6Header(){

printf("\n*****************************************************************
***********\n");
            printf("Packet Number: %d\n IP_Version: IPV6\n  Source IP: %s\n
Destination IP: %s\n Extension Headers:",packet_counter, sourceIp6, destIp6);
}

void Handle_ICMPV6(const u_char* packet, int* size){
        printf("\n");
        printf(" Protocol: ICMP\n");
        u_char *data;
        int dataLength = 0;

        struct icmp6_hdr* header_icmp6 = (struct icmp6_hdr*)(packet+*size);

        data = (u_char*)(packet + *size + sizeof(struct icmp6_hdr));
        dataLength = p - *size + sizeof(struct icmp6_hdr);

        printf(" Payload(%d bytes):\n", dataLength);

        PrintData(data, dataLength);
}

/* Handle ARP Headers */
void Handle_ARP(const u_char* packet, int* size){

    const struct ether_arp* arp_header;
    arp_header = (struct ether_arp*)(packet+*size);


printf("\n*****************************************************************
***********\n");
    printf("Packet Number: %d\n", packet_counter);

    /* Determine the ARP Operation Type */
    printf("ARP Operation: ");
    switch(ntohs(arp_header->arp_op)){
        case ARPOP_REQUEST:
            printf("ARP Request");
            break;
        case ARPOP_REPLY:
            printf("ARP Reply");
            break;
        case ARPOP_RREQUEST:
```

```
            printf("RARP Request");
            break;
        case ARPOP_RREPLY:
            printf("RARP RARP Reply");
            break;
        case ARPOP_InREQUEST:
            printf("InARP Request");
            break;
        case ARPOP_InREPLY:
            printf("InARP Request");
            break;
        case ARPOP_NAK:
            printf("(ATM)ARP NAK");
            break;
        default:
            printf("Unknown");
            break;
    }

    char sourceIp[INET_ADDRSTRLEN];
    char destIp[INET_ADDRSTRLEN];

    /* Determine the protocol hardware identifier */
    printf("\nProtocol Hardware Identifier: ");
    switch(ntohs(arp_header->arp_hrd)){
        case ARPHRD_NETROM:
            printf("From KA9Q: NET/ROM pseudo");
            break;
        case ARPHRD_IEEE1394:
            printf("IEEE 1394 IPv4 - RFC 2734");
            break;
        case ARPHRD_SLIP:
            printf("Serial Line Internet Protocol(SLIP)");
            break;
        case ARPHRD_ETHER:
            printf("Ethernet 10/100Mbps.");

            /* Determine the Protocol Type */
            printf("\nProtocol: ");
            int i;
            switch(ntohs(arp_header->arp_pro)){
                case ETHERTYPE_IP:
                    printf("IPv4\n");

                    printf("Sender MAC: ");

                    for(i=0; i<6;i++)
                        printf("%02X:", arp_header->arp_sha[i]);

                    printf("\nSender IP: ");

                     inet_ntop(AF_INET, &(arp_header->arp_spa), sourceIp,
INET_ADDRSTRLEN);
                     printf("%s", sourceIp);

                    printf("\nDestination MAC: ");
```

```
                    for(i=0; i<6;i++)
                        printf("%02X:", arp_header->arp_tha[i]);

                    printf("\nDestination IP: ");

                     inet_ntop(AF_INET, &(arp_header->arp_tpa), destIp,
INET_ADDRSTRLEN);
                     printf("%s", destIp);

                    printf("\n");

                break;
            default:
                printf("Unknown");
                break;
            }
            break;
        case ARPHRD_APPLETLK:
            printf("APPLEtalk");
            break;
        case ARPHRD_IEEE802:
            printf("IEEE 802.2 Ethernet/TR/TB");
            break;
        default:
            printf("Unknown");
            break;
    }
}
/* Function to handle TCP Headers */
void Handle_TCP (const u_char* packet, int* size){

            /* Initialise TCP header structure */
            const struct tcphdr* tcp_header;
            u_int sourcePort, destPort;
            u_char *data;

            tcp_header = (struct tcphdr*)(packet + *size);
            int dataLength = 0;

            /* Get the source and destination ports from the TCP header */
            sourcePort = ntohs(tcp_header->source);
            destPort = ntohs(tcp_header->dest);

            /* Initialise the data pointer to point to the data carryed by
the TCP and Initialise dataLength to the length of the data */
            *size+=tcp_header->doff*4;
            data = (u_char*)(packet + *size);
            dataLength = p - *size;

            /* Print the TCP header infomation and payload */
            printf(" Protocol: TCP\n    Source Port: %d\n   Destination Port:
%d\n Checksum: %d\n Payload(%d bytes):\n",
                    sourcePort, destPort, ntohs(tcp_header->check),
dataLength);

            /* Print the packet contents */
            PrintData (data , dataLength);
```

Adam Bates

300223031

```
}


/* Function to handle UDP Headers */
void Handle_UDP (const u_char* packet, int* size){

            /* Initialise UDP header structure */
            const struct udphdr* udp_header;
            u_int sourcePort, destPort;
            u_char *data;

            udp_header = (struct udphdr*)(packet + *size);
            int dataLength = 0;

            /* Get the source and destination ports from the UDP header */
            sourcePort = ntohs(udp_header->source);
            destPort = ntohs(udp_header->dest);

            /* Initialise the data pointer to point to the data carryed by
the UDP and Initialise dataLength to the length of the data */
            *size+=sizeof(struct udphdr);
            data = (u_char*)(packet + *size);
            dataLength = p - *size;

            /* Print the TCP header infomation and payload */
            printf(" Protocol: UDP\n    Source Port: %d\n   Destination Port:
%d\n Payload(%d bytes):\n",
                    sourcePort, destPort, dataLength);

            /* Print the packet contents */
            PrintData (data , dataLength);
}

void printIPV4header(char* source, char* dest){


printf("*********************************************************************
*********\n");
        printf("Packet number: %d\n", packet_counter);
        printf(" IP version: IPv4\n");

        printf("    Source IP: %s\n", source);
        printf("    Destination IP: %s\n", dest);
}

/* Convert and Print Data from protocols
 *  (PrintData Adapted from BinaryTides, "http://www.binarytides.com/packet-
sniffer-code-c-libpcap-linux-sockets/"
 *  written by Silver Moon)
 */
void PrintData (const u_char * data , int Size)
{
    int i , j;
    for(i=0 ; i < Size ; i++)
    {
        if( i!=0 && i%16==0)   //if one line of hex printing is complete...
        {
```

```
            printf("         ");
            for(j=i-16 ; j<i ; j++)
            {
                if(data[j]>=32 && data[j]<=128)
                    printf("%c",(unsigned char)data[j]); //if its a number or
alphabet

                else printf("."); //otherwise print a dot
            }
            printf("\n");
        }

        if(i%16==0) printf("   ");
        printf(" %02X",(unsigned int)data[i]);

        if( i==Size-1)  //print the last spaces
        {
            for(j=0;j<15-i%16;j++)
            {
                printf("   "); //extra spaces
            }

            printf("         ");

            for(j=i-i%16 ; j<=i ; j++)
            {
                if(data[j]>=32 && data[j]<=128)
                {
                    printf("%c",(unsigned char)data[j]);
                }
                else
                {
                    printf(".");
                }
            }
            printf("\n\n" );
        }
    }
}
```

Adam Bates

300223031

Appendix B – Comparison of results

IPv4 - UDP

Output:

```
**********************************************************************
Packet number: 57403
 IP version: IPv4
        Source IP: 130.195.6.17
        Destination IP: 130.195.4.179
 Protocol: UDP
        Source Port: 2049
        Destination Port: 1023
 Payload(24 bytes):
     00 09 53 12 00 00 00 01 00 00 00 00 00 00 00 00         ..S.............
     00 00 00 00 00 00 00 00                                 ........
```

Wireshark:

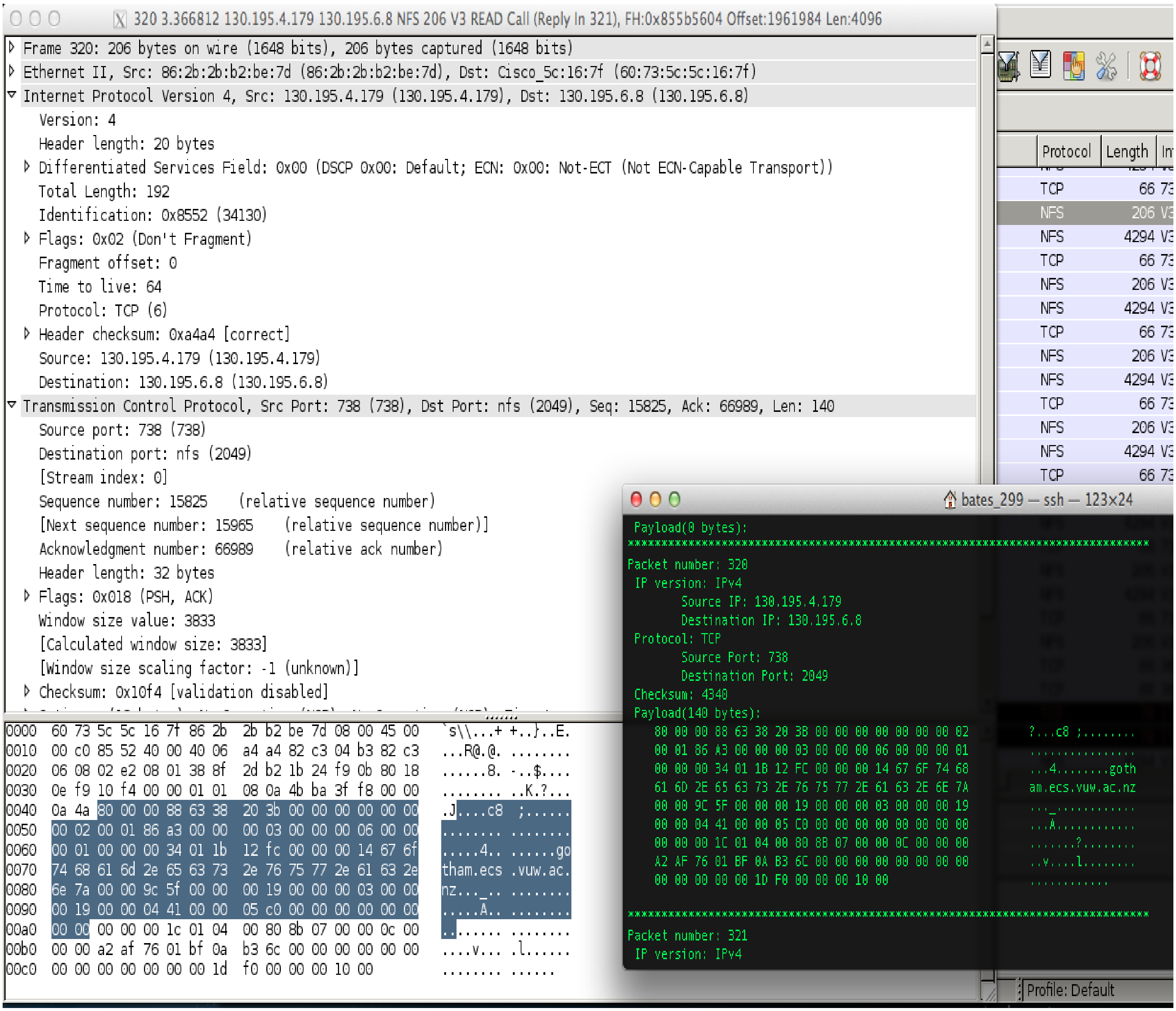
```
▿ Internet Protocol Version 4, Src: 130.195.6.17 (130.195.6.17), Dst: 130.195.4.179 (130.195.4.179)
    Version: 4
    Header length: 20 bytes
  ▷ Differentiated Services Field: 0x00 (DSCP 0x00: Default; ECN: 0x00: Not-ECT (Not ECN-Capable Transport))
    Total Length: 52
    Identification: 0x0000 (0)
  ▷ Flags: 0x00
    Fragment offset: 0
    Time to live: 63
    Protocol: UDP (17)
  ▷ Header checksum: 0x6b6f [correct]
    Source: 130.195.6.17 (130.195.6.17)
    Destination: 130.195.4.179 (130.195.4.179)
▿ User Datagram Protocol, Src Port: nfs (2049), Dst Port: 1023 (1023)
    Source port: nfs (2049)
    Destination port: 1023 (1023)
    Length: 32
  ▷ Checksum: 0x9047 [validation disabled]
▿ Remote Procedure Call, Type:Reply XID:0x00095312
    XID: 0x95312 (611090)
    Message Type: Reply (1)
    [Program: NFS (100003)]
    [Program Version: 3]
    [Procedure: NULL (0)]
    Reply State: accepted (0)
    [This is a reply to a request in frame 57402]
    [Time from request: 0.000169000 seconds]
  ▷ Verifier
    Accept State: RPC executed successfully (0)
▿ Network File System
    [Program Version: 3]
    [V3 Procedure: NULL (0)]

0010  00 34 00 00 00 00 3f 11  6b 6f 82 c3 06 11 82 c3   .4....?. ko......
0020  04 b3 08 01 03 ff 00 20  90 47 00 09 53 12 00 00   ........ .G..S...
0030  00 01 00 00 00 00 00 00  00 00 00 00 00 00 00 00   ........ ........
0040  00 00                                              ..
```

Adam Bates

300223031

IPv4- TCP



Adam Bates

300223031

IPv4 – ICMP

Output:

```
**********************************************************************
Packet number: 5649
 IP version: IPv4
        Source IP: 130.195.4.179
        Destination IP: 130.195.5.12
 Protocol: ICMP
 Code: 3
 Checksum: 3373
 Payload(237 bytes):
    45 00 00 ED 30 16 00 00 3F 11 3B A5 82 C3 05 0C         E...0...?.;.....
    82 C3 04 B3 00 35 4A 8C 00 D9 8A 1D 5D A2 81 90         .....5J.....]...
    00 01 00 01 00 04 00 05 03 67 6F 6F 02 67 6C 00         .........goo.gl.
    00 1C 00 01 C0 0C 00 1C 00 01 00 00 01 2C 00 10         .............,..
    24 04 68 00 40 06 08 03 00 00 00 00 00 00 10 08         $.h.@...........
    C0 0C 00 02 00 01 00 00 32 59 00 10 03 6E 73 33         ........2Y...ns3
    06 67 6F 6F 67 6C 65 03 63 6F 6D 00 C0 0C 00 02         .google.com.....
    00 01 00 00 32 59 00 06 03 6E 73 31 C0 44 C0 0C         ....2Y...ns1.D..
    00 02 00 01 00 00 32 59 00 06 03 6E 73 32 C0 44         ......2Y...ns2.D
    C0 0C 00 02 00 01 00 00 32 59 00 06 03 6E 73 34         ........2Y...ns4
    C0 44 C0 5C 00 01 00 01 00 05 28 41 00 04 D8 EF         .D.\......(A....
    20 0A C0 6E 00 01 00 01 00 05 28 41 00 04 D8 EF          ..n......(A....
    22 0A C0 40 00 01 00 01 00 05 28 41 00 04 D8 EF         "..@......(A....
    24 0A C0 80 00 01 00 01 00 05 28 41 00 04 D8 EF         $..?......(A....
    26 0A 00 00 29 10 00 00 00 80 00 00 00                  &...)....?...

**********************************************************************
```

Wireshark:

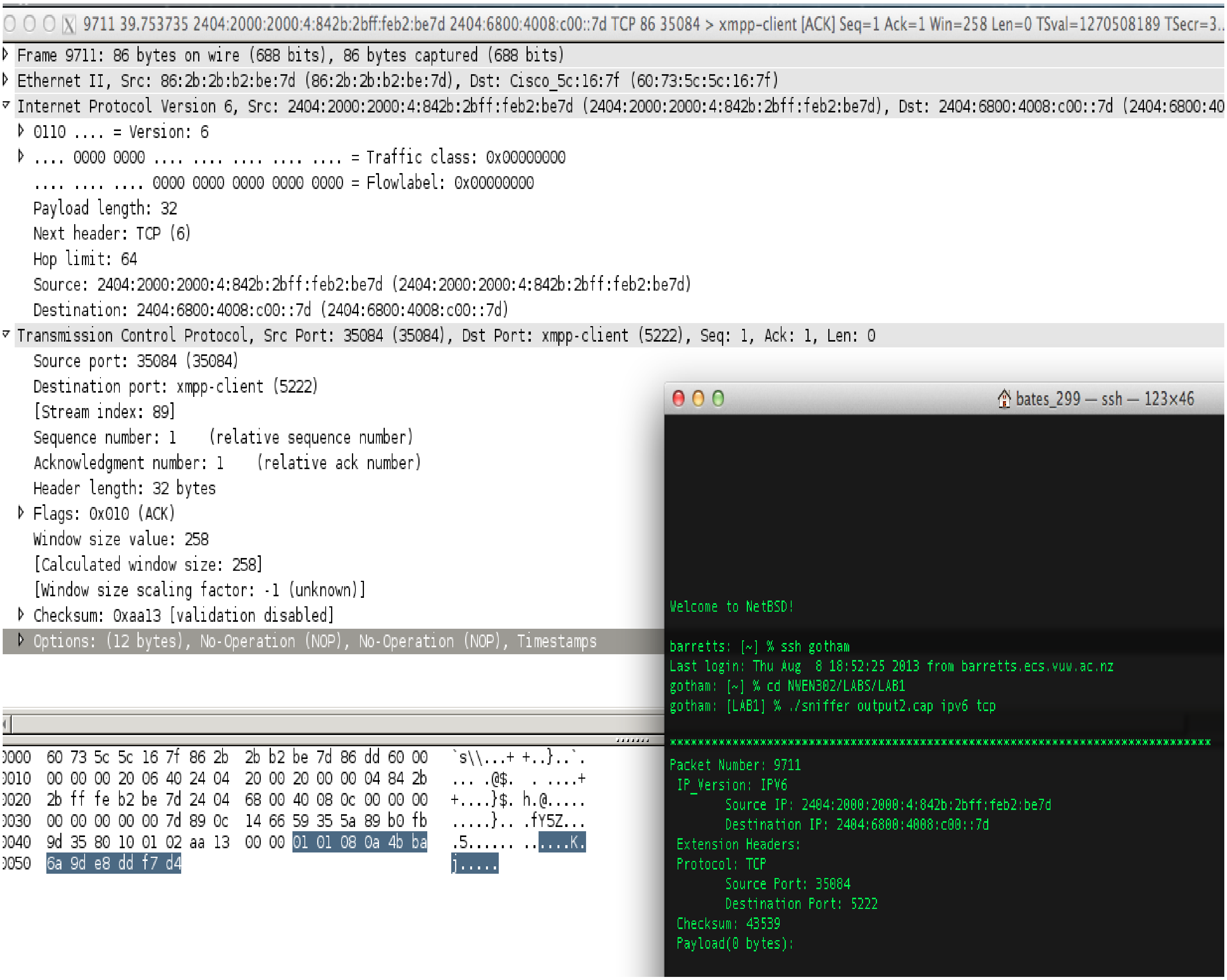
```
5649 36.105513 130.195.4.179 130.195.5.12 ICMP 27
    Version: 4
    Header length: 20 bytes
  ▽ Differentiated Services Field: 0xc0 (DSCP 0x30: Class Selector 6; ECN: 0x00: Not-ECT (Not ECN-Capable Transport))
      1100 00.. = Differentiated Services Codepoint: Class Selector 6 (0x30)
      .... ..00 = Explicit Congestion Notification: Not-ECT (Not ECN-Capable Transport) (0x00)
    Total Length: 265
    Identification: 0x98e2 (39138)
  ▷ Flags: 0x00
    Fragment offset: 0
    Time to live: 64
    Protocol: ICMP (1)
  ▷ Header checksum: 0xd10c [correct]
    Source: 130.195.4.179 (130.195.4.179)
    Destination: 130.195.5.12 (130.195.5.12)
▽ Internet Control Message Protocol
    Type: 3 (Destination unreachable)
    Code: 3 (Port unreachable)
    Checksum: 0x0d2d [correct]
  ▽ Internet Protocol Version 4, Src: 130.195.5.12 (130.195.5.12), Dst: 130.195.4.179 (130.195.4.179)
      Version: 4
      Header length: 20 bytes
    ▽ Differentiated Services Field: 0x00 (DSCP 0x00: Default; ECN: 0x00: Not-ECT (Not ECN-Capable Transport))
        0000 00.. = Differentiated Services Codepoint: Default (0x00)
        .... ..00 = Explicit Congestion Notification: Not-ECT (Not ECN-Capable Transport) (0x00)
      Total Length: 237
      Identification: 0x3016 (12310)
    ▷ Flags: 0x00
      Fragment offset: 0
      Time to live: 63
      Protocol: UDP (17)
    ▷ Header checksum: 0x3ba5 [correct]
      Source: 130.195.5.12 (130.195.5.12)
      Destination: 130.195.4.179 (130.195.4.179)
  ▷ User Datagram Protocol, Src Port: domain (53), Dst Port: 19084 (19084)
  ▽ Domain Name System (response)
      [Request In: 5597]
      [Time: 0.174311000 seconds]
      Transaction ID: 0x5da2
    ▷ Flags: 0x8190 Standard query response, No error
      Questions: 1
      Answer RRs: 1

0010  01 09 98 e2 00 00 40 01  d1 0c 82 c3 04 b3 82 c3   ......@. ........
0020  05 0c 03 03 0d 2d 00 00  00 00 45 00 00 ed 30 16   .....-.. ..E...0.
0030  00 00 3f 11 3b a5 82 c3  05 0c 82 c3 04 b3 00 35   ..?.;... .......5
0040  4a 8c 00 d9 8a 1d 5d a2  81 90 00 01 00 01 00 04   J.....]. ........
0050  00 05 03 67 6f 6f 02 67  6c 00 00 1c 00 01 c0 0c   ...goo.g l.......
0060  00 1c 00 01 00 00 01 2c  00 10 24 04 68 00 40 06   .......,  ..$.h.@.
0070  08 03 00 00 00 00 00 00  10 08 c0 0c 00 02 00 01   ........ ........
0080  00 00 32 59 00 10 03 6e  73 33 06 67 6f 6f 67 6c   ..2Y...n s3.googl
0090  65 03 63 6f 6d 00 c0 0c  00 02 00 01 00 00 32 59   e.com... ......2Y
00a0  00 06 03 6e 73 31 c0 44  c0 0c 00 02 00 01 00 00   ...ns1.D ........
00b0  32 59 00 06 03 6e 73 32  c0 44 c0 0c 00 02 00 01   2Y...ns2 .D......
00c0  00 00 32 59 00 06 03 6e  73 34 c0 44 c0 5c 00 01   ..2Y...n s4.D.\..
00d0  00 01 00 05 28 41 00 04  d8 ef 20 0a c0 6e 00 01   ....(A.. .. ..n..
00e0  00 01 00 05 28 41 00 04  d8 ef 22 0a c0 40 00 01   ....(A.. .."..@..
00f0  00 01 00 05 28 41 00 04  d8 ef 24 0a c0 80 00 01   ....(A.. ..$.....
0100  00 01 00 05 28 41 00 04  d8 ef 26 0a 00 00 29 10   ....(A.. ..&...).
0110  00 00 00 80 00 00 00                               .......
```

Adam Bates

300223031

<u>IPv6 - TCP</u>



Adam Bates

300223031

IPv6 - ICMP

Output:

```
**********************************************************************
Packet Number: 5090
 IP_Version: IPV6
        Source IP: fe80::842b:2bff:feb2:d2b5
        Destination IP: ff02::16
 Extension Headers: HOP-BY_HOP,

 Protocol: ICMP
 Payload(62 bytes):
    1A F4 00 00 00 02 02 00 00 00 FF 02 00 00 00 00          ................
    00 00 00 00 00 00 00 00 02 02 02 00 00 00 FF 02          ................
    00 00 00 00 00 00 00 00 00 01 FF B2 D2 B5 51 F0          ..............Q.
    8B EE 38 D0 55 6E 51 F0 8B EE 38 D0 55 6E                ..8.UnQ...8.Un


**********************************************************************
```

Wireshark:

```
    Type: IPv6 (0x86dd)
▿ Internet Protocol Version 6, Src: fe80::842b:2bff:feb2:d2b5 (fe80::842b:2bff:feb2:d2b5), Dst: ff02::16 (ff02::16)
  ▷ 0110 .... = Version: 6
  ▷ .... 0000 0000 .... .... .... .... .... = Traffic class: 0x00000000
    .... .... .... 0000 0000 0000 0000 0000 = Flowlabel: 0x00000000
    Payload length: 56
    Next header: IPv6 hop-by-hop option (0)
    Hop limit: 1
    Source: fe80::842b:2bff:feb2:d2b5 (fe80::842b:2bff:feb2:d2b5)
    Destination: ff02::16 (ff02::16)
  ▽ Hop-by-Hop Option
      Next header: ICMPv6 (58)
      Length: 0 (8 bytes)
    ▷ IPv6 Option (Router Alert)
    ▷ IPv6 Option (PadN)
▿ Internet Control Message Protocol v6
    Type: Multicast Listener Report Message v2 (143)
    Code: 0
    Checksum: 0x1af4 [correct]
    Reserved: 0000
    Number of Multicast Address Records: 2
  ▷ Multicast Address Record Exclude: ff02::202
  ▷ Multicast Address Record Exclude: ff02::1:ffb2:d2b5

0000  33 33 00 00 00 16 86 2b  2b b2 d2 b5 86 dd 60 00   33.....+ +.....`.
0010  00 00 00 38 00 01 fe 80  00 00 00 00 00 00 84 2b   ...8.... .......+
0020  2b ff fe b2 d2 b5 ff 02  00 00 00 00 00 00 00 00   +....... ........
0030  00 00 00 00 00 16 3a 00  05 02 00 00 01 00 8f 00   ......:. ........
0040  1a f4 00 00 00 02 02 00  00 00 ff 02 00 00 00 00   ........ ........
0050  00 00 00 00 00 00 00 00  02 02 02 00 00 00 ff 02   ........ ........
0060  00 00 00 00 00 00 00 00  00 01 ff b2 d2 b5         ........ ......
```

Adam Bates

300223031

ARP – Ethernet – IPv4

Output:

```
**************************************************************************************
Packet Number: 15967
ARP Operation: ARP Reply
Protocol Hardware Identifier: Ethernet 10/100Mbps.
Protocol: IPv4
Sender MAC: 60:73:5C:5C:16:7F:
Sender IP: 130.195.4.129
Destination MAC: 86:2B:2B:B2:D2:EB:
Destination IP: 130.195.4.171

**************************************************************************************
```

Wireshark:

```
▽ Frame 15967: 60 bytes on wire (480 bits), 60 bytes captured (480 bits)
    WTAP_ENCAP: 1
    Arrival Time: Jul 25, 2013 14:23:18.804577000 NZST
    [Time shift for this packet: 0.000000000 seconds]
    Epoch Time: 1374718998.804577000 seconds
    [Time delta from previous captured frame: 0.000796000 seconds]
    [Time delta from previous displayed frame: 0.000796000 seconds]
    [Time since reference or first frame: 46.653890000 seconds]
    Frame Number: 15967
    Frame Length: 60 bytes (480 bits)
    Capture Length: 60 bytes (480 bits)
    [Frame is marked: False]
    [Frame is ignored: False]
    [Protocols in frame: eth:arp]
    [Coloring Rule Name: ARP]
    [Coloring Rule String: arp]
▽ Ethernet II, Src: Cisco_5c:16:7f (60:73:5c:5c:16:7f), Dst: 86:2b:2b:b2:d2:eb (86:2b:2b:b2:d2:eb)
  ▽ Destination: 86:2b:2b:b2:d2:eb (86:2b:2b:b2:d2:eb)
      Address: 86:2b:2b:b2:d2:eb (86:2b:2b:b2:d2:eb)
      .... ..1. .... .... .... .... = LG bit: Locally administered address (this is NOT the factory default)
      .... ...0 .... .... .... .... = IG bit: Individual address (unicast)
  ▽ Source: Cisco_5c:16:7f (60:73:5c:5c:16:7f)
      Address: Cisco_5c:16:7f (60:73:5c:5c:16:7f)
      .... ..0. .... .... .... .... = LG bit: Globally unique address (factory default)
      .... ...0 .... .... .... .... = IG bit: Individual address (unicast)
    Type: ARP (0x0806)
    Padding: 000000000000000000000000000000000000
▽ Address Resolution Protocol (reply)
    Hardware type: Ethernet (1)
    Protocol type: IP (0x0800)
    Hardware size: 6
    Protocol size: 4
    Opcode: reply (2)
    Sender MAC address: Cisco_5c:16:7f (60:73:5c:5c:16:7f)
    Sender IP address: 130.195.4.129 (130.195.4.129)
    Target MAC address: 86:2b:2b:b2:d2:eb (86:2b:2b:b2:d2:eb)
    Target IP address: 130.195.4.171 (130.195.4.171)
```