Journal of Digital Forensics, Security and Law, Vol. 6(1)
(c)ADFSL 2011

Technology Corner: Internet Packet Sniffers

Nick V. Flor and Kenneth Guillory
Anderson School of Management
University of New Mexico
nickflor@unm.edu

1. SECTION EDITOR'S NOTE

Welcome to the Technology Section of the *Journal of Digital Forensics, Security, and Law*. The goal of this section of the journal is to explore the technical details of the various technologies used in digital forensics, security, and law—or conversely, to explore the technical details of the technologies used by hackers and other digital perpetrators. Since this column was initially proposed by me, I volunteered to start off with the initial article. However, I invite all readers to contribute articles to this section of *JDFSL*.

2. INTRODUCTION

The best way to understand an internet packet sniffer, hereafter "packet sniffer", is by analogy with a wiretap. A wiretap is a piece of hardware that allows a person to eavesdrop on phone conversations over a telephone network. Similarly, a packet sniffer is a piece of software that allows a person to eavesdrop on computer communications over the internet. A packet sniffer can be used as a diagnostic tool by network administrators or as a spying tool by hackers who can use it to steal passwords and other private information from computer users.

Whether you are a network administrator or information assurance specialist, it helps to have a detailed understanding of how packet sniffers work. And one of the best ways to acquire such an understanding is to build and modify an actual packet sniffer. But first, a disclaimer: the information contained in this paper is for educational purposes only—the use of packet sniffers to eavesdrop on private information is illegal, and violates the computer use policies of most organizations.

3. BACKGROUND: THE INTERNET AND TCP/IP

To build a packet sniffer, you need an understanding of how computers communicate over the Internet via the TCP/IP protocol, which is the subject of entire books and is beyond the scope of this brief technical article. Instead, I present a highly-simplified view; the reader interested in more details can refer to the many excellent books on these subjects (e.g., Dostalek and Kabelova, 2006; or Kozierok, 2005).

Suppose you have a computer in California that wants to send data to another computer in New York over the internet. For example, the computer in California might be a web server sending a web page to a user's computer in New York.

The computer sending the data is known as the *source* and the computer receiving the data is the *destination*. To send data over the internet, the source computer must first break the data into smaller pieces known as *packets.* The main benefit of breaking data into packets is *fault tolerance*. Namely, if a packet gets lost or corrupted during transmission, or if a part of the network goes down, only that one packet needs to be resent or rerouted—not the entire file. In short, the internet is a *packet switching network.*

However, for packet switching to work, each packet needs to be combined with additional information, much like a letter sent via mail needs to be placed inside an envelope with mailing addresses printed on the front. In particular, a packet needs at least a *source address* and a *destination address* (just like a mailed letter), in addition to a *sequence* number. The latter is necessary because it is possible for packets to arrive at the destination out of order, and the sequence numbers allows the destination computer to reassemble the packets for the original file in the proper order.

This additional information is placed in a *protocol header*. A packet sent out over the internet has two main protocol headers: (1) an IP header, which contains the source and destination addresses among other pieces of information; and (2) a TCP header, which contains the sequence number along with other information (see

Figure 1).

4. HOW TO WRITE A PACKET SNIFFER

From a design standpoint, a packet sniffer consists of two parts. The first part of the packet sniffer creates a *socket*, which you can think of as a “tap” into the network that the packet sniffer uses to eavesdrop on all communications packets either sent or received by computers on a local network. The second part of the packet sniffer consists of code that continuously listens for packets and extracts information from the packets. We will refer to these two parts of code as socket creation and eavesdropping, respectively. Before looking at the details of implementing the socket creation and eavesdropping code, we first look at the “boilerplate” code, which is common to all C# programs. Note: for those readers unfamiliar with programming in C# and Visual Studio (the development environment), Appendix A contains details on setting up the software and starting up a project.

4.1 Boilerplate Code

There are many excellent books on C# programming (see Blum, 2003; for a book specific to C# networking), so I will merely touch on some of the basics in this section.

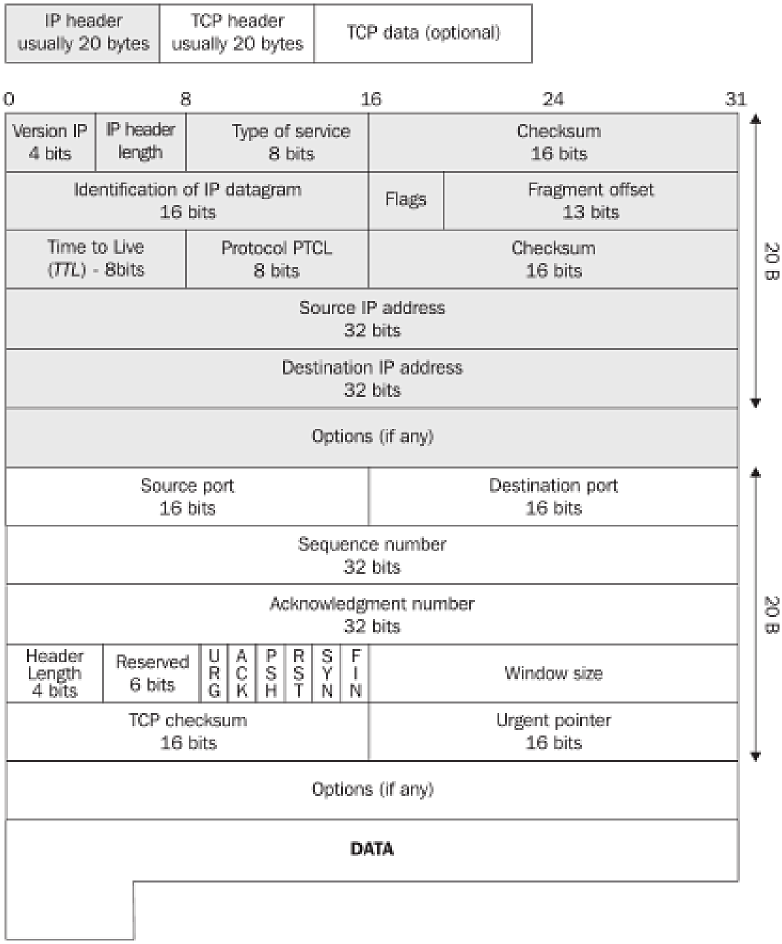


Figure 1. The data packet combined with the TCP & IP Headers (Dostalek and Kabelova, 2006; p. 245)

```
using System;

namespace JDFSLPacketSniffer
{
    public class packetsniffer
    {
        public static void Main()
        {
        }
    }
}
```

Figure 2. C# "boilerplate" code--code that is common to all C# programs.

All C# programs use one or more built-in code libraries, which are accessed with a “using” statement followed by the name of the library. In

Figure 2, this is the first line of code “using System;”. All C# programs must have a unique name, which is declared with the “namespace” keyword followed by a label, e.g., namespace JDFSLPacketSniffer. Finally, all C# programs must package code inside of an object designated by the “class” keyword followed by a label, e.g., class packetsniffer. The line “public static void Main()” is the entry point for code execution.

4.2 Socket Creation Code

Creating a socket requires using two libraries, “System.Net” and “System.Net.Sockets” (refer to

Figure 3 below). There are three steps required to create a socket, which I will summarize. The reader interested in a more in-depth discussion on the code particulars should refer to Blum (2003).

The first step in creating a socket is to get the address, specifically the *IP address*, of the computer running the packet sniffer. Because a computer can be connected to multiple networks—and therefore have multiple network addresses—it is necessary to loop through all possible network connections until an IP address is found.

Given the IP address of the computer that will run the packet sniffer, the next step is to create an *IP endpoint*—an object that combines the IP address with a *port*. To understand the concept behind an endpoint, note that a computer can be running multiple networked programs simultaneously. For example, you can be running a multi-player game like Guild Wars while simultaneously using a program like Skype to chat with a family member. Each networked program has a specific port associated with it, so that packets intended for one program do not get delivered to another program.

```
using System;
using System.Net;
using System.Net.Sockets;

namespace JDFSLPacketSniffer
{
    public class packetsniffer
    {
        public static void Main()
        {
            // Step 1. Get the IP Address
            IPHostEntry iphe;
            int i;
            iphe = Dns.GetHostEntry(Dns.GetHostName());
            for (i=0;i<iphe.AddressList.Length;i++)
            {
                if (iphe.AddressList[i].AddressFamily ==
AddressFamily.InterNetwork)
                    break; // iphe.AddressList[i] is the IP
Address
            }
            if (i == iphe.AddressList.Length) return; // Error:
No IP Address

            // Step 2. Create an Endpoint for Listening
            IPEndPoint ipep;
            ipep = new IPEndPoint(iphe.AddressList[i], 0);

            // Step 3.Create the socket
            Socket s;
            s = new Socket(AddressFamily.InterNetwork,
SocketType.Raw, ProtocolType.IP);
            s.Bind(ipep);
            s.SetSocketOption(SocketOptionLevel.IP,
SocketOptionName.HeaderIncluded, true);

            s.IOControl(IOControlCode.ReceiveAll,
BitConverter.GetBytes(1), null);


            }
        }
    }
}
```

Figure 3. The Socket Creation Code. A socket is best thought of as a “tap” where the packet sniffer can listen in on all communications between computer on the local network.

The third and final step is to actually create the socket. Creating a socket consists of specifying the type of network the socket will listen on. The type of network is necessary because, as mentioned, a computer can be connected to multiple networks simultaneously and one can create different sockets to listen in on the different types of networks. After creating the socket the endpoint—IP address and port—from step 1 must be bound to the socket. A socket normally listens only for packets specifically targeted for the endpoint, so the final line of code needed for socket creation places the socket in a mode to receive all packets. This is commonly known as “promiscuous mode” and is denoted by the line:

s.IOControl(IOControlCode.ReceiveAll, …);

```
// THE EAVESDROPPING CODE
int tcpoffset, dataoffset;
```

```
while (true)
{
    // Step 1. Read any incoming packet
    byte[] buf = new byte[s.ReceiveBufferSize];
    int count = s.Receive(buf);

    // Step 2. Extract (Parse) and print the IP Header
    string destaddr = buf[16] + "." + buf[17] + "." + buf[18] +
"." + buf[19];

    tcpoffset = (buf[0] & 0x0f) * 4;
    if (destaddr != "239.255.255.250")
    {
    Console.WriteLine("***** IP HEADER *****");
    Console.WriteLine("IP Version: {0}, Header Length: {1},
Packet Size: {2}, Id: {3}",
    (buf[0] >> 4), (buf[0] & 0x0f), (buf[2] << 8) + buf[3],
(buf[4] << 8) + buf[5]);
    Console.WriteLine("Source Address: {0}.{1}.{2}.{3}",
buf[12], buf[13], buf[14], buf[15]);
    Console.WriteLine("Destination Address: {0}.{1}.{2}.{3}",
buf[16], buf[17], buf[18], buf[19]);
    Console.WriteLine("**********");

    // Step 3. Parse and print the TCP Header
    Console.WriteLine("***** TCP HEADER *****");
    Console.WriteLine("Source Port: {0}", buf[tcpoffset] << 8 +
buf[tcpoffset+1]);
    Console.WriteLine("Destination Port: {0}", buf[tcpoffset +
2] << 8 + buf[tcpoffset + 3]);
    Console.WriteLine("Sequence Number: {0}", buf[tcpoffset + 4]
<< 24 + buf[tcpoffset + 5] << 16 + buf[tcpoffset + 6] << 8 +
buf[tcpoffset + 7]);

    dataoffset = (buf[tcpoffset + 12] >> 4)*4;
    Console.WriteLine("Data Offset: {0}", dataoffset);
    Console.WriteLine("**********");

    // Step 4. Parse the data
    Console.WriteLine("***** DATA *****");
    for (int b = (tcpoffset+dataoffset); b < count; b++)
    {
        if (Char.IsControl(Convert.ToChar(buf[b])))
            Console.Write(".");
        else
            Console.Write(Convert.ToChar(buf[b]));
    }
    Console.WriteLine();
    Console.WriteLine("**********");
}
```

Figure 4. The Eavesdropping Code. The code consists of four steps: (1) reading a packet; (2) parsing the IP header; (3) parsing the TCP header; and (4) printing out the data

4.3 Eavesdropping Code

The eavesdropping code (refer to

Figure 4) consists of four steps inside a loop. The first step, creates a buffer to hold the incoming data and then reads the packet into the buffer as an array of bytes. Recall from

Figure 1 that a packet consists of data combined with information necessary to deliver the data (IP header), and with information necessary to arrange the packets in the proper sequence (TCP header). The headers are placed before the data—the IP header first, followed by the TCP header.

The second step is to extract, or “parse”, and print the information in the IP header. One of the challenges in parsing the IP header information is that the data is stored in a byte buffer and some of the information can be contained in several bits within a single byte, while other information spans several bytes. Thus, C# *bit operations* are needed to reconstruct the information from the bits and the bytes. Examples of C# bit operations are AND (&), LEFT SHIFT (<<) and RIGHT SHIFT (>>). One of the first pieces of information to parse is the destination address (“destaddr” in

Figure 4). It is helpful to parse the destination address first, because some computers receive or transmit packets with high frequency, and you may want to filter out the printing of these packets. For instance, the example code filters out the destination IP address “239.255.255.250” because the volume of packets it transmits on the test computer drowned out the display of other packets.

The third step of the eavesdropping code is to parse the TCP header. To locate the start of the TCP header within the byte buffer, the code takes the IP header length and multiplies it by four. This line of code can be found in step 2 as “tcpoffset”, which is parsed right after the destination address. All parsing of TCP header information is done relative to this offset.

Similar to parsing the IP header, parsing the TCP header requires using bit operations to extract and print information that is either contained within the bits of a byte or that is spread out across multiple bytes. One important piece of information to extract is the offset of the packet data from the TCP header. In step 3, this is calculated as “dataoffset”. The tcpoffset when added to the dataoffset denotes the start of the data in the packet.

The fourth and final step is to print the data, which is done via a for-loop in the sample code. In the loop, code first checks if the byte is printable. If it is not printable, the code displays a “.”, otherwise the actual character is displayed. In a more sophisticated packet sniffer, the data would be further parsed and possibly stored in a file or database for later analysis. Appendix B contains the packet sniffer code in its entirety.

5. DISCUSSION: COUNTER MEASURES

There are several counter measures that network administrators and users can employ against packet sniffers (Ansari, Rajeev, S., and Chandrashekar, 2002). For example, a network administrator can use a program that sends packets out with invalid addresses. If a machine accepts the packet, this indicates the presence of a packet sniffer running on that machine. An example of a counter measure that users can employ is to encrypt their communications over the internet, e.g., by accessing web sites using the https protocol. While this measure still allows a hacker to sniff packets, the encrypted data is not easily viewed.

6. REFERENCES

Ansari, S., Rajeev, S., and Chandrashekar, H. (2002). Packet Sniffing: A Brief Introduction. *IEEE Potentials, 21,* 17-19.

Blum, R. (2003). *C# Network Programming.* San Francisco, CA: Sybex.

Dostalek, L., and Kabelova, A. (2006). *Understanding Tcp/ip: A Clear And Comprehensive Guide* (p. 245). Birmingham, UK: Packt Publising.

Kozierok, C. (2005). The TCP/IP Guide: A Comprehensive, Illustrated Internet Protocols Reference. San Francisco, CA: No Starch Press.

APPENDIX A: INSTALLING C# AND VISUAL C# 2010 EXPRESS

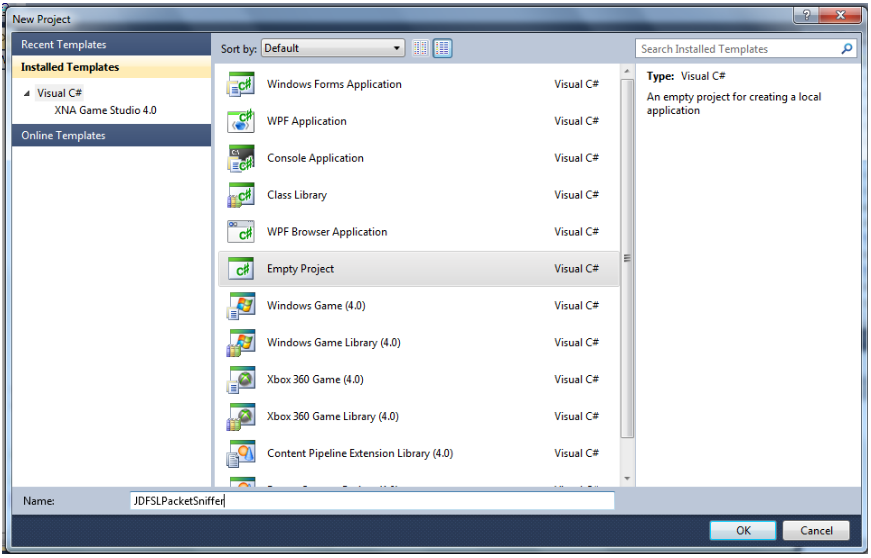
To run the code in this article, you must first download and install the free Visual C# 2010 Express software, which is located at:

http://www.microsoft.com/visualstudio/en-us/products/2010-editions/visual-csharp-express

After installing the software, run the program in administrator mode by right-clicking on the icon and selecting "Run as Administrator" in the pop-up dialog (not shown).

Select the menu item File > New Project; select Empty Project from the list; name your project "JDFSLPacketSniffer"; and finally click the "OK" button (see

Figure 5). This sets up an empty project.



New Project
Recent Templates
Installed Templates
Visual C#
XNA Game Studio 4.0
Online Templates
Search Installed Templates
Windows Forms Application	Visual C#
WPF Application	Visual C#
Console Application	Visual C#
Class Library	Visual C#
WPF Browser Application	Visual C#
Empty Project	Visual C#
Windows Game (4.0)	Visual C#
Windows Game Library (4.0)	Visual C#
Xbox 360 Game (4.0)	Visual C#
Xbox 360 Game Library (4.0)	Visual C#
Content Pipeline Extension Library (4.0)	Visual C#
OK
Cancel

Sort by: Default

Type: Visual C#

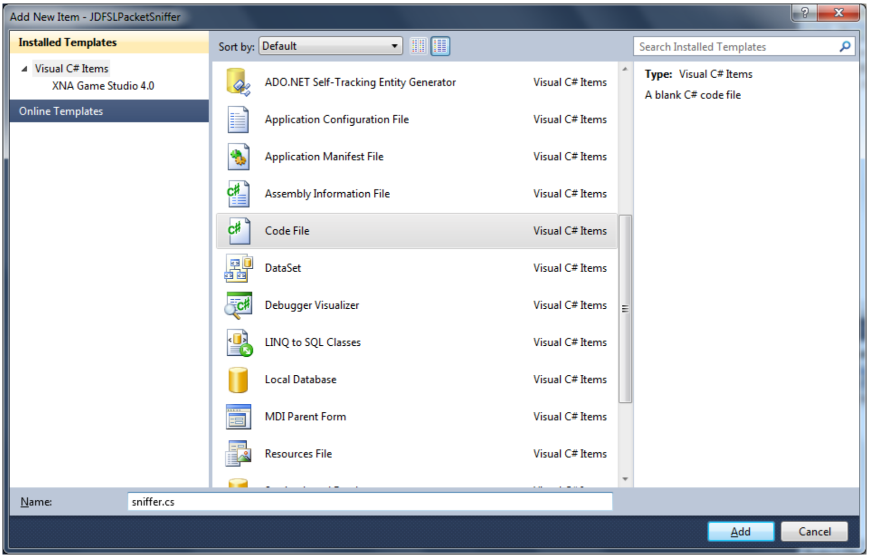
An empty project for creating a local application

Name: JDFSLPacketSniffer

Figure 5. New Project Dialog Box. Select "Empty Project" and name your project JDFSLPacketSniffer

Next you need to add at least one code file to this project. Select the menu item Project > Add New Item …, which will bring up the dialog box in

Figure 6.



Add New Item - JDFSLPacketSniffer

Installed Templates

- Visual C# Items
  - XNA Game Studio 4.0

Online Templates

Sort by: Default

ADO.NET Self-Tracking Entity Generator	Visual C# Items
Application Configuration File	Visual C# Items
Application Manifest File	Visual C# Items
Assembly Information File	Visual C# Items
Code File	Visual C# Items
DataSet	Visual C# Items
Debugger Visualizer	Visual C# Items
LINQ to SQL Classes	Visual C# Items
Local Database	Visual C# Items
MDI Parent Form	Visual C# Items
Resources File	Visual C# Items

Type: Visual C# Items

A blank C# code file

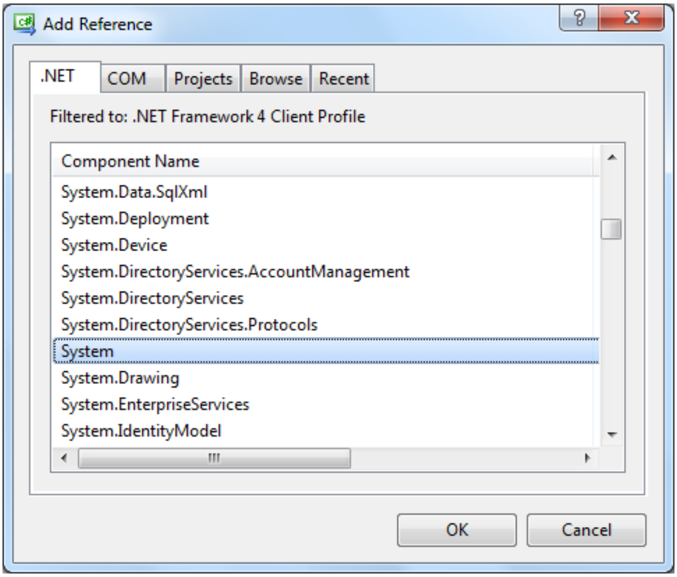
Name: sniffer.cs

Figure 6. New Item Dialog Box. Select "Code File" and name your project “sniffer.cs”

Select “Code File” from the list, and name your file “sniffer.cs”. Click the “Add” button when you are finished.

Finally, select the menu item Project > Add Reference…, which will bring up the dialog box in

Figure 7. Select the .NET tab, then find and select “System” in the list box. When you are done, click the “OK” button. You are now ready to add and run the code in this article (see Appendix B for the entire packet sniffer code).



Filtered to: .NET Framework 4 Client Profile

Component Name

System.Data.SqlXml
System.Deployment
System.Device
System.DirectoryServices.AccountManagement
System.DirectoryServices
System.DirectoryServices.Protocols
System
System.Drawing
System.EnterpriseServices
System.IdentityModel

Figure 7. Reference Dialog Box. Select the .NET tab, then find and select the System library.

APPENDIX B: THE ENTIRE PACKET SNIFFER CODE

```
using System;
using System.Net; // ipep,
using System.Net.Sockets; // s,

namespace JDFSLPacketSniffer
{
    public class packetsniffer
    {
        public static void Main()
        {
            // PART I. THE SOCKET CREATION CODE
            // Step 1. Get the IP Address
            IPHostEntry iphe;
            int i;

            iphe = Dns.GetHostEntry(Dns.GetHostName());
            for (i=0;i<iphe.AddressList.Length;i++)
            {
                if (iphe.AddressList[i].AddressFamily ==
                    AddressFamily.InterNetwork)
                    break; // iphe.AddressList[i] is the IP
Address
            }
            if (i == iphe.AddressList.Length) return; // Error:
No IP Address

            // Step 2. Create an Endpoint for listening
            IPEndPoint ipep;
            ipep = new IPEndPoint(iphe.AddressList[i], 0);

            // Step 3. Create the Socket
            Socket s;
            s = new Socket(AddressFamily.InterNetwork,
SocketType.Raw, ProtocolType.IP);
            s.Bind(ipep);
            s.SetSocketOption(SocketOptionLevel.IP,
SocketOptionName.HeaderIncluded, true);

            s.IOControl(IOControlCode.ReceiveAll,
BitConverter.GetBytes(1), null);

            // PART II. THE EAVESDROPPING CODE

            int tcpoffset, dataoffset;
            while (true)
            {
                // Step 1. Read any incoming packet
                byte[] buf = new byte[s.ReceiveBufferSize];
                int count = s.Receive(buf);

                // Step 2. Extract (Parse) and print the IP
Header
```

```
                string destaddr = buf[16] + "." + buf[17] + "."
+ buf[18] + "." + buf[19];

                tcpoffset = (buf[0] & 0x0f) * 4;
                if (destaddr != "239.255.255.250")
                {
                    Console.WriteLine("***** IP HEADER *****");
                    Console.WriteLine("IP Version: {0}, Header
Length: {1}, Packet Size: {2}, Id: {3}",
                    (buf[0] >> 4), (buf[0] & 0x0f), (buf[2] <<
8) + buf[3], (buf[4] << 8) + buf[5]);
                    Console.WriteLine("Source Address:
{0}.{1}.{2}.{3}", buf[12], buf[13], buf[14], buf[15]);
                    Console.WriteLine("Destination Address:
{0}.{1}.{2}.{3}", buf[16], buf[17], buf[18], buf[19]);
                    Console.WriteLine("**********");

                    // Step 3. Parse and print the TCP Header
                    Console.WriteLine("***** TCP HEADER *****");
                    Console.WriteLine("Source Port: {0}",
buf[tcpoffset] << 8 + buf[tcpoffset+1]);
                    Console.WriteLine("Destination Port: {0}",
buf[tcpoffset + 2] << 8 + buf[tcpoffset + 3]);
                    Console.WriteLine("Sequence Number: {0}",
buf[tcpoffset + 4] << 24 + buf[tcpoffset + 5] << 16 +
buf[tcpoffset + 6] << 8 + buf[tcpoffset + 7]);
                    dataoffset = (buf[tcpoffset + 12] >> 4)*4;
                    Console.WriteLine("Data Offset: {0}",
dataoffset);
                    Console.WriteLine("**********");

                    // Step 4. Parse and print the data
                    Console.WriteLine("***** DATA *****");
                    for (int b = (tcpoffset+dataoffset); b <
count; b++)
                    {
                        if
(Char.IsControl(Convert.ToChar(buf[b])))
                            Console.Write(".");
                        else

Console.Write(Convert.ToChar(buf[b]));
                    }
                    Console.WriteLine();
                    Console.WriteLine("**********");
                }
            }
        }
    }
}
```