### Resources

* [**Case Studies**](http://www.calsoftlabs.com/resources/default.html)
* [**Whitepapers**](http://www.calsoftlabs.com/resources/whitepapers.html)
* [**Brochures**](http://www.calsoftlabs.com/resources/brochures.html)

**Inside Whitepaper**

* [Introduction](http://www.calsoftlabs.com/whitepapers/website-blocking.html#01)
* [An Overview of how to capture all the packets that travel in the network](http://www.calsoftlabs.com/whitepapers/website-blocking.html#02)
* [How does a TCP connection occur?](http://www.calsoftlabs.com/whitepapers/website-blocking.html#03)
* [Establishing a TCP connection](http://www.calsoftlabs.com/whitepapers/website-blocking.html#04)
* [TCP connection reset](http://www.calsoftlabs.com/whitepapers/website-blocking.html#05)
* [Breaking a TCP connection (blocking a particular workstation)](http://www.calsoftlabs.com/whitepapers/website-blocking.html#06)
* [A Connection-Terminator Application](http://www.calsoftlabs.com/whitepapers/website-blocking.html#07)
* [A Case study on Rule-based control](http://www.calsoftlabs.com/whitepapers/website-blocking.html#08)
* [Summary](http://www.calsoftlabs.com/whitepapers/website-blocking.html#09)
* [Home](http://www.calsoftlabs.com/default.html) »
* [Whitepapers](http://www.calsoftlabs.com/resources/whitepapers.html) »
* [WebSite Blocking & TCP IP Connection](http://www.calsoftlabs.com/whitepapers/website-blocking.html)
* Blocking TCP requests on an Ethernet Network for Bandwidth Control

## Introduction

People have from time to time been eager to see what is going on in a network. Network administrators and higher level people have always been curious to know how their network resources are utilized. There are companies that want their professionals to use the Internet to increase their productivity, and these companies look at doing it by blocking some sites that are obviously very unproductive. Moreover people may also not like their employees visiting pornographic and other sites that are absolutely of no advantage to the company. Here we provide you with a paper that aids programmers to develop applications that can monitor a network tell which user accesses which site and also restrict the access of some users and sites that are of no advantage.

[[http://www.calsoftlabs.com/images/icon_top.gif](http://www.calsoftlabs.com/whitepapers/website-blocking.html#top)](http://www.calsoftlabs.com/whitepapers/website-blocking.html#top)

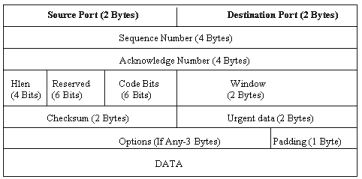
## An Overview of how to capture all the packets that travel in the network

The previous paper titled - Monitoring Ethernet Networks with NDIS drivers clearly describes how we can capture all the packets that travel in the network and extract the URL that is contained in these packets. Briefly describing, if the Ethernet card of your computer is placed in a mode called the promiscuous mode then it accepts all the packets that come to it. An Ethernet network is said to be a broadcasting network in the sense that it broadcasts all the packets that travel in the network. More clearly, every machine in the network will receive a copy of the packet that travels in the network. We use a driver and place the Ethernet card in the so-called promiscuous mode. Then we examine the header and the data of the captured packets to extract the URL contained in them. With these URLs that are obtained it is possible to find out the site the particular user has visited.

[[http://www.calsoftlabs.com/images/icon_top.gif](http://www.calsoftlabs.com/whitepapers/website-blocking.html#top)](http://www.calsoftlabs.com/whitepapers/website-blocking.html#top)

## How does a TCP communication occur?

Inorder to capture a print-job, we do not write one single executable, but make a series of changes to the printing subsystem. We develop the following for this To understand this we should first of all know the various fields that are present in the TCP header.



The important fields in the TCP header are

* Source Port (2 Bytes)
* Destination Port (2 Bytes)
* Sequence Number (4 Bytes)
* Acknowledge Number (4 Bytes)
* Hlen, Reserved and Code Bits (2 Bytes)
* Window (2 Bytes)
* Checksum (2 Bytes)
* Urgent Pointer (2 Bytes)

The various fields that are of interest to us are elaborated below:

**Source Port:** This field specifies the port from which the request originated. The source port value can be used to find out whether the packet is a HTTP packet, FTP packet etc.

**Destination Port:** This field specifies the port to which the packet is destined. The destination port value can be used to find out whether the packet is a HTTP response, FTP response etc.

**Sequence Number:** The sequence number identifies the position of the packet in the sender's byte stream.

**Acknowledgment Number:** The acknowledgment number field identifies the number of the octet that the source expects to receive next.

**Header Length:** Since TCP header includes the variable length options and padding fields, the length of the TCP header is not fixed. This field specifies the length of the header. This is a 4-bit field.

**Reserved Bits:** This is a 6-Bit field and is reserved for future use.

**Code Bits:** The code bits totally comprise 6 bits and each has its own significance.

The six code bits are:

**URG:** This specifies that the packet has some urgent data in it. The urgent pointer field that follows will hold the offset from where the urgent data begins.

**ACK:** This signifies that the acknowledgment number field is valid.

**PSH:** This specifies that this segment requests a push. This flag is set usually for packets that contain TCP requests or responses.

**RST:** This is the reset bit that tells the host to reset the connection. That is, if this bit is set then it means that the connection has to be reset immediately.

**SYN:** This bit specifies that the sequence number is valid. In other words this bit tells both the machines to synchronize the sequence numbers. Usually set when the hosts try for a connection.

**FIN:** This specifies that the sender has reached the end of the byte stream.

**Checksum:** This field contains a 16-bit integer checksum that is used to verify the integrity of the data as well as the TCP header.

Now that we have had a look at the TCP header, we can see how a connection is established between a host and a server.

[[http://www.calsoftlabs.com/images/icon_top.gif](http://www.calsoftlabs.com/whitepapers/website-blocking.html#top)](http://www.calsoftlabs.com/whitepapers/website-blocking.html#top)

## Establishing a TCP connection

To establish a connection, TCP uses a three-way handshake.

First the host that wants to connect sends a request to the server that has the SYN bit set. The host then replies to this segment with another segment in which both the SYN and the ACK bits are set. Then the host again sends a third segment that has the ACK bit alone set. With this the three-way handshake gets over and the connection is established between the two machines.

Once the connection is established the data transfer can occur both ways.

[[http://www.calsoftlabs.com/images/icon_top.gif](http://www.calsoftlabs.com/whitepapers/website-blocking.html#top)](http://www.calsoftlabs.com/whitepapers/website-blocking.html#top)

## TCP connection Reset

Apart from the normal closing operation, there may be certain abnormal conditions that force the application or the network software to break a connection. TCP provides the reset facility for such conditions.

To reset a connection, one side initiates the reset by sending a segment that has the RST bit set. The other side responds immediately by aborting the connection. This reset is instantaneous, in the sense that transfer in both directions ceases immediately and all the resources such as buffers are released.

[[http://www.calsoftlabs.com/images/icon_top.gif](http://www.calsoftlabs.com/whitepapers/website-blocking.html#top)](http://www.calsoftlabs.com/whitepapers/website-blocking.html#top)

## Breaking a TCP connection (blocking a particular workstation)

As we have seen to break an ongoing TCP communication it is enough if we send a segment that has the RST bit set to anyone side of the connection. Once a machine receives this segment it immediately breaks the connection.

The following illustration will make this concept clear.

Consider that a machine M1 in our LAN is accessing a server S1 through the Internet. The machine M1 will send a TCP request packet to the server S1, which will in turn send a response packet to M1 and their communication is established. Our aim is to restrict the machine M1 from accessing the Internet. How do we do this?

To break a connection we have to send a segment with the RST bit set to either machine M1 or to the server S1. Consider the situation where the machine M1 has sent a request to the server S1 and is waiting for the response. The request packet that is sent to the server S1 is captured by our monitoring application that is capable of capturing all the packets that travel in the network. Once we get this packet we immediately send the packet that has the RST bit set to the machine M1, since LAN is much faster than the Internet our packet will reach machine M1 before the response from the actual server. Assuming for now that the packet that we send is properly framed, it will be accepted by the IP and the TCP software on machine M1. Thus machine M1 will think that the server S1 has initiated a reset and will immediately reset the connection. The actual response that comes from the original server is ignored.

The above section describes the logic involved in blocking a workstation from accessing the Internet. But, the catch lies in the framing of the packet with the RST bit set and sending it to the machine M1.

#### Framing of a RST packet

The packet that we send to machine M1 has to be framed properly so that the IP and the TCP software in the target machine accept it.

The various data that are required for framing a reset packet are:

* Source Ethernet Address
* Destination Ethernet Address
* Source IP Address
* Destination IP Address
* Total length of the packet as given by the IP header length field
* IP Checksum
* Source port
* Destination port
* Sequence number
* Acknowledgment number
* TCP Checksum

The values for most of the above fields are picked up from the packet that we obtain. Let us see how the values in these fields are filled. Let us call the packet that we have captured as P1 and the new packet that we are framing as P2.

The **source Ethernet address** of P2 is the **destination Ethernet address** in P1.

The **destination Ethernet address** of P2 is the **source Ethernet address** in P1.

The **source IP address** of P2 is the **destination IP address** in P1.

The **destination IP address** of P2 is the **source IP address** in P1.

The total length of the packet is the length of the packet in bytes minus the Ethernet header length.

The IP checksum is then computed using the standard algorithm for the IP checksum computation routine that has been listed in the RFC.

The **source port** of P2 is the **destination port** in P1.

The **destination port** of P2 is the **source port** in P1.

The sequence number and the acknowledgment numbers are ignored so we need not fill anything in these fields.

The TCP checksum is computed using the standard algorithm for TCP checksum computation as listed in the RFC.

Here is the code listing of a routine called SendPacket that frames a reset packet and sends this packet to the target machine. Along with this the IP and the TCP checksum computation routines are also listed.

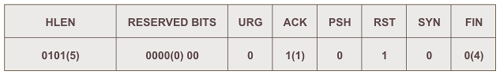
**Note:** The definitions of the various structures used here are given in the previous paper.

[view plaincopy to clipboardprint?](http://www.calsoftlabs.com/whitepapers/website-blocking.html)

1. CONTROL\_BLOCK Adapter;
2. void SendPacket( int gnSend )
3. {
4. short int nSendIndex;
5. unsigned short \*pIP,\*pTcp;
6. int nAckNum,finalAck;
7. SeqAndAckNos ackNum;
8. Length lTotalLength;
9. Checksum ipChksum,tcpChksum;
10. unsigned char IPHead[20]; // IPAddr[8];
11. PVOID Packet;
12. char seqNo[20],ackNo[20],szAddr[25];
13. ::FillMemory(seqNo,20,'\0');
14. ::FillMemory(ackNo,20,'\0');
15. ::FillMemory(szAddr,25,'\0');
16. // Allocate memory for the packet to be sent to the Ws.
17. Packet=PacketAllocatePacket((LPADAPTER)Adapter.hFile);
18. if (Packet != NULL)
19. {
20. PacketInitPacket(
21. (PACKET \*)Packet,
22. Adapter.lpMem2,
23. 54
24. );
25. /\*\*\*\*\*\*Start Filling the Packet to be sent \*\*\*\*\*\*\*/
26. /\*\*\*\*\*\*\*\*\*\* Ethernet header \*\*\*\*\*\*\*\*\*\*\*\*\*\*/
27. // 14 bytes
28. // Destination and Source ethernet address.
29. for(nSendIndex = 0;nSendIndex < 6;nSendIndex++)
30. {
31. // Copy the packets destination address
32. // into our source address and the source address
33. // into our destination address.
34. Adapter.lpMem2[nSendIndex] =
35. pdBlockedData[gnSend].pData[nSendIndex+6];
36. Adapter.lpMem2[nSendIndex+6] =
37. pdBlockedData[gnSend].pData[nSendIndex];
38. }
39. // Protocol type is Internet protocol.
40. Adapter.lpMem2[12] = 0x08;
41. Adapter.lpMem2[13] = 0x00;
42. /\*\*\*\*\*\*\*\*\*\* IP header details.\*\*\*\*\*\*\*\*\*\*\*\*\*\*/
43. // 20 bytes
44. // Version and Header length.
45. IPHead[0] = Adapter.lpMem2[14] = 0x45;
46. // Type of service. Normal.
47. IPHead[1] = Adapter.lpMem2[15] = 0x00;
48. // Total packet length
49. IPHead[2] = Adapter.lpMem2[16] = 0x00;
50. IPHead[3] = Adapter.lpMem2[17] = 0x28;
51. // Also get total packet length of the obtained packet.
52. lTotalLength.szTotalLength[1] =
53. pdBlockedData[gnSend].pData[16];
54. lTotalLength.szTotalLength[0] =
55. pdBlockedData[gnSend].pData[17];
56. finalAck = lTotalLength.nTotalLength;
57. // Identification number.
58. IPHead[4] = Adapter.lpMem2[18] =
59. pdBlockedData[gnSend].pData[18];
60. IPHead[5] = Adapter.lpMem2[19] =
61. pdBlockedData[gnSend].pData[19];
62. // flags + fragment offset.. set it to nothing.
63. IPHead[6] = Adapter.lpMem2[20] = 0x00;
64. IPHead[7] = Adapter.lpMem2[21] = 0x00;
65. // Time to live for the packet.
66. // no significance for selecting this 77.
67. IPHead[8] = Adapter.lpMem2[22] = 0x77;
68. // Protocol used (TCP)
69. IPHead[9] = Adapter.lpMem2[23] = 0x06;
70. // previous checksum.
71. IPHead[10] = 0x00;
72. IPHead[11] = 0x00;
73. // source IP address and destination IP address.
74. for(nSendIndex = 26;nSendIndex < 30;nSendIndex++)
75. {
76. // As with the ethernet address,
77. // do the same swapping with the IP
78. // addresses also.
79. IPHead[nSendIndex-14] = Adapter.lpMem2[nSendIndex] =
80. pdBlockedData[gnSend].pData[nSendIndex+4];
81. IPHead[nSendIndex-10] = Adapter.lpMem2[nSendIndex+4] =
82. pdBlockedData[gnSend].pData[nSendIndex];
83. }
84. // Set the pointer to point to
85. // the buffer where the IP header lies.
86. pIP = (unsigned short \*)&IPHead[0];
87. // Call the checksum computation routine.
88. ipChksum.iChksum = htons( IpChksum( pIP ) );
89. ::ZeroMemory( szAddr, 25 );
90. // Fill the computed checksum
91. Adapter.lpMem2[24] = ipChksum.szChksum[0];
92. Adapter.lpMem2[25] = ipChksum.szChksum[1];
93. ::ZeroMemory( szAddr, 25 );
94. /\*\*\*\*\*\*\*\*\*\*\*\*  Fill the TCP header \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/
95. // 20 bytes.
96. // Source Port
97. Adapter.lpMem2[34] = pdBlockedData[gnSend].pData[36];
98. Adapter.lpMem2[35] = pdBlockedData[gnSend].pData[37];
99. // Destination port
100. Adapter.lpMem2[36] = pdBlockedData[gnSend].pData[34];
101. Adapter.lpMem2[37] = pdBlockedData[gnSend].pData[35];
102. // Sequence number
103. Adapter.lpMem2[38] = pdBlockedData[gnSend].pData[42];
104. Adapter.lpMem2[39] = pdBlockedData[gnSend].pData[43];
105. Adapter.lpMem2[40] = pdBlockedData[gnSend].pData[44];
106. Adapter.lpMem2[41] = pdBlockedData[gnSend].pData[45];
107. // Acknowledgment number computation
108. ackNum.szSeqOrAck[0] = pdBlockedData[gnSend].pData[38];
109. ackNum.szSeqOrAck[1] = pdBlockedData[gnSend].pData[39];
110. ackNum.szSeqOrAck[2] = pdBlockedData[gnSend].pData[40];
111. ackNum.szSeqOrAck[3] = pdBlockedData[gnSend].pData[41];
112. // compute ack number as the sum of seq number
113. // "plus"
114. // Number of bytes received
115. // "minus"
116. // 40 ( header length of IP and TCP )
117. // Convert every thing to network-Byte order
118. ::ZeroMemory( szAddr, 25 );
119. // convert the ack number from
120. // Network-byte order to host byte order.
121. nAckNum = htonl(ackNum.nNumber);
122. ::ZeroMemory( szAddr, 25 );
123. // increment the ack number by one alone.
124. nAckNum = finalAck - 40;
125. // convert it back to host-byte order
126. ackNum.nNumber = htonl(nAckNum);
127. // fill the ack. number to be sent
128. Adapter.lpMem2[42] = ackNum.szSeqOrAck[0];
129. Adapter.lpMem2[43] = ackNum.szSeqOrAck[1];
130. Adapter.lpMem2[44] = ackNum.szSeqOrAck[2];
131. Adapter.lpMem2[45] = ackNum.szSeqOrAck[3];
132. // Header length + reserved + flags
133. Adapter.lpMem2[46] = 0x50;
134. Adapter.lpMem2[47] = 0x14;
135. // Window. For the reset packet 0 is enough.
136. Adapter.lpMem2[48] = 0x00;
137. Adapter.lpMem2[49] = 0x00;
138. // Checksum computation.
139. // Fill zero initially in the checksum field.
140. Adapter.lpMem2[50] = 0x00;
141. Adapter.lpMem2[51] = 0x00;
142. // Urgent pointer
143. Adapter.lpMem2[52] = 0x00;
144. Adapter.lpMem2[53] = 0x00;
145. Incase we are trying to send a page across
146. to the machine then include the code that
147. has been listed in the "sending a page across
148. to the user" section here.
149. // set the corresponding pointers to
150. // be passed to the checksum
151. // computation routine.
152. pIP = (unsigned short \*)&IPHead[12];
153. pTcp = (unsigned short \*)&Adapter.lpMem2[34];
154. // Call the checksum computation routine.
155. tcpChksum.iChksum = TcpChksum( pTcp, pIP );
156. // Fill the actual checksum value.
157. Adapter.lpMem2[50] = tcpChksum.szChksum[0];
158. Adapter.lpMem2[51] = tcpChksum.szChksum[1];
159. // Send the packet to the blocked workstation.
160. PacketSendPacket(
161. (LPADAPTER)Adapter.hFile,
162. (PACKET \*)Packet,
163. TRUE
164. );
165. // Free the allocated pool.
166. PacketFreePacket((PACKET \*)Packet);
167. }
168. }



#### Code Analysis

* Lines 1-14: Just have the local declarations that are needed for the routine.
* Line 15: Calls the routine PacketAllocatePacket for allocating the buffer that is to be filled with the proper values and sent.
* Lines 18-22: Calls the PacketInitPacket routine that initializes the buffer to be sent.
* Lines 23-31: Filling of the Ethernet header details.
  + Lines 23-29: Fill the Ethernet Source and destination addresses. As mentioned earlier it is a swapping of the addresses that have been obtained from the captured packet.
  + Lines 30 & 31: Setting the frame type as 0800 for specifying that the packet is a TCP/IP packet.
* Lines 32-59: Filling the packet with IP header details.
  + Line 32: Set the IP version and the header length. The value 45 is separated into 2 nibbles. 4 in 45 specifies that we use the Ipv4 and the 5 in 45 specifies that the header length is 5\*4=20 bytes in length.
  + Line 33: Set the type of service to 0. This signifies normal delivery. The various other values for this field are beyond the scope of this discussion.
  + Line 34 & 35: Fill the total packet length. This length includes only the IP header, TCP header and the data. It does not include the Ethernet header. Since the packet that we send does not contain any data and it contains the three headers alone, its length is 54 bytes. With Ethernet header length reduced the total packet length for IP is 40 bytes.
  + Lines 39 & 40: Set the Identification number to the same.
  + Lines 41 & 42: Set the flags and the fragment offset to 0. Since we do not have any offset bytes specified in the packet and also there is no need to set any flags in the IP header since for the reset packet all the flags are insignificant.
  + Line 43: Set the TTL for the packet to some number. The number 77 that we have chosen here does not have any significance.
  + Line 44: Set the protocol used field to 06. This value specifies that the protocol used is the Transmission Control Protocol (TCP).
  + Line 45 & 46: Set the checksum initially before computing to zero.
  + Line 47-53: Fill the source and the destination IP addresses. Assign the destination IP address of the packet captured as the source IP address of the packet to be sent and assign the source IP address of the packet captured as the destination IP address of the packet to be sent.
  + Lines 54-58: Calls the routine IpChksum to compute the IP checksum and then fill it in the fields that were initially filled with zeroes. The routine IpChksum is clearly explained in the following section.
* Lines 60-94: Filling the packet with TCP header details.
  + Lines 60 & 61: Fill the source port of the packet with the destination port of the packet that we obtained.
  + Lines 62 & 63: Fill the destination port of the packet with the source port of the packet that we obtained.
  + Lines 64-67: Copy the acknowledgment number in the packet that we obtained into the sequence number of the packet that we have to send. The sequence number does not have any significance. It is treated invalid because the SYN bit in the TCP header is not set.
  + Lines 68-80: If the packet contains some reliable data that has to be sent then the acknowledgment number has to be computed properly. The acknowledgment number is computed as the sum of sequence number "plus" Number of bytes received "minus" the sum of the header lengths of IP and the TCP headers.
  + Lines 81 & 82: Set the header length, the reserved bits and the flags. The header length is set to 5 this has the same interpretation as in the IP header. The reserved bits are set to zero and in the code bits the RST bit is set. How we form this value is shown below.   
    Consider the 2 bytes that constitute the header length, reserved bits and the code bits.  
    
  + In the above figure it is clearly shown how we set the code bits. The numbers in brackets are the decimal equivalents for the 4 bits in binary.
  + Lines 83 & 84: Set the window size to zero since it has got no significance in a RST packet.
  + Lines 85 & 86: Initially as done earlier set the checksum value to zero before calling the checksum computation routine.
  + Lines 87 & 88: Set the urgent pointer to zero since we don't have any urgent data in the packet.
  + Lines 91-94: Call the routine TcpChksum to compute the TCP checksum of the packet to be sent. The routine TcpChksum is clearly listed in the following section.
* Lines 95-99: Call the PacketSendPacket routine to send the packet out on the network.
* Line 100: Call the routine PacketFreePacket to release the allocated space.

**Note:** The functions Packet, Packet called by the above routine are listed and clearly explained in the paper titled - Monitoring Ethernet Network activity using NDIS drivers.

Here is the listing of the routine that computes the IP checksum. The routine has been implemented as described in the standard RFC for the Internet Protocol (RFC 791).

Before we see the code for the checksum computation, let us look at the algorithm as given in RFC 791.

The checksum algorithm is:

The checksum field is the 16 bit one's complement of the one's complement sum of all 16-bit words in the header. For purposes of computing the checksum, the value of the checksum field is zero.

This algorithm has been implemented in the subsequent routine that has been listed.

[view plaincopy to clipboardprint?](http://www.calsoftlabs.com/whitepapers/website-blocking.html)

1. unsigned short IpChksum(unsigned short \*pIP)
2. {
3. // variable holding the value of the IP header length.
4. unsigned short nLen=20;
5. // Variable that holds the checksum value.
6. unsigned chksum = 0;
7. // Variable that holds the value of the final checksum after
8. // computing the one's complement.
9. unsigned short finalchk;
10. // Divide the length by 2 for computing 16-bit checksum.
11. nLen >>= 1;
12. // compute the 16 bit checksum.
13. for(int i = 0; i < nLen;i++)
14. {
15. chksum += htons(\*pIP++);
16. }
17. // Fold the 32 bit value into the 16-bit field.
18. chksum = (chksum >> 16) + (chksum & 0xffff);
19. chksum += (chksum >> 16);
20. // Compute the one's complement of the checksum.
21. finalchk = (~chksum & 0xffff);
22. // return the final checksum.
23. return finalchk;
24. }



#### Code Analysis

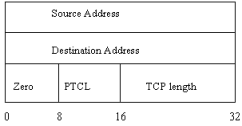
* Line 4: Set the header length to 20. This is the length of the IP header.
* Line 9: Divide the length by 2 since we are computing the 16-bit sum of the header.
* Lines 11-14: Add all the bytes in the IP header as the 16-bit word as specified in the algorithm.
* Line 16: Since the checksum is only a 16-bit word, fold the 32-bit checksum that we have computed into 16-bit word.
* Line 19: Compute the one's compliment of the checksum.
* Line 21: Return the computed checksum.

Here is the listing for the TCP checksum computation. The routine has been implemented as described in the standard RFC for Transmission Control Protocol (RFC 793).

The algorithm for the computation of the checksum as given in the RFC:

*The checksum field is the 16 bit one's complement of the one's complement sum of all 16-bit words in the header and text. If a segment contains an odd number of header and text octets to be checksummed, the last octet is padded on the right with zeros to form a 16-bit word for checksum purposes. The pad is not transmitted as part of the segment. While computing the checksum, the checksum field itself is replaced with zeros. The checksum also covers a 96-bit pseudo header conceptually prefixed to the TCP header. This pseudo header contains the Source Address, the Destination Address, the Protocol, and TCP length. This gives the TCP protection against misrouted segments. This information is carried in the Internet Protocol and is transferred across the TCP/Network interface in the arguments or results of calls by the TCP on the IP.*

The diagram below shows the contents of the pseudo header.



*The TCP Length is the TCP header length plus the data length in octets (this is not an explicitly transmitted quantity, but is computed), and it does not count the 12 octets of the pseudo header.*

The above algorithm has been implemented in the routine shown below.

[view plaincopy to clipboardprint?](http://www.calsoftlabs.com/whitepapers/website-blocking.html)

2. unsigned short TcpChksum(unsigned short \*pTcp, unsigned short \*pIP )
3. {
4. // Variable that holds the checksum not complemented.
5. unsigned chksum = 0;
6. // Variable that holds the final checksum value.
7. unsigned short finalchk;
8. // Variables that the TCP & IP header lengths.
9. unsigned short nLen = 20, nIpLen = 8;
10. // divide both length values by 2 since we are adding 16 bits.
11. nIpLen >>= 1;
12. nLen >>= 1;
13. // compute the 16 bit sum of the pseudo header.
14. for(int i = 0;i < nIpLen;i++)
15. {
16. chksum += (\*pIP++);
17. }
18. // length + the protocol type.
19. chksum += htons( 26 );
20. // compute the 16 bit sum of the TCP header and data.
21. for(i = 0;i < nLen;i++)
22. {
23. chksum += (\*pTcp++);
24. }
25. // Fold the 32 checksum into 16 bit.
26. chksum = (chksum >> 16) + (chksum & 0xffff);
27. chksum += (chksum >> 16);
28. // Get the one's complement of the checksum.
29. finalchk = (~chksum & 0xffff);
30. // return the final checksum.
31. return finalchk;
32. }



#### Code Analysis

* Line 8: Set the TCP header length to 20 bytes.
* Line 10: As done in the IP checksum computation divide the length field by 2 since we are going to compute the 16-bit sum of the header.
* Lines 13-18: Compute the 16-bit sum of the pseudo header. In this, we add the 16-bit sum of the IP addresses (source and the destination addresses), protocol used and the TCP header length.
* Lines 20-23: Compute the 16-bit sum of the TCP header and the data.
* Lines 25 & 26: Fold the 32-bit checksum that we have computed in to a 16-bit word.
* Line 28: Compute the one's complement of the checksum that we have computed.
* Line 30: Return this checksum value as the TCP checksum of the packet.

#### Sending a page across to the user

The previous section clearly describes how we can frame a TCP reset packet and send it across to a machine that we want to block. The sending of the reset packet actually does not make the user know that the access to that machine has been restricted instead just displays a message saying that the connection with the server was reset. Instead it would be better if we can send that particular machine a page that says "Access Denied" or some error message that will make the user aware that the machine has been blocked.

This requires some modification to the SendPacket and the TcpChksum routines.

Say we want to display the word "Access" on the client's browser if he tries accessing the Internet.

#### The modifications that have to be done to the SendPacket routine for this are:

* Line 21: The value 54 has to be changed to 122 we will see how it is 122 shortly.
* Line 35: The value 0x28 (total packet length excluding the Ethernet header, which is 54-14=40) has to be replaced with 0x6c(total packet length excluding the Ethernet header, which is 122-14=108).
* Line 82: The most important change that has to be done to the packet is changing the flags. Previously, since we were trying to reset the connection we had set the RST bit in the code bits section of the TCP header. Now since we are sending a proper HTTP response we have to set the ACK, PSH and the FIN flags. ACK has to be filled in response to the request packet that we have obtained. The PSH flag is set to indicate that this is a response packet and the FIN bit is set to tell the other end that the data transfer from our side is over.

After doing all this, please include these lines of code after line 88.

[view plaincopy to clipboardprint?](http://www.calsoftlabs.com/whitepapers/website-blocking.html)

1. /\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* Data follows \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/
2. Strcpy( Adapter.lpMem+54, “Http 1.0 200 OK\r\nContent-type..
3. text/htmt\r\n\r\n<HTML>Access</HTML>\r\n\r\n\0” );



The data part of the packet contains:

Http 1.0 200 OK..Content-type.. text/html..<HTML >Access </HTML>...

This is the HTML code of the page that is sent to the target machine's browser.

Now to the routine TcpChksum we need to make a single change.

* Line 8: Change the value of the nLen variable from 20 to 88. Previously since there was no data and only the header was there we had set the length to the length of the header. Now that we have the data also, the length includes the length of the data also. Thus the total length of the TCP header and data is total length of the packet excluding the IP and the Ethernet headers. That is 122-34=88.

With the above method we can send any amount of data. That is, we can hijack the connection that is going on between the machine in our network and the server. Instead of the server sending the data, we can send whatever data we want to the machine in our network.

To frame the reset packets and the blocking packets, using the PACKET.SYS driver is not the only way. We can use similar drivers that are capable of sending packets on to the Ethernet Network.

#### Why sockets cannot do this?

Sockets as you know are higher level abstractions of the TCP/IP and other protocols. They have been provided to ease the work of Network programmers in framing packets that travel across networks. It is enough if the network programmers just pass the data that they need to send through a socket interface, the other details such as the header, checksum and the rest are filled in by the IP and the TCP software in the machine. But for framing a reset or a blocking packet we need to get the header details like the sequence number and acknowledgment number and then form a packet after computing the value for these numbers.

Consider a situation where an application uses the sockets to communicate. This application will ask the IP and the TCP layers to read the IP packets and send the data that is present. The TCP software that is present will maintain the header information of the packet that it read. So when the application sends a reply back to the client, the TCP software fills the appropriate sequence number, the acknowledgment numbers and the checksum and sends the packet across. But this is only for the packets destined to our machine.

Now looking at how this can fit into our application, we see that since the IP software and the TCP software are above the Ethernet, they cannot capture all the packets that come to the card. This is done by the card and the entire packet instead of been handed over to the IP software is given to our application. Thus our driver over rides the IP layer and the TCP layer. Since the TCP layer does not get the packet at all, it cannot hold any of the header information. Thus when the application has to send a blocking packet it cannot do so by just providing the socket with the data part alone, but it has to specify the header information. Since sockets does not allow the user to enter the header information and all but does that on its own, it is not possible for us to enter the header information. Since the IP and the TCP software present does not know the header information of the previous packet they tend to fill the wrong sequence and the acknowledgment numbers there by causing the packet to be reje cted. Thus the use of sockets for sending the reset or the blocking packets will not work.

[[http://www.calsoftlabs.com/images/icon_top.gif](http://www.calsoftlabs.com/whitepapers/website-blocking.html#top)](http://www.calsoftlabs.com/whitepapers/website-blocking.html#top)

## A Connection terminator application

Now that we have seen how blocking can be done, let us look at an application that is capable of terminating the connections for specified machines.

This application captures all the packets that come to the NIC; it compares the source IP address in the captured packet with the addresses that are specified in the blocking file. If the source IP address of the packet matches any of the addresses in the file, then that particular machine is restricted from accessing the Internet, in other terms; its connection is terminated.

This section explains the architecture of the connection terminator but does not include any code samples for the application.

The application comprises two threads one for grabbing the packets from the network and another for performing matching and sending the reset packets.

#### Why do we go in for a multi-threaded application?

As we have already seen, to perform a blocking operation, after obtaining the packet we have to compare it with the addresses that are specified in the blocking file, frame the reset packet and send it to the machine that has to be restricted. These operations put together will be very slow when compared to the rate at which the packets come to the NIC. If the application is a single-threaded application, which will read a packet and perform the above operations and then read the next packet, it is sure of losing packets. To prevent this we go in for a multi-threaded application with one thread continuously reading packets and the other doing the blocking.

The application has to make a ReadFile call to read a packet that comes to the NIC. Once the driver reads a packet from the NIC, it has to copy that buffer to the buffer that we have provided, in other words, the packet has to be lifted up from the kernel level to the user level. The lifting process takes quite sometime, if with in this time a packet comes to the NIC that packet is not grabbed and is lost. Thus even if we call the ReadFile function continuously in an infinite while loop we tend to lose packets. This is because only after one ReadFile returns the other is initiated and so on. Thus if during a ReadFile is in progress a packet comes, the packet is lost. To avoid this, instead of making the ReadFile call synchronous, we make this asynchronous. By asynchronous, we mean that it is not necessary to wait until the previous ReadFile call returns, instead we can initiate some more reads. Let us see with this how we overcome the problem of missing the packets that come during the ReadFile is in progress. Consider that we have initiated 15 reads, now when a packet comes, the first read is invoked and it keeps reading. During this time if another packet comes then the second read that we have initiated gets invoked and similarly the other reads that we have initiated. Once a read is completed the blocking thread gets notified and this thread gets the buffer in which the packet has been read. The thread matches the obtained packet's IP address with those specified in the blocking file and then calls the SendPacket routine that we have already discussed to send a reset packet. Once this is over, the blocking thread indicates the grabber thread that initiates another read on this buffer. Thus at any time there will be a pending read and if any packet comes to the NIC it will be automatically grabbed. Even after this, if the rate at which the packets come to the NIC is very high then there are possibilities of missing the packets. But doing asynchronous reads gives the maximum efficiency in the sense that losing packets has the least probability.

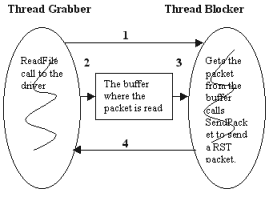
One way of doing this asynchronous reading is to use the IO completion ports. In this method, apart from creating 2 threads we also create 2 IO completion ports. In one thread we initiate some 15 reads by calling the ReadFile method, and then wait in the ReadPort for a read to return. Once a ReadFile call returns, an IO packet is posted to this port. Along with this the buffer that holds the data of the read packet is also present. The thread that does the blocking waits in the ReadPort for a ReadFile call to return. Once a ReadFile call returns, the blocking thread performs the blocking operations. After sending a packet onto the network, the function posts an IO packet to the read thread. The read-thread after initiating 15 reads waits in at a WritePort, and once this receives the IO packet that is posted by the blocking thread, it initiates another read. Thus at any point of time a ReadFile call will be pending to pick a packet that comes to the NIC.

Since we are using a multi-threaded application, we have to take care that all the synchronization issues are addressed, otherwise the application may not function as expected in dual-processor machines.

All this while we saw how we can improve the performance of the application by modifying the higher level application that we have written. We can also get a better performance by modifying the driver also. What can be done at the driver level?

As we have already seen, it takes quite sometime for the driver to lift a packet from the kernel mode to the user mode. In the promiscuous mode, the driver lifts all the packets that come to the NIC from the kernel mode to the user mode. This includes the non-TCP/IP packets also, though we are interested only in TCP/IP packets. Thus if we can modify the driver to filter the non-TCP/IP packets then the amount of packets that are lifted to the user mode from the kernel mode is reduced there by increasing the speed of capture. This is quite simple, instead of writing the code for checking whether the packet is a TCP/IP packet in the application, we can write that piece of code in the driver. If the packet is a TCP/IP packet then the driver will lift that packet from the kernel level and give it to the application otherwise it can return an error code and need not fill the buffer at all.

**A Diagram that shows the multi-threaded architecture of the Connection terminator application:**



Arrow 1: Once a ReadFile returns the grabber thread indicates the completion of the read to the blocker thread. The blocker waits on an IO completion port for the Read to complete. Once the read is over the buffer is presented to the blocker thread. The blocker thread then compares the IP addresses and if it matches it calls the SendPacket to send a reset packet to the machine that has to be blocked.

Arrow 2: Indicates that the Grabber thread fills the buffer with the packet that it has picked up from the network.

Arrow 3: Indicates that the blocker thread reads the packet that is placed by the grabber in the buffer.

Arrow 4: Once the blocker sends a reset packet to the machine or finishes comparing the addresses, it posts a message with the buffer to the grabber thread. Once the grabber thread receives this message and the buffer, it initiates another ReadFile call on this buffer.

Thus the application continuously keeps grabbing all the packets that come to the network and will not allow the machines that are added in the blocking file from accessing the Internet. To enter the IP addresses of the machines that need to be blocked, we can provide the user with a simple UI and store them as text files itself.

Similarly, instead of blocking the entire workstation from accessing the Internet, if we want to block certain sites alone, then what we can do is specify another file in which we will enter IP addresses of the sites that have to be restricted. Once a packet is obtained, the blocker thread along with comparing the source IP address will also compare the destination IP address of the packet and act accordingly. Likewise the application can be extended to perform a lot more functions say blocking certain workstations from accessing certain sites alone. Enabling the user to enter instead of IP address, hostnames also in the blocking files. These are just some extensible features to the above-described application though the core of the application remains the same.

Since we do not use any queue or other variables that are accessed by both the threads simultaneously we have not dealt with the thread synchronization issues. If a queue or some other data structure is used that is accessed by both the threads, then the accesses have to be synchronized using critical section variables or other thread synchronization mechanisms.

[[http://www.calsoftlabs.com/images/icon_top.gif](http://www.calsoftlabs.com/whitepapers/website-blocking.html#top)](http://www.calsoftlabs.com/whitepapers/website-blocking.html#top)

## A Case study on Rule-based control

Here we will look at an application that monitors and blocks the TCP requests based on rules that are created and stored by an administrator.

Before going into what the rules are and the structure of these rules, let us understand the architecture of the system.

**The system comprises few engines:**

**The Grabbing engine:** This engine captures all the packets that come to the network card and add them on to a queue.

**The Rule-matching engine:** This engine removes the packets from the queue, one at a time and matches them with the rules that have been specified (we will look at how this is done shortly).

**The Log Writer engine:** This engine logs the packet details like the URL accessed into a log file.

**The Blocking engine:** This is the engine that sends a reset packet to the machine that has to be restricted from accessing the Internet.

Now that we have seen the basic architecture of the engine, let us see the structure of the rules, how they are framed and stored.

**The basic structure of a rule:**

The various fields that are present in a rule are as follows:

**Rule no:** This member holds the rule number that specifies the priority of the rule.

**Source:** This member holds any one of the following.

* IP address
* Host name
* Host object-List name

**Destination**: This member holds any one of the following

* IP address
* Host name
* Host Object-list name

**Type**: This member holds any one of the following.

* protocol /service name
* port number
* Service object-list name

**Data**: This member holds the URL name or Keyword or the Data object List name. The URL in the packet is compared with this member.   
**From Time**: This member holds the time from which the rule will be effective.   
**To Time**: This member holds the time until which the rule will be in effect.   
**Flag**: Flag is used for interpreting the data in the source, destination, type and data members.   
**Action**: This member holds the action that has to be carried out on a packet that satisfies the rule.

The source and the destination members in the rule structure can hold either the IP address of a machine or the host name or can hold an object-list name. An object-list is a file that consists of a set of IP addresses/host names/object-list names. This object-list termed the Host Object-list.

Similarly the type member of the rule-structure holds the service type say HTTP/FTP etc. This member can either hold the port number that the service uses for example 80 for HTTP or it can hold the service name itself or it can hold an object-list name. Again here the object-list is a file that holds a set of port numbers or service names that have to be matched. This is called a Service-object list.

The data member of the rule can contain any text that will be matched with the data in the packet that is obtained. Here again a set of words to be matched can be stored as an object-list file and the name can be assigned to this member. This object-list is called the data object-list.

The action member of the rule structure specifies the action that has to be performed on the packet that matches this rule. There are three types of actions that can be specified namely **Allow**, **Disallow** and **log**.

The FromTime and the ToTime members of the rule structure specify the time limit within which the particular action will be performed. If the packet is obtained at a time that is less than the FromTime or is greater than the ToTime then the packet is just allowed without performing the specified action on that packet.

Before going into further details of how the rules are stored and loaded on to memory for matching purposes, let us see how the system functions on the whole.

The grabbing engine continuously keeps capturing all the packets that come to the NIC and place them on a queue. The rule matching engine removes the packets from the queue on a per packet basis and matches each packet with the rules that have been specified. Once a packet matches a particular rule then the corresponding action that is specified for that rule is taken. As mentioned earlier, the action can be anyone of Allow, disallow or log. Other valid actions are Allow + log, Disallow + log. Depending on the action the corresponding engine is signaled and action is performed. For example, if the action is Log then the rule matching engine signals the log writer engine and the packet details are logged into the log file. Similarly if the action is to disallow, then the blocking engine is signaled and the blocking packet is sent to the machine that needs to be blocked.

**Note:** How logging is done is very clearly explained in the previous paper titled - Monitoring Ethernet Activity using NDIS drivers and how blocking is done is very clearly explained in this paper.

Now that we have seen how the system functions let see how we can form a rule, how it is stored and how it is matched with the packets.

The rules are composed through a user interface and are stored in a file. These rule files are then loaded on to memory when the system is started. By loading we mean that all the rules that are in the file are buffered in memory and matching is done with this buffer. This is because reading a memory buffer is faster than reading a file. The user can compose as many rules as he needs and can store it as files.

Similarly an object-list can also be composed by the user and stored as object-list files that are also loaded along with the rule file while matching is done.

Let us now see how a rule appears and how it can be matched.   
A rule will be something like this:   
Source: 304.4.34.99   
Destination: 216.23.45.56   
Type: 80   
Data: NULL   
Action: Allow + Log

Now consider that we get a packet. First the source IP address is matched with the source member of the rule. If it matches then the destination IP address of the packet is matched with the destination member of the rule. If the destination also matches then the destination port of the packet is matched with the type member of the rule. If even this matches then the action specified is performed. In this case the details of the packet (say the URL) is logged and the machine is allowed to access the Internet. If any of the members don't match then the rule matching engine ignores this rule and tries matching the packet with the next rule in the rule file that has been loaded. If none of the rules match then the packet is allowed and no action is performed on that. Similarly, if any of the rules match then the other rules in the file are ignored as far as that packet is concerned.

Once the rule file name is specified and the system loads that rule file and starts functioning, it is possible for us to modify that particular rule file as well as load another rule file. Let us see how this is done. If we have to modify, since the file is buffered for matching purposes, the file that is stored on the disk is accessible. We can modify the file by adding new rules or deleting existing rules. Once this is done, for the changes to get reflected we have to stop all the engines and then start the system again.

The stopping of the engine is performed in a series of steps.   
First, the grabbing engine is stopped and the rule matcher engine flushes all the remaining packets in the queue. Once this queue becomes empty, the rule matching engine, the log writer and the blocking engines are stopped. Then the existing rule buffer is cleared and the file is again buffered. Once this is done, the engines are restarted and the new rules come into effect.

In case of loading a new rule file, the same happens but after stopping all the engines and clearing the rule buffer, the new rule file is loaded on to that buffer and the engines are started.

[[http://www.calsoftlabs.com/images/icon_top.gif](http://www.calsoftlabs.com/whitepapers/website-blocking.html#top)](http://www.calsoftlabs.com/whitepapers/website-blocking.html#top)

## Summary

This paper focuses on how we can block a particular workstation from accessing the Internet. The paper also provides some code patches that can be directly used for framing the RST packets. Since the paper is an extension of the previous paper titled - Monitoring Ethernet activity with NDIS drivers, this paper omits all explanations of the driver and the other related DLLs. The reader of this paper is strongly advised to read the previous paper so that this paper will be of maximum help for the reader. The paper also throws light on how actually frames are formed and sent on an Ethernet network. Also the paper provides insight on what are checksums and how they are computed and used by real-time network applications. The paper provides a case study of a rule-based controlling engine, which would aid the user to develop application that would be able to monitor and control networks based on set of rules that are composed by the network administrator