On Teaching TCP/IP Protocol Analysis to Computer Forensics Examiners

Gary C. Kessler[1]
Champlain College
Burlington, VT, USA
gary.kessler@champlain.edu

Abstract

Digital investigators have an increasing need to examine data network logs and traffic, either as part of criminal or civil investigations or when responding to information security incidents. To truly understand the contents of the logs and the data packets, examiners need to have a good foundation in the protocols comprising the Transmission Control Protocol/Internet Protocol (TCP/IP) suite. This paper introduces the use of protocol analyzers and packet sniffers for TCP/IP traffic, and provides examples of normal and suspect TCP/IP traffic. This paper also provides a basis for a discussion of intrusion detection and signature analysis.

Keywords

TCP/IP, protocol analysis, packet sniffing

INTRODUCTION

An earlier paper by the author notes that analysis and interpretation of network traffic is increasingly important to the digital forensics community. Network data -- live traffic, stored communications, or server logs -- contain information that might be of use to the digital investigator, either for information security incident response, forensics applications, research purposes, or intelligence gathering. There is, in fact, so much potentially valuable information in system log files that due diligence requires the investigator to look at as much of this information as possible and the sheer volume makes it nearly impossible to examine every source of data in every case (Kessler & Fasulo, 2007).

There are a variety of sources and types of network information that can be gathered, including (Casey, 2004; Nikkel, 2005):

- Intrusion detection system (IDS) and firewall logs
- Hypertext Transfer Protocol (HTTP), File Transfer Protocol (FTP), e-mail, and other server logs
- Network application logs
- Backtracking of Internet Protocol (IP) and other network packets, and Transmission Control Protocol (TCP) logical connections
- Artefacts and remnants of network traffic on hard drives of seized systems
- Live traffic captured by a packet sniffer or network forensic acquisition software
- Individual systems' routing and Address Resolution Protocol (ARP) tables, and responses to port scans and Simple Network Management Protocol (SNMP) messages

This paper is intended to provide an overview of several aspects of protocol analysis for a computer forensics examiner with a good foundation in basic networking. First, it is meant as an introduction to normal and abnormal TCP/IP traffic. Second, it is intended to demonstrate the use of a packet sniffer, which can be an important tool for network-based examinations or incident response. The examples here use output from tcpdump and WinDump, both command line packet sniffing tools. (A brief guide on their use can be found in the appendix). The paper will, hopefully, spark an interest in a deeper understanding of the internal workings of network protocols and the application of protocol analysis.

[1] Also affiliated with Edith Cowan University, Mount Lawley, Western Australia.

PROTOCOL ANALYZERS

While the format of log files and IDS messages differ from application to application, knowledge of one such system makes interpretation of all similar systems that much easier. Most real-time TCP/IP packet sniffing systems are based on the Unix/Linux packet capture library (libpcap) or its Windows counterpart (winpcap). For purposes of this discussion, then, using libpcap/winpcap software is possibly the best place to start.

Packet sniffers are hardware or software that read all of the packets on a communications channel; in the distant past, all such sniffing required expensive hardware whereas today the same functionality can be found with free software. Packet sniffers can display traffic in real-time, store the packets for later display, and provide detailed interpretation. Packet sniffers are useful tools for network monitoring, investigations, reconnaissance, or just to learn how the protocols work.

The most common command line packet sniffers based on libpcap and winpcap are tcpdump and WinDump, respectively. There are also a number of software applications that offer protocol analysis using a graphical-user interface (GUI), such as Analyzer, Packetyzer, and WireShark (formerly Ethereal).[2]

Although the GUI is better at aggregating network traffic, making it visually appealing and understandable, and providing strong features for analysis and examination, it is important for the digital investigator to start with the command line. The reason is simple; understanding the command line output gives an examiner a much better understanding and appreciation of what is going on under the hood of the GUI. Just as computer forensics examiners learn disk geometry, how the directory stores file entries, and how to use a hex editor so that they can truly understand how "deleted" data can be retrieved and convincingly tell a jury, network analysts need to understand what is happening on the wire so that they truly understand what the GUI is showing.

TCP/IP TRAFFIC EXAMPLES

The following sections will show some TCP/IP traffic in an effort to lead the reader through a demonstration of normal traffic so that abnormal traffic stands out (Chappell & Tittel, 2007; Kessler, 2007; Stevens, 1994). Although abnormal traffic is often flagged by an IDS (if one is present), examiners need to know what normal is to truly appreciate how subtle abnormal can be. In the scenarios below, extraneous traffic has been removed for clarity. On real systems, a lot of traffic will often appear that is "uninteresting" to the examiner and which has to be removed by the use of filters, but one has to see that traffic to know to ignore it.

Scenario #1: Normal TCP Connection Setup

Listing 1 shows a user (at *altamont.champlain.edu*) connecting to an FTP server (at *ftp.example.net*).

The first two packets[3] (Lines 1-2) show *altamont* sending a Domain Name System (DNS) request to a name server (*ns.champlain.edu*) to determine the IP address of the intended destination and being sent the IP address in response. Note use of port 53 (dns) in these User Datagram Protocol (UDP) datagrams.

After obtaining the IP address, *altamont* can now make the connection to the FTP server. Note that the first three packets exchanged between the two systems is the classic TCP 3-way handshake (highlighted within the box):

3. The client sends the server a segment with the synchronization (SYN) bit set and an initial sequence number (ISN) in the client-server direction.
4. The server sends a segment with the SYN and acknowledgement (ACK) bits set, containing the ISN in the server-client direction as well as an acknowledgement of the client's ISN.
5. The client sends a segment with only the ACK bit set and acknowledges the server's ISN.

The three-way handshake is so fundamental to understanding TCP that it is important that individuals learning the protocol see it in action. It is particularly important because manipulating the handshake is the basis for many attacks on computers and networks (Northcutt, Cooper, Fearnow, & Frederick, 2001; Northcutt, Novak, & McLachlan, 2001).

[2] http://networking.champlain.edu/download/tcpip/sniffers.html
[3] In this paper, a *packet* refers to an IP protocol data unit, a *segment* is a TCP data unit, and a *datagram* is a UDP data unit.

After the three-way handshake, the listing shows the server sending 64 bytes of data (Line 6); this is undoubtedly the banner message, welcome message, and login request. The client sends back an acknowledgement (Line 7) and some additional information (Line 8), and the exchange of data begins (Line 9).

The last four lines in the listing (Lines 10-13) show the termination of the TCP connection. Think of the bidirectional TCP logical connection as two unidirectional connections and the termination scenario then makes sense; each host sends the other a segment with the finish (FIN) flag set and the receiver responds by acknowledging the FIN. After the two pairs of FIN/ACK segments are exchanged, the logical connection is terminated.

```
 1      11:47:50.047936 altamont.champlain.edu.4490 > ns.champlain.edu.53:  37273+ A?
        ftp.example.net. (33)
 2      1:47:50.048450 ns.champlain.edu.53 > altamont.champlain.edu.4490:  37273 1/3/0 A
        209.198.87.45 (106)
 3      11:47:50.881828 altamont.champlain.edu.1074 > ftp.example.net.21: S
        1704258988:1704258988(0) win 65535 <mss 1460,nop,nop,sackOK>
 4      11:47:50.937928 ftp.example.net.21 > altamont.champlain.edu.1074: S
        3565913320:3565913320(0) ack 1704258989 win 8760 <mss 1460>
 5      11:47:50.938011 altamont.champlain.edu.1074 > ftp.example.net.21: . ack 1 win 65535
 6      11:47:51.764584 ftp.example.net.21 > altamont.champlain.edu.1074: P 1:65(64) ack 1 win
        8760
 7      11:47:51.926930 altamont.champlain.edu.1074 > ftp.example.net.21: . ack 65 win 65471
 8      11:47:57.478597 altamont.champlain.edu.1074 > ftp.example.net.21: P 1:17(16) ack 65 win
        65471
 9      11:47:57.506130 ftp.example.net.21 > altamont.champlain.edu.1074: P 65:123(58) ack 17 win
        8760
 :                     :
 :                     :
 :                     :
10      11:48:19.215466 ftp.example.net.21 > altamont.champlain.edu.1074: F 421:421(0) ack 123 win
        8760
11      11:48:19.215555 altamont.champlain.edu.1074 > ftp.example.net.21: . ack 422 win 65115
12      11:48:19.220559 altamont.champlain.edu.1074 > ftp.example.net.21: F 123:123(0) ack 422 win
        65115
13      11:48:19.296084 ftp.example.net.21 > altamont.champlain.edu.1074: . ack 124 win 8760
```

Listing 1

It is relatively straight-forward for users to learn how the protocols used on the Internet operate -- install protocol analysis software (e.g., WireShark), turn it on, perform a single function on the network (e.g., download a Web page, send and retrieve e-mail, or logon to an FTP server), stop the protocol analyzer, and save the captured packets. Study of this set of packet traces can teach a lot about the normal operation of the protocols. Trying this on different operating systems can also be quite illuminating; ping and traceroute, for example, operate noticeably differently on Linux and Windows system.[4]

Abnormal and other interesting traffic can also be obtained by using so-called hacker tools (e.g., port scanners) and/or applications that allow the user to craft their own packets (e.g., hping2). In this way, users can learn some of the patterns of abnormal traffic and the signatures (or syndromes) of suspicious events.

It is important to note that while the actual attacks on systems often employ bogus or spoofed IP addresses, nearly all such network attacks are preceded by some form of reconnaissance. Thus, at some point in time, probing traffic using a *real* IP address pointing to the attacker must have been employed. It is necessary to locate that traffic in order to trace an attack back to its roots.

The remainder of this section will show some packet listings of TCP/IP traffic, provide a discussion of the main points of interest, and introduce network intrusion signature detection and analysis (Northcutt, Cooper, et al., 2001; Northcutt, Novak, & McLachlan, 2001). The traffic shown in the following scenarios do not have any necessary relationship to each other but are representative of a number of different network-based Internet attacks.

Scenario #2[5]

Listing 2 shows four transmissions; host *foo.example.net* sending a packet to *holmes* and *holmes*' response, and *foo* sending a packet to *watson* and *watson*'s response. Note that *foo* has sent nearly identical packets to both *holmes* and *watson*, including several TCP options (maximum segment size, use of selective-acknowledgements, a timestamp,

[4] A set of packet traces that includes HTTP, FTP, telnet, e-mail, ping (Linux and Windows), traceroute (Linux and Windows), and secure traffic can be found at http://digitalforensics.champlain.edu/public/capture_files.zip.
[5] Ftp is the name of the service associated with TCP port 21.

and a window scale value) but that the two responses are different, both in the options that are accepted and the order in which the options are listed.

```
13:58:18.387461 foo.example.net.1565 > holmes.ftp: S 2115515674:2115515674(0) win 32120 <mss
1460,sackOK,timestamp 126466936 0,nop,wscale 0> (DF)
13:58:18.392988 holmes.ftp > foo.example.net.1565: S 3478333904:3478333904(0) ack 2115515675 win
10136 <nop,nop,timestamp 126470710 126466936,nop,wscale 0,nop,nop,sackOK,mss 1460> (DF)
14:00:46.999055 foo.example.net.4238 > watson.ftp: S 2262117252:2262117252(0) win 32120 <mss
1460,sackOK,timestamp 126481796 0,nop,wscale 0> (DF)
14:00:47.029487 watson.ftp > foo.example.net.4238: S 26347543:26347543(0) ack 2262117253 win 8760
<mss 1460,nop,nop,sackOK> (DF)
```

Listing 2

The different responses might be expected since *holmes* and *watson* are different computers. But they are also different operating systems; the former is Linux and the latter is Windows. Although IP and TCP are standardized protocols, there are no standardized implementations. Sending segments with many different types of requested options and/or TCP flag settings is one way in which a host can perform *operating system fingerprinting* of another host; the way in which a host responds to both normal and abnormal events gives a clue as to the type of operating system.

Scenario #3[6]

Listing 3 shows a large number of UDP datagrams sent from the host *holmes* to host *alabaster.champlain.edu*; there are also several Internet Control Message Protocol (ICMP) Port Unreachable responses from *alabaster*. What is happening here is a *UDP port scan*, where *holmes* is trying to determine which UDP ports in the range 130-140 are listening (i.e., open) on *alabaster*. Due to the nature of UDP, no response to a probe on a given port *might* mean that the port is closed; an ICMP Port Unreachable message definitely means that the port is closed. In this case, *holmes* tries three times to connect to this set of UDP ports on *alabaster* unless it gets some sort of response.

```
16:25:00.711083 holmes.39272 > alabaster.champlain.edu.135: udp 0
16:25:00.711923 holmes.39272 > alabaster.champlain.edu.140: udp 0
16:25:00.712543 holmes.39272 > alabaster.champlain.edu.136: udp 0
16:25:00.712703 alabaster.champlain.edu > holmes: icmp: alabaster.champlain.edu udp port 140
unreachable
16:25:00.713223 alabaster.champlain.edu > holmes: icmp: alabaster.champlain.edu udp port 136
unreachable
16:25:00.714060 holmes.39272 > alabaster.champlain.edu.netbios-dgm: udp 0
16:25:00.714763 holmes.39272 > alabaster.champlain.edu.130: udp 0
16:25:00.715275 holmes.39272 > alabaster.champlain.edu.netbios-ns: udp 0
16:25:00.715337 alabaster.champlain.edu > holmes: icmp: alabaster.champlain.edu udp port 130
unreachable
16:25:00.715822 holmes.39272 > alabaster.champlain.edu.132: udp 0
16:25:00.716797 holmes.39272 > alabaster.champlain.edu.netbios-ssn: udp 0
16:25:00.716858 alabaster.champlain.edu > holmes: icmp: alabaster.champlain.edu udp port 132
unreachable
16:25:00.717692 holmes.39272 > alabaster.champlain.edu.134: udp 0
16:25:00.718276 holmes.39272 > alabaster.champlain.edu.133: udp 0
16:25:00.718388 alabaster.champlain.edu > holmes: icmp: alabaster.champlain.edu udp port 134
unreachable
16:25:00.718895 alabaster.champlain.edu > holmes: icmp: alabaster.champlain.edu udp port 133
unreachable
16:25:00.802856 holmes.39272 > alabaster.champlain.edu.131: udp 0
16:25:01.024999 holmes.39272 > alabaster.champlain.edu.135: udp 0
16:25:01.025761 holmes.39272 > alabaster.champlain.edu.netbios-dgm: udp 0
16:25:01.026355 holmes.39272 > alabaster.champlain.edu.netbios-ns: udp 0
16:25:01.026930 holmes.39272 > alabaster.champlain.edu.netbios-ssn: udp 0
16:25:01.028203 alabaster.champlain.edu > holmes: icmp: alabaster.champlain.edu udp port 131
unreachable
16:25:01.844952 holmes.39272 > alabaster.champlain.edu.netbios-ssn: udp 0
16:25:01.845647 holmes.39272 > alabaster.champlain.edu.netbios-ns: udp 0
16:25:01.846286 holmes.39272 > alabaster.champlain.edu.netbios-dgm: udp 0
16:25:01.846932 holmes.39272 > alabaster.champlain.edu.135: udp 0
```

Listing 3

[6] Netbios-ns, netbios-dgm, and netbios-ssn are the names of the services associated with UDP ports 137, 138, and 139, respectively.

Scenario #4[7]

Listing 4A shows a series of TCP packets from host *holmes* to host *watson* (responses from *watson* are not shown). Close examination of the packets show that these are all the first part of TCP's three-way handshake and they are all sent in less than 0.01 seconds. Furthermore, although the source and destination ports appear to be in quasi-random order, the reader will observe that if the segments are rearranged in sequential order by the source port, the destination ports are also in sequential order.

What we have here is a classic TCP port scan using a set of TCP ports from a user-defined list.

```
13:21:45.010117 holmes.4033 > watson.220: S 93266:93266(0) win 8192
13:21:45.011128 holmes.4003 > watson.ftp: S 92918:92918(0) win 8192
13:21:45.012014 holmes.4005 > watson.telnet: S 92946:92946(0) win 8192
13:21:45.013095 holmes.4004 > watson.22: S 92932:92932(0) win 8192
13:21:45.014107 holmes.4019 > watson.110: S 93094:93094(0) win 8192
13:21:45.015865 holmes.4010 > watson.63: S 93016:93016(0) win 8192
13:21:45.016763 holmes.4021 > watson.nntp: S 93106:93106(0) win 8192
13:21:45.018001 holmes.4016 > watson.80: S 93076:93076(0) win 8192
13:21:45.018456 holmes.4017 > watson.92: S 93154:93154(0) win 8192
13:21:45.018997 holmes.4034 > watson.396: S 93280:93280(0) win 8192
13:21:45.019562 holmes.4031 > watson.215: S 93238:93238(0) win 8192
13:21:45.020017 holmes.4002 > watson.17: S 92912:92912(0) win 8192
```

Listing 4A

Listing 4B is similar in nature to Listing 4A although, clearly, different. In this case, host *foo.example.net* is sending the first part of the three-way handshake to a series of hosts on the *192.168.77.0* subnet, always trying to connect on port 80 (HTTP). This is a simple site scan for any Web servers (presumably listening on port 80) on this subnet.

```
13:21:45.012014 foo.example.com.1090 > 192.168.77.27.80: S 92946:92946(0) win 8192
13:21:45.013095 foo.example.com.1092 > 192.168.77.28.80: S 92932:92932(0) win 8192
13:21:45.014107 foo.example.com.1093 > 192.168.77.29.80: S 93094:93094(0) win 8192
13:21:45.015865 foo.example.com.1095 > 192.168.77.30.80: S 93016:93016(0) win 8192
13:21:45.016763 foo.example.com.1096 > 192.168.77.31.80: S 93106:93106(0) win 8192
13:21:45.018001 foo.example.com.1097 > 192.168.77.32.80: S 93076:93076(0) win 8192
13:21:45.018456 foo.example.com.1100 > 192.168.77.33.80: S 93154:93154(0) win 8192
13:21:45.018997 foo.example.com.1102 > 192.168.77.34.80: S 93280:93280(0) win 8192
```

Listing 4B

Scenario #5

Listing 5 shows a number of segments from host *foo.example.com* to a series of hosts on the *172.16.0.0* and *192.168.0.0* subnets, all probing port 23 (telnet). What is odd about these packets is that *foo* uses the same source port number (45820) when attempting a connection with the *192.168.0.0* hosts and the same port number (52526) when attempting to connect to a *172.16.0.0* host. In addition, note that these connection attempts all include four bytes of data. Although nothing in the TCP specification prohibits the transmission of data during the three-way handshake, it is very unusual to see data during connection setup. The TCP specification suggests that such data be buffered and not processed until after the connection is completed although this is a way to slip data past filtering and logging in weak implementations.

```
foo.example.com.45820 > 192.168.209.5.23: S 4195942931:4195942935(4) win 4096
foo.example.com.45820 > 192.168.216.5.23: S 4195944723:4195944727(4) win 4096
foo.example.com.52526 > 172.16.68.5.23: S 357331986:357331990(4) win 4096
foo.example.com.45820 > 192.168.183.5.23: S 4196001810:4196001814(4) win 4096
foo.example.com.52526 > 172.16.248.5.23: S 357312531:357312535(4) win 4096
foo.example.com.45820 > 192.168.205.5.23: S 4196007442:4196007446(4) win 4096
foo.example.com.52526 > 172.16.250.5.23: S 357313043:357313047(4) win 4096
foo.example.com.52526 > 172.16.198.5.23: S 357365266:357365270(4) win 4096
foo.example.com.52526 > 172.16.161.5.23: S 357355794:357355798(4) win 4096
```

Listing 5

Scenario #6

Listing 6A shows two sets of connection attempts to port 22 (Secure Shell, SSH) at host *watson*; the first from host *state.example.net* and the next from *foo.example.com*. This is a sequence of connection attempts and retries,

[7] Ftp, telnet, and nntp are the names of the services associated with TCP ports 21, 23, and 119, respectively.

evidenced by the fact that both *state* and *foo* repeatedly try to connect using the same source port number (1739 and 62555, respectively). In this case, *watson* is busy and does not respond to any of the connection attempts.

It is interesting, however, to look at the timing of the connection requests. The host *state* tries to connect to *watson*; after no response, it tries again three more times waiting roughly 1.5 seconds between each attempt. The host *foo* is not so persistent; after failing to connect the first time, it waits roughly 3 seconds to try again, then 6 seconds for the third attempt, and then 12 seconds for the final attempt. Clearly, *foo*'s implementers decided that if a connection could not be achieved at a given moment, the destination host might actually be busy.

In this case, *state* is a Windows system and *foo* is a Linux system.

```
10:08:23.472378 state.example.net.1739 > watson.22: S 72549644:72549644(0) win 8192 (DF)
10:08:25.009256 state.example.net.1739 > watson.22: S 72549644:72549644(0) win 8192 (DF)
10:08:26.504518 state.example.net.1739 > watson.22: S 72549644:72549644(0) win 8192 (DF)
10:08:28.006168 state.example.net.1739 > watson.22: S 72549644:72549644(0) win 8192 (DF)

17:14:18.726864 foo.example.com.62555 > watson.22: S 20583734:20583734(0) win 8192 <mss 1380>
(DF)
17:14:21.781140 foo.example.com.62555 > watson.22: S 20583734:20583734(0) win 8192 <mss 1380>
(DF)
17:14:27.776662 foo.example.com.62555 > watson.22: S 20583734:20583734(0) win 8192 <mss 1380>
(DF)
17:14:39.775929 foo.example.com.62555 > watson.22: S 20583734:20583734(0) win 8192 <mss 1380>
(DF)
```

Listing 6A

By way of contrast, observe the action in Listing 6B (as in Listing 6A, we only see segments that are attempting to establish TCP connections). In this case, host *foo* is making four attempts to connect to port 80 at *watson* over a period of a few seconds. Note that each attempt by *foo* is using a different source port number. What is being shown here is actually a normal event; *foo* is, presumably, downloading a Web page from *watson* and needs to open multiple TCP connections to access different files associated with the page, such as images or other multimedia. (Note that some implementations of TCP/IP permit only a limited number of simultaneously open port 80 connections.)

```
16:03:40.763603 foo.example.com.39344 > watson.80: S 523285584:523285584(0) win 8760 (DF)
16:03:41.919170 foo.example.com.39345 > watson.80: S 523517577:523517577(0) win 8760 (DF)
16:03:42.348706 foo.example.com.39358 > watson.80: S 526418601:526418601(0) win 8760 (DF)
16:03:42.491895 foo.example.com.39359 > watson.80: S 526509044:526509044(0) win 8760 (DF)
```

Listing 6B

Scenario #7

Listing 7A demonstrates *tcpdump*'s ability to filter the display; in this case, it will only show messages associated with the ICMP protocol. This listing shows a series of Echo Reply messages and there are several noteworthy items.

Echo Reply messages are commonly used with the Ping command, where one host sends an ICMP Echo message containing a string of characters to another host and the receiving host responds with an Echo Reply message repeating the same string of characters. The listing below displays both directions of ICMP traffic; it is highly suspicious to see a series of outgoing Echo Replies with no incoming Echo messages.

Another interesting aspect of the listing is the timing of the messages. The second message follows the first by approximately 10 seconds, followed by another message 51 seconds later, and the pattern of 10 second and 51 second interval pairs seems to continue. This is clearly the action of some automated process and not of a human.

```
[root@altamont gck]# tcpdump 'icmp'
12:03:36.016502 doggie.example.edu > 192.0.2.7: icmp: echo reply (DF)
12:03:46.016502 doggie.example.edu > 192.0.2.7: icmp: echo reply (DF)
12:04:37.016502 doggie.example.edu > 192.0.2.7: icmp: echo reply (DF)
12:04:47.006502 doggie.example.edu > 192.0.2.7: icmp: echo reply (DF)
12:05:38.016502 doggie.example.edu > 192.0.2.7: icmp: echo reply (DF)
12:05:48.016502 doggie.example.edu > 192.0.2.7: icmp: echo reply (DF)
12:06:39.006502 doggie.example.edu > 192.0.2.7: icmp: echo reply (DF)
12:06:49.006502 doggie.example.edu > 192.0.2.7: icmp: echo reply (DF)
12:07:40.006502 doggie.example.edu > 192.0.2.7: icmp: echo reply (DF)
12:07:50.006502 doggie.example.edu > 192.0.2.7: icmp: echo reply (DF)
12:08:41.006502 doggie.example.edu > 192.0.2.7: icmp: echo reply (DF)
12:08:51.006502 doggie.example.edu > 192.0.2.7: icmp: echo reply (DF)
```

```
12:09:42.016502 doggie.example.edu > 192.0.2.7: icmp: echo reply (DF)
12:09:52.016502 doggie.example.edu > 192.0.2.7: icmp: echo reply (DF)
12:10:43.016502 doggie.example.edu > 192.0.2.7: icmp: echo reply (DF)
12:10:53.016502 doggie.example.edu > 192.0.2.7: icmp: echo reply (DF)
12:11:44.016502 doggie.example.edu > 192.0.2.7: icmp: echo reply (DF)
```

Listing 7A

Listing 7B uses a *tcpdump* switch to display the data in hexadecimal, an option invoked here in order to more closely examine the messages. The 20 bytes starting with 0x4500 and ending with 0x9307 are the IP packet header; this is followed by eight bytes of ICMP message header (0x0000-9ca3-1a1a-0000). The remainder of the display shows the contents of the Echo Reply message.

Embedded inside the string of zeroes is the value 0x736b-696c-6c7a. If these characters are interpreted according to International Alphabet 5 (IA5), we retrieve the character string *skillz*. This is a well-known signature of the Stacheldraht distributed denial-of-service (DDoS) tool. Further analysis of Stacheldraht shows that ICMP Echo Reply messages are the mechanism for communication between the DDoS handlers and agents (Dittrich, 1999).

This traffic, then, shows a host (*doggie*) that has been compromised with Stacheldraht.

```
[root@altamont gck]# tcpdump 'icmp' -x
12:27:09.016502 doggie.example.edu > 192.0.2.7: icmp: echo reply (DF)
                        4500 0414 0000 4000 4001 cdf9 c670 431e
                        cc59 9307 0000 9ca3 1a0a 0000 0000 0000
                        0000 0000 0000 0000 0000 0000 0000 0000
                        736b 696c 6c7a 0000 0000 0000 0000 0000
                        0000 0000 0000 0000 0000 0000 0000 0000
                        0000
12:28:00.016502 doggie.example.edu > 192.0.2.7: icmp: echo reply (DF)
                        4500 0414 0000 4000 4001 cdf9 c670 431e
                        cc59 9307 0000 9ca3 1a0a 0000 0000 0000
                        0000 0000 0000 0000 0000 0000 0000 0000
                        736b 696c 6c7a 0000 0000 0000 0000 0000
                        0000 0000 0000 0000 0000 0000 0000 0000
                        0000
12:28:10.016502 doggie.example.edu > 192.0.2.7: icmp: echo reply (DF)
                        4500 0414 0000 4000 4001 cdf9 c670 431e
                        cc59 9307 0000 9ca3 1a0a 0000 0000 0000
                        0000 0000 0000 0000 0000 0000 0000 0000
                        736b 696c 6c7a 0000 0000 0000 0000 0000
                        0000 0000 0000 0000 0000 0000 0000 0000
                        0000
```

Listing 7B

Scenario #8

Listing 8A shows a sequence of packet fragments. There are two noteworthy items. First, Echo messages generally contain a small amount of data; 32, 56, and 64 bytes are typical values. This Echo message is clearly much larger. Second, note that the last fragment contains 1480 bytes starting at byte offset 65,120. This would result in an IP packet that is 66,600 bytes in length, greater than the maximum IP packet length of 65,535 bytes. This is the so-called Ping of Death; well-written IP kernels will merely discard the overly long IP packet but weaker implementations will crash the computer when such packets are detected (Insecure.Org, n.d.).

```
01:14:10.016000 foo.example.com > watson: icmp: echo request (frag 56980:1480@0+)
01:14:10.018000 foo.example.com > watson: (frag 56980:1480@1480+)
01:14:10.026000 foo.example.com > watson: (frag 56980:1480@2960+)
01:14:10.032000 foo.example.com > watson: (frag 56980:1480@4440+)
01:14:10.038000 foo.example.com > watson: (frag 56980:1480@5920+)
                         :
                         :
01:14:11.056000 foo.example.com > watson: (frag 56980:1480@59200+)
01:14:11.062000 foo.example.com > watson: (frag 56980:1480@60680+)
01:14:11.070000 foo.example.com > watson: (frag 56980:1480@62160+)
01:14:11.072000 foo.example.com > watson: (frag 56980:1480@63640+)
01:14:11.080000 foo.example.com > watson: (frag 56980:1480@65120)
```

Listing 8A

Listing 8B shows another IP fragmentation attack. In this case, the first packet contains the first 36 bytes of a UDP datagram. The second fragment contains the four bytes occupying byte offsets 24-27 in the original packet. Clearly, the second fragment overlaps the first. This is the so-called Teardrop attack; as above, most IP kernels handle this situation one way or another (i.e., they either overwrite the old information or ignore the new information) but weak implementations crash the host computer (CERT, 1998).

```
09:42:27.826000 foo.example.com.137 > watson.137: udp 28 (frag 242:36@0+)
09:42:27.828000 foo.example.com > watson: (frag 242:4@24)
```

Listing 8B

Scenario #9

This listing is one of the author's favorites and is an object lesson for students of TCP/IP. Note that this listing shows well less than a second of a large number of packets between a single pair of hosts; the listing actually went on continuously, at a high rate of speed and volume of data.

In this case, the author was using a host named *alabaster*. In an effort to observe SSH traffic between two hosts, the author initiated an SSH connection to host *altamont.gck.net* and then started *tcpdump*. Starting the packet sniffer, however, generated traffic that needed to be displayed remotely which, in turn, generated even more traffic. Essentially, the author started a *self denial-of-service*. What is learned is that one must take care when monitoring traffic across remote links.

```
alabaster:~ $ ssh root:secret@altamont.gck.net

[root@altamont gck]# tcpdump
09:31:15.747498 altamont.gck.net.ssh > granite.example.net.841: P 691789802:691789850(48) ack
3378360866 win 8576 <nop,nop,timestamp 35003172 37891099> (DF) [tos 0x10]
09:31:15.747646 altamont.gck.net.ssh > granite.example.net.841: P 48:208(160) ack 1 win 8576
<nop,nop,timestamp 35003172 37891099> (DF) [tos 0x10]
09:31:15.968039 granite.example.net.841 > altamont.gck.net.ssh: . 1:1(0) ack 208 win 8760
<nop,nop,timestamp 37891100 35003172> (DF) [tos 0x10]
09:31:15.968103 altamont.gck.net.ssh > granite.example.net.841: P 208:1360(1152) ack 1 win 8576
<nop,nop,timestamp 35003194 37891100> (DF) [tos 0x10]
09:31:15.968610 altamont.gck.net.ssh > granite.example.net.841: P 1360:1712(352) ack 1 win 8576
<nop,nop,timestamp 35003194 37891100> (DF) [tos 0x10]
09:31:15.968904 altamont.gck.net.ssh > granite.example.net.841: P 1712:1920(208) ack 1 win 8576
<nop,nop,timestamp 35003194 37891100> (DF) [tos 0x10]
09:31:16.029210 granite.example.net.841 > altamont.gck.net.ssh: . 1:1(0) ack 208 win 8760
<nop,nop,timestamp 37891100 35003172> (DF) [tos 0x10]
09:31:16.029303 altamont.gck.net.ssh > granite.example.net.841: P 1920:2128(208) ack 1 win 8576
<nop,nop,timestamp 35003200 37891100> (DF) [tos 0x10]
09:31:16.029224 granite.example.net.841 > altamont.gck.net.ssh: . 1:1(0) ack 208 win 8760
<nop,nop,timestamp 37891100 35003172> (DF) [tos 0x10]
09:31:16.029247 granite.example.net.841 > altamont.gck.net.ssh: . 1:1(0) ack 1920 win 7048
<nop,nop,timestamp 37891100 35003194> (DF) [tos 0x10]
09:31:16.029979 altamont.gck.net.ssh > granite.example.net.841: P 2128:2800(672) ack 1 win 8576
<nop,nop,timestamp 35003200 37891100> (DF) [tos 0x10]
09:31:16.030289 altamont.gck.net.ssh > granite.example.net.841: P 2800:3008(208) ack 1 win 8576
<nop,nop,timestamp 35003200 37891100> (DF) [tos 0x10]
09:31:16.067973 granite.example.net.841 > altamont.gck.net.ssh: . 1:1(0) ack 2128 win 8760
<nop,nop,timestamp 37891100 35003200> (DF) [tos 0x10]
09:31:16.068046 altamont.gck.net.ssh > granite.example.net.841: P 3008:3216(208) ack 1 win 8576
<nop,nop,timestamp 35003204 37891100> (DF) [tos 0x10]
09:31:16.068168 granite.example.net.841 > altamont.gck.net.ssh: . 1:1(0) ack 3008 win 7880
<nop,nop,timestamp 37891100 35003200> (DF) [tos 0x10]
09:31:16.068651 altamont.gck.net.ssh > granite.example.net.841: P 3216:3728(512) ack 1 win 8576
<nop,nop,timestamp 35003204 37891100> (DF) [tos 0x10]
09:31:16.068951 altamont.gck.net.ssh > granite.example.net.841: P 3728:3936(208) ack 1 win 8576
<nop,nop,timestamp 35003204 37891100> (DF) [tos 0x10]
09:31:16.069236 altamont.gck.net.ssh > granite.example.net.841: P 3936:4144(208) ack 1 win 8576
<nop,nop,timestamp 35003204 37891100> (DF) [tos 0x10]
```

Listing 9

CONCLUSION

Most computer forensics examiners and digital investigators enter the field by gaining expertise in computer hardware and software. This is consistent with the historic realm of *computer forensics*, which used to be totally about examining and analyzing a hard drive that had been removed from a computer.

The growth of network investigations and live traffic acquisition demands that a cadre of examiners be educated in network protocols and their operation. Today, nearly every computer exam needs to include at least the consideration of live data acquisition, be it memory or network traffic. Computers -- even at residential scenes -- have not been stand-alone devices in some years; awareness of network-attached devices ranging from other computers and wireless devices to printers and additional peripherals is a must in any seizure (Casey, 2004; Nikkel, 2005). The Internet, of course, is the scene of any number of cybercrimes, network attacks, and other information security events (Kessler & Fasulo, 2007). And since TCP/IP is the *lingua franca* of networking, a firm understanding of its functions is essential before examiners can begin to detect and investigate nefarious activity on the local net and the global Net.

This paper is neither a TCP/IP tutorial nor a course in intrusion detection and analysis. It is intended to demonstrate the need to pay closer attention to networks and protocols and to whet the appetite of those examiners who are beginning to recognize the need for this skill within their own practices.

ACKNOWLEDGEMENTS

The author is partially supported by Grant No. 2006-DD-BX-0282 awarded by the Bureau of Justice Assistance. The Bureau of Justice Assistance is a component of the Office of Justice Programs, which also includes the Bureau of Justice Statistics, the National Institute of Justice, the Office of Juvenile Justice and Delinquency Prevention, and the Office for Victims of Crime. Points of view or opinions in this document are those of the author and do not represent the official position or policies of the United State Department of Justice.

ABOUT THE AUTHOR

Gary C. Kessler, Ed.S., CCE, CISSP is director of the Champlain College Center for Digital Investigation (C3DI) and an associate professor and director of the Computer & Digital Forensics program at Champlain College in Burlington, Vermont, and an adjunct associate professor at Edith Cowan University in Mount Lawley, Western Australia. He is a member of the High Technology Crime Investigation Association (HTCIA) and International Society of Forensic Computer Examiners (ISFCE). Kessler is also a technical adviser to the Vermont Internet Crimes Against Children (ICAC) and Internet Crimes Task Forces, a member of the editorial board of the *Journal of Digital Forensics, Security and Law*, an associate editor of the *Journal of Digital Forensic Practice*, and a principal in GKS Digital Services, LLC.

REFERENCES

Casey, E. (2004). Network traffic as a source of evidence: Tool strengths, weaknesses, and future needs. *Digital Investigation*, *1*(1): 28-43.

CERT. (1998, May 26). *IP denial-of-service attacks*. CERT Advisory CA-1997-28. Retrieved September 16, 2007, from http://www.cert.org/advisories/CA-1997-28.html

Chappell, L.A., & Tittel, E. (2007). *Guide to TCP/IP* (3rd ed.). Boston: Thomson Course Technology.

Dittrich, D. (1999, December 31). *The "stacheldraht" distributed denial of service attack tool*. University of Washington. Retrieved September 16, 2007, from http://staff.washington.edu/dittrich/misc /stacheldraht.analysis

Insecure.Org. (n.d.). *Ping of death*. Retrieved September 16, 2007, from http://insecure.org/sploits/ping-o-death.html

Kessler, G.C. (2006). TCP/IP pocket reference guide. Retrieved September 16, 2007, from http://digitalforensics.champlain.edu/public/tcpip_prg.pdf

Kessler, G.C. (2007, January 16). An overview of TCP/IP protocols and the Internet. Retrieved September 16, 2007, from http://www.garykessler.net/library/tcpip.html

Kessler, G.C., & Fasulo, M. (2007). The case for teaching network protocols to computer forensics examiners. In G. Dardick (ed.), *Proceedings of the Conference on Digital Forensics, Security and Law*, April 18-20, Arlington, VA, pp. 115-137. Retrieved September 16, 2007, from http://www.garykessler.net/library/CDFSL_network_analysis.pdf

Nikkel, B.J. (2005). Generalizing sources of live network evidence. *Digital Investigation*, *2*(3): 193-200.

Northcutt, S., Cooper, M., Fearnow, M., & Frederick, K. (2001). *Intrusion signatures and analysis*. Indianapolis: New Riders.

Northcutt, S., Novak, J., & McLachlan, D. (2001). *Network intrusion detection: An analyst's handbook* (2nd ed.). Indianapolis: New Riders.

Stevens, W.R. (1994). *TCP/IP illustrated, Volume 1: The protocols*. Reading, MA: Addison-Wesley.

APPENDIX

Interpreting tcpdump/WinDump Output

This appendix will describe the generic elements in the output of tcpdump/WinDump. It is assumed that the reader is already familiar with the TCP/IP protocols. For more information on TCP/IP, see Chappell & Tittel (2007), Kessler (2007), or Stevens (1994). A TCP/IP protocol pocket reference guide can be found at Kessler (2006).

INTERPRETING TCP

The following line is a sample TCP segment:

```
12:12:32.912328 holmes.21 > altamont.champlain.edu.1074: S 3565913320:3565913320(0) ack 1 win
8760 <mss 1460> (DF)
```

- `12:12:32.912328` -- Timestamp
- `holmes.21` -- Source host (holmes) and port number (21)
- `altamont.champlain.edu.1074` -- Destination host (altamont.champlain.edu) and port number (1074)
- `S` -- Indicates that certain TCP flags are set; F = finish, P = push, R = reset, S = syn, and . = neither of these four flags
- `3565913320:3565913320(0)` -- Sequence number of first byte of data (3565913320), sequence number of next byte of data to send (3565913320), and total number of data bytes with this segment (0). In the initial setup segments, sequence numbers are absolute; once data exchange starts, the relative sequence numbers are usually displayed, starting with 1
- `ack 1` -- TCP Acknowledgement field (i.e., next expected data sequence number) value (relative sequence number 1); implies also that the TCP ack flag is set
- `win 8760` -- TCP Window field value (8760)
- `<mss 1460>` -- TCP setup options (in this case, Maximum Segment Size = 1460)
- `(DF)` -- IP flag settings (in this case, the Don't Fragment flag is ON)

INTERPRETING UDP

The following is a sample UDP datagram:

```
01:13:57.992080 foo.example.com.137 > watson.137: udp 28
```

- `01:13:57.992080` -- Timestamp
- `foo.example.com.137` -- Source host (foo.example.com) and port (137)
- `watson.137` -- Destination host (watson) and port (137)
- `udp` -- Protocol
- `28` -- Length of data in UDP datagram

INTERPRETING ICMP

The following is a sample ICMP message:

```
12:04:47.006502 doggie.example.edu > 192.0.2.7: icmp: echo reply (DF)
```

`12:04:47.006502` -- Timestamp

`doggie.example.edu` -- Source host

`192.0.2.7` -- Destination host (IP address)

`icmp` -- Protocol

`echo reply` -- ICMP message type

`(DF)` -- IP flag settings (in this case, the Don't Fragment flag is ON)

INTERPRETING FRAGMENTED PACKETS

IP packets can be up to 65,535 bytes in length. Few, if any, networks will transport a single packet of that size so long packets may be fragmented into smaller ones. In order for the fragments to be reassembled, IP packets contain a Fragment Identifier value and an Offset value, indicating where in the original packet the current data appears.

```
01:14:10.016000 foo.example.com > watson: icmp: echo request (frag 56980:1480@0+)
01:14:10.018000 foo.example.com > watson: (frag 56980:1480@1480+)
01:14:10.026000 foo.example.com > watson: (frag 56980:100@2960)
```

The listing above shows how *tcpdump* shows fragmented packets. In this case, the original packet was an ICMP Echo message containing 3060 bytes and a packet identification of 56,980. It has been fragmented into three smaller packets with a limit of 1480 bytes each.

The first packet identifies this as an Echo message containing 1480 bytes starting at the beginning (byte 0) of the original packet. The plus sign ("+") indicates that there is another fragment to follow.

The second packet contains 1480 bytes of data starting at byte position 1480 in the original packet; note that there is no information here about the type of higher layer data because that information would not be available beyond the data header in the content. Again, the plus sign indicates another fragment.

The final packet contains 100 bytes of data starting at byte position 2960 in the original packet. The absence of the plus sign indicates that this is the final fragment.

CITATION:

Kessler, G.C. (2008, March). On teaching TCP/IP protocol analysis to computer forensics examiners. *Journal of Digital Forensic Practice*, *2*(1), 43-53.