Interpreting Network Traffic:

A Network Intrusion Detector's Look

at Suspicious Events

by
Richard Bejtlich, TaoSecurity
richard@bejtlich.net
www.bejtlich.net

v 2.8 14 May 00

The purpose of this paper is to discuss interpretations of selected network traffic events from the viewpoint of a network intrusion detection analyst. (I define an "event" as any TCP/IP-based network traffic which prompts an analyst to investigate further. Generally, a suspicion that traffic has an abnormal or malicious character should prompt a closer look.) I assume the analyst has no knowledge of the source of the event outside of the data collected by his network-based intrusion detection system (NIDS) or firewall logs. I do not concentrate on the method by which these events are collected, but I assume it is possible to obtain data in TCPDump format. Using this standard allows a consistent presentation and interpretation of network traffic.

I thank Steven Northcutt for writing Network Intrusion Detection: An Analyst's Handbook. His work prompted me to analyze my own IDS output more closely, resulting in the traces you see today. I must also thank my coworkers for sharing their technical expertise and for reviewing this paper.

The Problem

Network intrusion detection is part art and part science. When confronted by abnormal network traffic, one must answer several questions:

- What could cause the traffic to be generated?
- What did the NIDS miss?
- How does reality differ from textbooks?
- Should I share with the community?

Since few people in the network security field are "experts," answering several of these questions requires a combination of creativity and logic. Thinking creatively helps imagine what sort of activity might have generated the traffic seen in his NIDS or firewall logs. Thinking logically assists the understanding of the actions necessary to generate suspicious traffic.

While the interpretation techniques explained here are pertinent to activity logged by a NIDS or firewall, I approach the subject from the NIDS angle. This my favorite subject, and I present this data with a warning: know the inner workings of your NIDS, or suffer frequent false positives and false conclusions.

For example, perhaps a NIDS records connections only to ports 21, 23, 25, and 80, because you run services on these ports. If the NIDS alerts you to attempted connections to these ports, it does not mean an intruder scanned those ports alone. He may have hit ports 0 to 1023, with the NIDS only seeing four attempted connections. Always wonder "what did the NIDS miss?" This question is at the heart of an excellent paper by Tim Newsham and Tom Ptacek, titled "Insertion, Evasion, and Denial of Service: Eluding Network Intrusion Detection," available at http://www.nai.com/media/ps/nai_labs/ids.ps .

Jonathan Skinner's summary is also worth perusing: http://www.nai.com/media/doc/nai_labs/ids-simple.doc .

Newsham and Ptacek remind us a NIDS may not be able to reconstruct events properly. From our earlier example, perhaps ports 21, 23, 25 and 80 were not the destination on the host; they could be the source ports of another system sending packets to us. However, being low ports, the NIDS might assume they are destination ports on our host. The NIDS then presents a reversed direction of traffic. (If your NIDS does not make these mistakes often, consider yourself fortunate and a smart selector of NIDS software!)

Having done network intrusion detection since December 98, I have learned the most interesting activity occurs below the level of detail offered by the NIDS console. Although many NIDS have improved collection, interpretation, and presentation functions, some traffic can best be understood at the packet trace level. Relying solely on the alerts show by the NIDS can lead to missed or misunderstood events. If the NIDS cannot show you packet-level action, the analyst is at the mercy of the NIDS engine's interpretation abilities.

A final goal of this paper is to promote the discussion of unrecognized traffic in the NIDS community. I present several events which could be seen at first glance as scanning or forms of reconnaissance. Without a collection of properly categorized network signatures, preferably TCPDump or Snoop-based, every new event forces analysts to "reinvent the wheel." (Note I prefer TCPDump as it was the format of choice for Richard Steven's TCP Illustrated volumes.) Should you disagree with my interpretations, I ask you to email me so we can discuss those differences. I am no expert but I do recognize the need to start a conversation among those concerned with network intrusion detection. I recommend perusing the arachNIDS database at `http://whitehats.com` .

The Tool

TCPDump is a utility which can help cut through the fog of mysterious traffic. It is a network monitoring program developed by the Lawrence Berkeley National Laboratory. It captures and reports traffic in a consistent and frequently enlightening way. You can get the UNIX version at `ftp://ftp.ee.lbl.gov/TCPDump.tar.Z` .

A team of students at the Italian Politecnico di Torino developed a Microsoft Windows 95/98/NT port of this program called Windump, available at `http://netgroup-serv.polito.it/windump` .

You can even use TCPDump to build a simple NIDS, as described by the Naval Surface Warfare Center Dahlgren at `http://www.nswc.navy.mil/ISSEC/CID/step.htm` .

You may also profit by examining the pioneering work done by the Network Flight Recorder and L0pht at `http://www.nfr.net` and `http://www.l0pht.com` .

TCPDump Format

A quick discussion of TCPDump output will help explain the traces which follow. I highlight interesting portions of the traces by starting with a short, standard, simple exchange of data via file transfer protocol.

[Note: All traces have been "sanitized" to remove the original IPs. Any similarity to IPs actually in use is purely coincidental. TCP service names are based on IANA's list at `http://www.isi.edu/in-notes/iana/assignments/port-numbers` . I assume working knowledge of the transmission control protocol. See the late Richard Stevens' "TCP/IP Illustrated, Volume 1: The Protocols" Thank you Mr. Stevens.]

Here is a packet-level conversation as seen by TCPDump, representing the TCP three-way handshake and an exchange of data. Note I have not run TCPDump with the -v (verbose) option, although I do so in selected traces which follow. For the purposes of this example, verbose information does not add significantly to the explanation. (Essentially, verbose data in later examples displays time to live and protocol id values.) I present the entire exchange first, with line-by-line analysis following.

```
1.  14:05:27.083238 ftp.client.org.1057 > ftp.server.edu.21:
     S 1484414:1484414(0) win 8192 <mss 536,nop,nop,sackOK> (DF)
2.  14:05:27.250587 ftp.server.edu.21 > ftp.client.org.1057:
     S 3037133697:3037133697(0) ack 1484415 win 9112 <mss 536> (DF)
3.  14:05:27.259810 ftp.client.org.1057 > ftp.server.edu.21:
     . ack 1 win 8576 (DF)
4.  14:05:27.402888 ftp.server.edu.47309 > ftp.client.org.113:
     S 3037242401:3037242401(0) win 8760 <mss 1460> (DF)
5.  14:05:27.412512 ftp.client.org.113 > ftp.server.edu.47309:
     R 0:0(0) ack 3037242402 win 0
6.  14:05:27.564336 ftp.server.edu.21 > ftp.client.org.1057:
     P 1:88(87) ack 1 win 9112 (DF) [tos 0x10]
7.  14:05:27.727461 ftp.client.org.1057 > ftp.server.edu.21:
     . ack 88 win 8489 (DF)
```

(seven packets deleted for clarity)

```
15.  14:05:35.774099 ftp.client.org.1057 > ftp.server.edu.21:
      P 31:37(6) ack 183 win 8394 (DF)
16.  14:05:35.895233 ftp.server.edu.21 > ftp.client.org.1057:
      P 183:197(14) ack 37 win 9112 (DF) [tos 0x10]
17.  14:05:35.903492 ftp.server.edu.21 > ftp.client.org.1057:
      F 197:197(0) ack 37 win 9112 (DF) [tos 0x10]
18.  14:05:35.906748 ftp.client.org.1057 > ftp.server.edu.21:
      . ack 198 win 8380 (DF)
19.  14:05:35.917049 ftp.client.org.1057 > ftp.server.edu.21:
      F 37:37(0) ack 198 win 8380 (DF)
20.  ?
```

The exchange has concluded, and I begin my explanation.

```
1.  14:05:27.083238 ftp.client.org.1057 > ftp.server.edu.21:
     S 1484414:1484414(0) win 8192 <mss 536,nop,nop,sackOK> (DF)
```

Line one shows an initiating time of 14:05:27.083238, which means 14 hours (2 pm), 05 minutes, and 27.083238 seconds. Packet transmission rate may help classify the activity as manually-inputted or computer-scripted. Packet type, combined with time, can help identify an event. The many hundreds of packets sent per second help define a SYN flood, which I discuss later.

We see ftp.client.org using port 1057 to connect to port 21 (ftp) at ftp.server.edu. Ports will play a crucial role in deciphering odd traces. In addition to trying to resolve any IP addresses listed, you should check the service name associated with any relevant ports. Port 1057 is not one of the well-known ports which can generally only be accessed as root (0-1023), but it does fall in the range of the registered ports (1024-49151), which can be accessed by most users and user processes. It is also in the range of the so-called "ephemeral" ports, from 1024 to 5000, from which many hosts initiate connections to well-know ports. As port 1057 does not have a service registered to it, it alone should not arouse suspicion.

"S" represents "SYN," or the synchronization flag in the TCP header. Setting the SYN flag, without other flags (like ACK, FIN, PUSH, etc.) shows this is the first step of the three-way handshake. Part of this first step is setting synchronization numbers. These numbers help each side of the conversation track the exchange of data. "1484414:1484414(0)" means the sending TCP stack is setting 1484414 as the initial synchronization number (ISN), and "0" (no) data is being passed in this packet. Although the numbers before and after the colon (:) are the same for this packet, in later packets they will be different and will have explanatory value. Richard Stevens (TCP/IP Illustrated Vol 1 p. 231) explains the format:

sequence number:implied ending sequence number (data)

However, as our traces will demonstrate, the format seems to be:

sequence number of first byte in packet:sequence number of first byte in NEXT packet (data)

TCPDump will only display the number:number (data) information for packets with more than 0 bytes of data or those setting the SYN, FIN, or RST flags.

Our initiating IP uses the ISN to begin counting bytes in the packets it sends to ftp.server.edu. Tracking the synchronization number used by the first observed packet may help identify malicious activity. Some tools use default synchronization numbers. In certain packets shown later, we see a host ACK 674719802 and 674711610; we assume they are responses to ISNs of 674719801 and 676711609 from an initiating host's SYN packet.

Of interest are the TCP available window size of 8192 bytes, the maximum segment size of 536 bytes, two "nop" options, and the "DF," or "don't fragment" option. The TCP window is a flow control mechanism which allows the sender to transmit multiple packets before stopping to wait for acknowledgements. Here ftp.client.org is advertising its window size of 8192 bytes to ftp.server.edu. Next, maximum segment size is advertised by the ftp client as 536 bytes. It is an admission by the client that its local network segment can accommodate a packet, without fragmentation, no larger than 536 bytes. 536 bytes is only the size of the data payload; the TCP and IP headers must still be added to the packet, and are assumed to occupy 40 bytes total.

Following are two "nop" options, which denote "no option." They are present to help ftp.client.org "pad" its TCP header to form four-byte fields. In this case, the MSS occupies four bytes (one byte for "kind=2," one byte for "len=4," and two bytes for the actual MSS value). "sackOK" denotes acceptance by the ftp client of the "selective acknowledgement" option, described in RFC 2018. Selective acknowledgement is a method allowing the data receiver to tell the sender which segments arrived successfully. This lets the sender retransmit only lost packets, in an attempt to improve upon TCP's cumulative acknowledgement process. Since this option occupies two bytes (one byte for "kind=4" and another for "len=2"), the two single-byte "nop" options round out the fields to two even four-byte sections. (The four byte value is significant, as it is the "width" of the standard TCP header.) Finally, the DF option means ftp.client.org is telling routers between itself and the ftp server not to fragment its packets. If the router cannot handle that request, due to its MTU being smaller than the packet, the router should return an ICMP error message to the client.

While innocent in this first packet, these options may be worth studying in other traces. You may see traffic scattered across several NIDS with little in common but the window sizes, maximum segments sizes, or other options. While certainly not indicative individually, taken collectively such clues might help correlate related events. (Although no data is passed in this packet, we will encounter a trace which attempts to send 64 bytes of data to another host. While unusual, it is not illegal per the TCP RFCs, and makes an excellent signature identifying element!)

```
2.  14:05:27.250587 ftp.server.edu.21 > ftp.client.org.1057:
     S 3037133697:3037133697(0) ack 1484415 win 9112 <mss 536> (DF)
```

The second packet is the ftp server's response, setting its own initial sequence number (actually an "initial response number") as 3037133697. The ftp server acknowledges our client's ISN by setting its "ack" flag and associating it with 1484415, which is the next sequence number it expects from the client, or essentially ISN + 1. The first byte of data sent by the client will be number 1484415. Notice we have used one sequence number already, 1484414, without sending any data. This "waste" of a sequence number will not be repeated, as all other sequence numbers will be used to indicate specific bytes sent during the TCP exchange.

Observe the different window and maximum segment sizes for the ftp server (i.e., 9112 bytes and 536 bytes, respectively), compared to the client system. While innocent here, they might help identify a scan or tool signature, since many packet-forging scripts will set these values manually. Notice that since the MSS option occupies four bytes by itself, no "nop" byte padders are needed.

```
3.  14:05:27.259810 ftp.client.org.1057 > ftp.server.edu.21:
     . ack 1 win 8576 (DF)
```

The third packet concludes the TCP three-way handshake with an acknowledgment by the client of the ftp server's initial response number. Note that this trace does not employ the TCPDump's -S option, which outputs absolute sequence numbers for each packet. These traces use relative sequence numbers, which make it easier to track the number of bytes passed. For example, packet three shows an "ack 1", with the 1 being the difference between the client's initial sequence number and the sequence number of packet three. In other words, the -S option would have printed 3037133698 here. Remember, the purpose of an ACK is to help track bytes exchanged. By ACKing 3037133698, (or 1 in relative terms), the client is telling the server "I expect the first byte you send me to be number 3037133698 (or 1 in relative terms.) The "." means neither the SYN, FIN, RST nor PSH flags are set.

(If the sequence number issue still puzzles you, the appendix includes a trace where absolute sequence numbers were used.)

```
4.  14:05:27.402888 ftp.server.edu.47309 > ftp.client.org.113:
     S 3037242401:3037242401(0) win 8760 <mss 1460> (DF)
5.  14:05:27.412512 ftp.client.org.113 > ftp.server.edu.47309:
     R 0:0(0) ack 3037242402 win 0
```

Packets four and five present an opportunity to discuss closed ports. The ftp server is attempt to use port 40739 connect to the client machine's port 113 (auth), for authentication purposes. Since the client's port 113 is closed, it responds with a reset "R", and acknowledges the server's sequence number 3037242401 by adding one to it, to make 3037242402. The server does not respond, since there is no need per the RFC. (A RST should never be ACKed.)

Data exchange follows with packets six and higher. I have deleted packets eight through fourteen, because they do not add anything new to our discussion.

```
15.  14:05:35.774099 ftp.client.org.1057 > ftp.server.edu.21:
      P 31:37(6) ack 183 win 8394 (DF)
16.  14:05:35.895233 ftp.server.edu.21 > ftp.client.org.1057:
      P 183:197(14) ack 37 win 9112 (DF) [tos 0x10]
```

Packets fifteen and sixteen are later stages of data transfer. Note the "P", or "push" flag. This tells the ftp server to "push" its queue of packets stored in the TCP/IP stack directly to the application above. The push from the ftp server to the ftp client in packet sixteen tells the ftp client to push its queue of packets up its stack, to the client application. The information "pushed" consists of all data in the segment containing the push flag, plus any data stored in the receiving TCP buffer. The presence of this flag is normal for an interactive session, such as a ftp connection. It may represent a command sent by the client; as the client usually waits for the server's response, it is logical for the client to request rapid processing of all data stored in the server's TCP buffer.

Although not seen in this paper, one may encounter the URG or "urgent" flag in other traces. This flag tells the receiving TCP stack that "urgent" data is present, and leaves the receiver to interpret it as it wishes. The telnet and rlogin applications typically use this flag to signal transmission of the interrupt key, while ftp uses urgent to signal aborting file transfer.

Packet sixteen conclude with the IP option "type of service," shown as [tos 0x10]. This particular value means "minimize delay." Other possible values are maximize throughput, maximize reliability, and minimize monetary cost, all of which are beyond the scope of this paper. I highly recommend Eric Hall's Internet Core Protocols.

Turning back to data flow, packet fifteen shows the ftp client sending 6 bytes of data, with a relative sequence number showing 36 total bytes sent during the entire TCP conversation. The next set of bytes sent to the server will begin with number 37. Here we see this format at work:

sequence number of first byte in packet:sequence number of first byte in NEXT packet (data)

In packet 15 the client sent bytes 31, 32, 33, 34, 35, and 36, and will send 37 next. By ACKing 183, the ftp client acknowledges receipt of 182 bytes from the server, and says it expects the next data from the server to begin with byte 183.

Packet sixteen shows the ftp server sending 14 bytes and acknowledging receipt of 36 bytes from the client, while expecting byte 37 next.

```
17.  14:05:35.903492 ftp.server.edu.21 > ftp.client.org.1057:
      F 197:197(0) ack 37 win 9112 (DF) [tos 0x10]
18.  14:05:35.906748 ftp.client.org.1057 > ftp.server.edu.21:
      . ack 198 win 8380 (DF)
19.  14:05:35.917049 ftp.client.org.1057 > ftp.server.edu.21:
      F 37:37(0) ack 198 win 8380 (DF)
20.  ?
```

Packet seventeen begins the process of closing the connection in a graceful manner, and introduces another TCP header flag. (Richard Stevens calls this "orderly release." In contrast, "abortive release" is the abrupt termination of a TCP connection, as caused by a host shutting down or loss of connectivity.) The server sends a packet with the F or "FIN" flag sent, indicating it has no more data to send to the client. The server also acknowledges the last byte sent by the client (37-1=36), and shows it has sent all bytes needed with the 37:37 (0) value.

Packet eighteen demonstrates that the session does not close gracefully until both sides agree. Here, the client acknowledges the server's FIN request. The client then sends its own FIN. According to Richard Stevens, we should see one last packet (number 20) from the server to the client, where the server acknowledges the client's FIN. We do not see that packet in this trace, which can remind us that some events do not correspond exactly to the logical models which we follow. I imagine that the packet was lost, or that the TCPDump ended abruptly.

Many of the traces in this paper and most scanning activity does not observe this graceful close process, and instead uses resets from the source host. This process is demonstrated below.

```
1.  11:48:02.545940 dialup.modem.net.1092 > 206.61.52.32.119:
     S 436537:436537(0) win 8192 <mss 536,nop,nop,sackOK> (DF)
2.  11:48:02.774748 host.news.org.119 > dialup.modem.net.1092:
     S 4231438324:4231438324(0) ack 436538 win 0
3.  11:48:02.784542 dialup.modem.net.1092 > host.news.org.119:
     . ack 1 win 8576 (DF)
4.  11:48:02.952477 host.news.org.119 > dialup.modem.net.1092:
     R 1:1(0) ack 1 win 0
```

Lines one to three are the standard three-way handshake associated with normal TCP connections. Line four, however shows the R or RST (reset) flag. This is a request by host.news.org to immediately reset the connection between itself and dialup.modem.net. No acknowledgement by dialup.modem.net occurs and none is required by the RFCs. Resets will be seen in upcoming traces quite often.

Let the Tracing Begin!

Let's start looking at malicious network activity by examining a scan which obeys TCP's three-way handshake -- the plain TCP connect scan. This scan type is old but will provide a baseline for some of the later traces. Any intrusion detection system should log this activity. (Whether the analyst reacts to it may be another matter!)

```
11:56:20.442740 connect.scanner.net.1141 > victim.cablemodem.com.21:
 S 929641:929641(0) win 8192 <mss 536,nop,nop,sackOK> (DF)
11:56:21.191786 victim.cablemodem.com.21 > connect.scanner.net.1141:
 S 779881634:779881634(0) ack 929642 win 8576 <mss 1460> (DF)
11:56:21.201490 connect.scanner.net.1141 > victim.cablemodem.com.21:
 . ack 1 win 8576 (DF)

11:56:23.954930 connect.scanner.net.1144 > victim.cablemodem.com.37:
 S 932103:932103(0) win 8192 <mss 536,nop,nop,sackOK> (DF)
11:56:24.647238 victim.cablemodem.com.37 > connect.scanner.net.1144:
 R 0:0(0) ack 1 win 0
```

The first trace shows a successful three-way handshake between the scanning host and the unsuspecting victim; this means port 21 (ftp) is open. The second trace shows the scanner's SYN being met by a reset, meaning port 37 (time) is closed on the victim's machine.

Contrast that activity with the traces below.

```
10:22:45.030552 half.scanner.net.49724 > victim.cablemodem.com.21:
  S 2421827136:2421827136(0)
10:22:45.030552 victim.cablemodem.com.21 > half.scanner.net.49724:
  S 4046313668:4046313668(0) ack 2421827137
10:22:45.030552 half.scanner.net.49724 > victim.cablemodem.com.21:
  R 2421827137:2421827137(0)

10:22:45.050552 half.scanner.net.49724 > victim.cablemodem.com.37:
 S 2418821749:2418821749(0)
10:22:45.050552 victim.cablemodem.com.37 > half.scanner.net.49724:
 R 0:0(0) ack 2418821750
```

This is a TCP SYN, or "half connect" scan. The scanner sends a reset to any port reported as open by the victim, rather than continue with the three-way handshake. The original purpose of this scan was to evade a NIDS which only logged connections completing the three-way handshake, like the TCP connect scan. All newer NIDS should catch this activity.

In an effort to evade newer NIDS, some scanner programmers have tried other tactics. Consider this trace:

```
21:03:59.711106 snap 0:0:0:8:0 scanner.net.53 > host201.victim.org.21:
 F 0:0(0) win 2048 (ttl 48, id 55097)
21:04:05.738307 snap 0:0:0:8:0 scanner.net.53 > host201.victim.org.21:
 F 0:0(0) win 2048 (ttl 48, id 50715)
21:05:10.399065 snap 0:0:0:8:0 scanner.net.53 > host202.victim.org.21:
 F 0:0(0) win 3072 (ttl 49, id 32642)
21:05:16.429001 snap 0:0:0:8:0 scanner.net.53 > host202.victim.org.21:
 F 0:0(0) win 3072 (ttl 49, id 31501)
21:09:12.202997 snap 0:0:0:8:0 scanner.net.53 > host24.victim.org.21:
 F 0:0(0) win 2048 (ttl 52, id 47689)
21:09:18.215642 snap 0:0:0:8:0 scanner.net.53 > host24.victim.org.21:
 F 0:0(0) win 2048 (ttl 52, id 26723)
21:10:22.664153 snap 0:0:0:8:0 scanner.net.53 > host3.victim.org.21:
 F 0:0(0) win 3072 (ttl 53, id 24838)
21:10:28.691982 snap 0:0:0:8:0 scanner.net.53 > host3.victim.org.21:
 F 0:0(0) win 3072 (ttl 53, id 25257)
21:11:10.213615 snap 0:0:0:8:0 scanner.net.53 > host102.victim.org.21:
 F 0:0(0) win 4096 (ttl 58, id 61907)
21:11:10.227485 snap 0:0:0:8:0 host102.victim.org.21 > scanner.net.53:
 R 0:0(0) ack 4294947297 win 0 (ttl 25, id 38400)
```

First, note this trace was produced using TCPDump's verbose option, where "snap 0:0:0:8:0" refers to the subnetwork access protocol. SNAP is a method of differentiating between non-OSI network layer protocols. "0:0:0" represents a three-byte organizational unit identifier, which for TCP/IP is 0x0. "8:0" represents a two-byte Ethertype, which for Ethernet is 0x800. (Looking at the SNAP byte-by-byte, we have the OUI as 0x0 : 0x0 : 0x0, with the Ethertype being 0x08 : 0x0.)

Let's look at this activity systematically, beginning with IPs:

- IPs: We see traffic from scanner.net to multiple hosts on the victim.org domain. Each IP is probed twice.

- Ports: The originating IP sends packets from port 53 (dns) to port 21 (ftp) on each system. Activity to TCP port 53 can usually be associated with DNS zone transfers or other resolution processes. (For example, responses to DNS queries via UDP cannot exceed 512 bytes. If the response is more than 512 bytes, a connection via TCP must be established. Therefore, legitimate DNS information exchange can occur over TCP channels.) The ftp port would be an attractive target, especially if the scanner is looking for an ftp server with anonymous logins.

- Flags: Most of the packets have the FIN flag set. This is not normal behavior. Unlike some of the activity we will discuss below, we cannot envision a network event which would generate these packets as an appropriate response. Therefore, they must have been specially crafted.

- Traffic direction/activity: Every packet save one is a FIN sent from scanner.net to a target host. The only difference is the R ACK reply by host102.victim.org. This indicates port 21 is *closed* on this host. The lack of a reply by any other host demonstrates two possibilities. First, the hosts may not exist. If ICMP is allowed to cross the security boundary, then perhaps the scanner will receive ICMP destination unreachable error messages, signifying non-existent target hosts. Second, the hosts may exist, and may each be running the ftp service. Per RFC 793, open ports should remain silent when receiving a lone FIN packet.

- Time: This is not an especially fast scan, but it is undoubtedly an automated event.

- Window size, TTL, and other features: Several other characteristics deserve attention. Window size values are 2048, 3072, and 4096 bytes for various packets. TTLs vary from 48 to 58, which is a wide margin. The IP ID numbers also vary, without apparent regularity. While it is difficult to discern patterns in this case, other traces may yield more recognizable results. (Thank you to Judy Novak for pointing out these features.)

- Bottom line: This event was a FIN scan, designed to evade some NIDS, which found a closed ftp port at host102.victim.org. Without knowing if the other hosts exist, and not seeing ICMP error messages, the scanner cannot decide if the other hosts are running the ftp service. I recommend considering these factors when making judgments about any network event you investigate.

Consider this traffic:

```
22:08:33.913622 cable.modem.net.23 > dns.one.org.23:
 . ack 499410217 win 1028 (ttl 30, id 39426)
22:08:33.915481 dns.one.org.23 > cable.modem.net.23:
 R 499410217:499410217(0) win 0 (ttl 254, id 34512)
22:08:33.954076 cable.modem.net.23 > dns.two.org.23:
 . ack 499410217 win 1028 (ttl 0, id 39426)
22:08:33.955461 dns.two.org.23 > cable.modem.net.23:
 R 499410217:499410217(0) win 0 (ttl 254, id 5962)
22:08:33.982753 cable.modem.net.143 > dns.one.org.143:
 . ack 499410217 win 1028 (ttl 30, id 39426)
22:08:33.983904 dns.one.org.143 > cable.modem.net.143:
 R 499410217:499410217(0) win 0 (ttl 254, id 34514)
22:08:34.061121 cable.modem.net.143 > dns.two.org.143:
 . ack 499410217 win 1028 (ttl 30, id 39426)
22:08:34.064069 dns.two.org.143 > cable.modem.net.143:
 R 499410217:499410217(0) win 0 (ttl 254, id 5967)

22:08:39.161874 cable.modem.net.1146 > dns.two.org.23:
 S 2585837477:2585837477(0) win 16324 <mss 1484,sackOK,timestamp
 50637 0,nop,wscale 0> (DF) (ttl 52, id 770)
22:08:39.170887 cable.modem.net.1147 > dns.two.org.143:
 S 2585589580:2585589580(0) win 16324 <mss 1484,sackOK,timestamp
 50637 0,nop,wscale 0> (DF) (ttl 52, id 771)
22:08:39.182221 cable.modem.net.1149 > dns.two.org.111:
 S 2583756762:2583756762(0) win 16324 <mss 1484,sackOK,timestamp
 50637 0,nop,wscale 0> (DF) (ttl 52, id 773)
22:08:39.183010 cable.modem.net.1151 > dns.two.org.79:
 S 2588862233:2588862233(0) win 16324 <mss 1484,sackOK,timestamp
 50637 0,nop,wscale 0> (DF) (ttl 52, id 775)
22:08:39.190551 cable.modem.net.1152 > dns.two.org.53:
 S 2590571910:2590571910(0) win 16324 <mss 1484,sackOK,timestamp
 50637 0,nop,wscale 0> (DF) (ttl 52, id 776)
22:08:39.212060 cable.modem.net.1153 > dns.two.org.31337:
 S 2585840391:2585840391(0) win 16324 <mss 1484,sackOK,timestamp
 50637 0,nop,wscale 0> (DF) (ttl 52, id 777)
22:08:39.224956 cable.modem.net.1157 > dns.two.org.21:
 S 2585906418:2585906418(0) win 16324 <mss 1484,sackOK,timestamp
 50637 0,nop,wscale 0> (DF) (ttl 52, id 781)

22:08:39.261488 cable.modem.net.1162 > dns.one.org.23:
 S 2589761527:2589761527(0) win 16324 <mss 1484,sackOK,timestamp
 50638 0,nop,wscale 0> (DF) (ttl 52, id 786)
22:08:39.264445 cable.modem.net.1163 > dns.one.org.143:
 S 2588328359:2588328359(0) win 16324 <mss 1484,sackOK,timestamp
 50638 0,nop,wscale 0> (DF) (ttl 52, id 787)
22:08:39.292663 cable.modem.net.1165 > dns.one.org.111:
 S 2590160130:2590160130(0) win 16324 <mss 1484,sackOK,timestamp
 50638 0,nop,wscale 0> (DF) (ttl 52, id 789)
22:08:39.305784 cable.modem.net.1167 > dns.one.org.79:
 S 2594918730:2594918730(0) win 16324 <mss 1484,sackOK,timestamp
 50638 0,nop,wscale 0> (DF) (ttl 52, id 791)
22:08:39.307131 cable.modem.net.1168 > dns.one.org.53:
 S 2582494230:2582494230(0) win 16324 <mss 1484,sackOK,timestamp
 50638 0,nop,wscale 0> (DF) (ttl 52, id 792)
22:08:39.307209 cable.modem.net.1169 > dns.one.org.31337:
 S 2590958670:2590958670(0) win 16324 <mss 1484,sackOK,timestamp
 50638 0,nop,wscale 0> (DF) (ttl 52, id 793)
22:08:39.344336 cable.modem.net.1173 > dns.one.org.21:
 S 2593455289:2593455289(0) win 16324 <mss 1484,sackOK,timestamp
 50638 0,nop,wscale 0> (DF) (ttl 52, id 797)
```

Again, systematically:

- IPs: cable.modem.net is concentrating on two hosts we monitor: dns.one.org and dns.two.org. Although not shown, each host is hit with the second set of SYN packets three total times.

- Ports: The first half of the activity targets tcp ports 23 (telnet) and 143 (imap). The second half involves those ports plus 111 (SUN Remote Procedure Call, or portmapper), 79 (finger), 53 (dns), 31337 (Back Orifice tcp port), and 21 (ftp). All are of use to a potential intruder. Of more interest, perhaps, are the source ports involved. Note the stealthy nature of the first stage, where source port is set to destination port, in an attempt to confuse packet-filtering devices. The second stage is less cunning, but more analyst-friendly. Observe the orderly incrementation of ports used to contact dns.two.org, starting with 1146, then 1147, then 1149. Where is 1148? Most likely this packet was destined for a port not monitored by our NIDS. It was probably not lost, as the traffic to dns.two.org shows. Here, we see source port 1162, then 1163, then 1165 (another port missing!) Using this "gap-counting" technique, we can assume packets were sent to at least four ports not watched by our NIDS. This does not count the four "missing" ports between port 781 and 786, where attention shifts from dns.two.org to dns.one.org!

- Flags: The first half of the event involves no flags set, with RST ACK packets sent back from the targets. These initial packets do not occur naturally unless a preceded by the SYN / SYN ACK exchange of the three way handshake. The RST ACK packets are assumed to be returned from closed ports, as an open port would usually remain silent. (This is the default for the Linux TCP/IP stack, as documented by Vicki Irwin and Hal Pomeranz. Your mileage may vary.) Interestingly, the second half of the event shows only SYN packets sent, with zero replies. This may indicate the cablem.modem.net's initial packets, with the ACK bit set, successfully evaded a packet filtering device. This device, however, probably intercepted the later packets with the SYN bit set.

- Traffic direction/activity: All traffic seems to involve a prompt by cable.modem.net, followed by an indication that the target ports are closed.

- Time: The entire event elapses in six seconds, with an apparent five second delay between the ACK and SYN stages.

- Window size, TTL, and other features: We see a wide variance between the TTL 30 of stage one and TTL 52 of stage two. As these packets presumably come from the same host, we assume the tool generate the packets sets
initially TTLs differently for each technique. Stage one shows IP id values each forged to be 39426. This may provide a signature clue for future encounters with this tool. The IP id values increment nicely in stage two, matching the TCP port technique mentioned earlier. Window sizes for stage one (1028) contrast strongly with stage two (16324).

- Bottom line: We appear to have an ACK scan combined with some sort of SYN scan. The packet filtering device which allows ACK packets but prevents answers to the SYN packets keeps us from knowing more about stage two. This case emphasizes the need to understand the operation of your IDS, as it helped us to recognize the port "gaps" and their possible relevance to our investigation.

To Flood or Not to Flood

Now we turn to a core issue of this paper -- the SYN flood. Anyone unfamiliar with SYN floods would greatly benefit by reading Route's definitive work on the subject in Phrack 48. Essentially, a SYN flood is a denial of service attempt, where an attacker attempts to fill the backlog queue of a victim machine's TCP server. To prevent the victim from tearing down these memory-consuming connections, the attacker spoofs one or more source IPs, choosing IPs which presumably do not exist. The victim of a properly executed SYN flood cannot reply to the spoofed source(s), as the source(s) will not exist and therefore cannot clear the victim's potential connections. An
attacker might take these actions to attempt a TCP hijack, as Kevin Mitnick did against the rlogin port of a

machine owned by Tsutomu Shimomura. By shutting down the TCP service of a host trusted by Shimomura, Mitnick was
able to impersonate that host without it interfering in his communications with Shimomura's box.

A SYN flood consists of dozens of SYN packets sent from a spoofed source IP, or multiple spoofed source IPs, to a victim. Note the high frequency of packets sent:

```
11:46:14.212003 spoofed.ip.one.1053 > flood.victim.com.23:
 S 322286:322286(0) win 8192 <mss 536,nop,nop,sackOK> (DF)
11:46:14.598008 spoofed.ip.one.1054 > flood.victim.com.23:
 S 322286:322286(0) win 8192 <mss 536,nop,nop,sackOK> (DF)
11:46:14.975522 spoofed.ip.one.1055 > flood.victim.com.23:
 S 322286:322286(0) win 8192 <mss 536,nop,nop,sackOK> (DF)
```
etc...

The desperate victim tries to reply to the spoofed source IP. If the spoofed host truly does not exist, the victim is out of luck. But what if the spoofed source does exist? Below is what an intrusion detection analyst at a
site owning the spoofed IP might see, if the target port is open and behaves as traditionally expected:

```
11:46:14.765043 flood.victim.com.23 > spoofed.ip.one.1053:
 S 4137483508:4137483508(0) ack 322287 win 8192 <mss 1460>
11:46:14.891108 flood.victim.com.23 > spoofed.ip.one.1054:
 S 4164828806:4164828806(0) ack 322287 win 8192 <mss 1460>
11:46:15.019029 flood.victim.com.23 > spoofed.ip.one.1055:
 S 4192020032:4192020032(0) ack 322287 win 8192 <mss 1460>
```
etc...

Why would an you, an innocent bystander, witness such an act? The answer is: you own spoofed.ip.one, which may or may not exist! Why would anyone spoof your IPs? (Hint: do your routers or firewalls block ICMP?) In some cases, SYN flood utilities allow the attacker to select a range of IPs for the spoofed source, or it will generate its own list. The utility may ping that range, trying to determine if any hosts exist. If no ICMP echo replies are heard, the SYN flood program assumes the IPs do not exist and are ideal spoofed sources. However, if those IPs belong to hosts behind a router or firewall denying ICMP echo requests, they will not respond with ICMP echo replies. This "flaw" in choosing good spoofable IPs causes much of the "third party" traffic in this paper. Essentially, the site you monitor may become a third party to a SYN flood, by virtue of having blocked ICMP echo requests. A second possibility, which is demonstrated in actual code below, involves SYN flooding tools which randomly choose source IPs to spoof. In such cases, your network addresses could be selected without your involvement. This method of source IP selection appears to be the reason why I detected activity from two commonly seen ACK numbers, described later.

The preceding example appears straightforward. A single IP is spoofed, and the sender increments his source ports in an orderly manner (1053, 1054, 1055). The trace as seen by the innocent bystander shows the flood victim's open port 23 replying with SYN ACK packets, in an attempt to establish a TCP three-way handshake. What happens if the target port of a SYN flood is closed? The following was confirmed as a SYN flood by the author, who observed the traffic, contacted victim.isp.net, and learned the ISP was indeed SYN flooded on the date and time in question.

```
20:31:15.794717 victim.isp.net.68 > spoofed.ip.one.29470:
 R 0:0(0) ack 723645348 win 0 (ttl 242, id 40923)
20:31:20.190800 victim.isp.net.68 > spoofed.ip.one.48926:
 R 0:0(0) ack 960212644 win 0 (ttl 242, id 56829)
20:31:24.622496 victim.isp.net.68 > spoofed.ip.one.2846:
 R 0:0(0) ack 1196779940 win 0 (ttl 242, id 7276)
20:31:37.634797 victim.isp.net.68 > spoofed.ip.one.61214:
 R 0:0(0) ack 1906481828 win 0 (ttl 242, id 54177)
20:31:42.021607 victim.isp.net.68 > spoofed.ip.one.15134:
 R 0:0(0) ack 2143049124 win 0 (ttl 242, id 4485)

20:31:17.754903 victim.isp.net.77 > spoofed.ip.two.44376:
 R 0:0(0) ack 1861342051 win 0 (ttl 242, id 25377)
20:31:22.054453 victim.isp.net.77 > spoofed.ip.two.13400:
 R 0:0(0) ack 454770019 win 0 (ttl 242, id 40905)
20:31:26.394198 victim.isp.net.77 > spoofed.ip.two.47960:
 R 0:0(0) ack 1195681635 win 0 (ttl 242, id 56409)
20:31:39.370211 victim.isp.net.77 > spoofed.ip.two.20568:
 R 0:0(0) ack 1270932835 win 0 (ttl 242, id 38330)
20:31:43.735814 victim.isp.net.77 > spoofed.ip.two.55128:
 R 0:0(0) ack 2011844451 win 0 (ttl 242, id 54069)
```

Here we see a SYN flood directed against two closed ports, 68 (boot-p) and 77 (RJE private service). (Although many other ports were targeted, these two are shown as examples. Since spoofed.ip.one and spoofed.ip.two occupied separate class C networks, I chose to separate the two traces.) Observe the two closed ports return RST ACK packets to the spoofed source IPs. The ACK numbers appear random, as do the ports of the two spoofed IPs. Furthermore, a fairly high packet rate is seen. This may not always be true from the vantage point of the intrusion detection analyst. If the SYN flood tool does not spoof IPs which are all monitored by your IDS, you may not get a complete picture of the event. (And, from the perspective of the victim, more than one organization appears to be the originator of the attack!) For example:

```
05:23:07.968535 victim.isp.net.22 > spoofed.ip.one.10180:
 R 0:0(0) ack 1 win 0 (ttl 53, id 57295)
05:23:55.594577 victim.isp.net.23 > spoofed.ip.two.64876:
 R 0:0(0) ack 1 win 0 (ttl 53, id 62163)
05:27:36.116580 victim.isp.net.23 > spoofed.ip.three.48279:
 R 0:0(0) ack 1 win 0 ttl 53, id 18796)
05:32:34.963053 victim.isp.net.23 > spoofed.ip.four.55483:
 R 0:0(0) ack 1 win 0 (ttl 53, id 48851)
05:33:01.308930 victim.isp.net.23 > spoofed.ip.five.48412:
 R 0:0(0) ack 1 win 0 (ttl 53, id 51512)
05:35:12.400935 victim.isp.net.22 > spoofed.ip.six.57306:
 R 0:0(0) ack 1 win 0 (ttl 53, id 64704)
05:36:40.823582 victim.isp.net.22 > spoofed.ip.seven.46819:
 R 0:0(0) ack 1 win 0 (ttl 53, id 8090)
05:38:50.740540 victim.isp.net.23 > spoofed.ip.eight.29217:
 R 0:0(0) ack 1 win 0 (ttl 53, id 21089)
```

This trace shows several seconds elapsing between each packet as a malicious Internet user spoofs our IPs, SYN flooding ports 22 (ssh) and 23 (telnet) at victim.isp.net. Given the time delay, it is reasonable to assume the attacker is also spoofing IPs not monitored by our IDS, and could be truly pounding the victim.

ACK 674711610 and 674719802: The Mystery Tool?

The following cases involve specific signatures which many of you may recognize. Steven Northcutt notes two acknowledgement numbers which he believes characterize a tool which conducts "reset scans." Here I outline two confirmed cases showing the 674711610 and 674719802 acknowledgement numbers as third party effects of SYN floods.

Observe the following trace:

```
06:20:51.570058 firstclass.server.edu.510 > spoofed.ip.one.7002:
 R 0:0(0) ack 674711610 win 0 (ttl 116, id 48680)
23:30:53.567215 firstclass.server.edu.510 > spoofed.ip.two.32771:
 R 0:0(0) ack 674711610 win 0 (ttl 117, id 25440)
13:55:27.737433 firstclass.server.edu.510 > spoofed.ip.three.6666:
 R 0:0(0) ack 674711610 win 0 (ttl 118, id 54468)
```

This trace seemed to conform to the model of a third party effect of a SYN flood. However, there is an extreme delay in the time between packets. This could be the result of a wide variety of spoofed sources, and I saw only a few. I guessed firstclass.server.edu to be a target host. These packets looked like responses, where port 510 was closed, or at least some mechanism was in place to resist a SYN flood. These three packets are a sample of the total traffic collected.

Researching port 510, I found it is the "firstclass" service, registered by SoftArc. SoftArc sells a product called the FirstClass Intranet Server, which can provide email, collaboration, and other services. The source IP belonged to a university, and the hostname resolution included the word "firstclass." It seemed that if a malicious Internet user wanted to perform a denial of service against this university, it might make sense to target port 510 (firstclass) on the school's FirstClass server. Given the presence of RST ACK packets from port 510 to multiple IPs, it seemed the host's buffer for port 510 was flooded and the port was now closed.

I contacted the school and confirmed their FirstClass server had been under a denial of service attack at the time and date noted in the packets sent to my hosts. The attacker was SYN flooding ports 68 (boot-p) and 510 (firstclass). The firstclass.server.edu system was not compromised and it was not originating the packets sent to my hosts. It was an innocent victim. The ACK 674711610 was generated by the tool used to SYN flood the hapless host. (To be precise, the packets sent by the tool used initial sequence numbers of 674711609, to which firstclass.server.edu replied with RST ACK 674711610.) "shaft" is one such tool; it chooses source IPs randonly. An analysis can be found here:
http://packetstorm.securify.com/distributed/shaft_analysis.txt

While I specifically confirmed this case as being the third party effect of a SYN flood against an innocent victim, I have found dozens of similar traffic involving ACK 674711610. Here are two cases: the first with the SYN flooded ports open (6666 and 6667), replying SYN ACK; the second with the SYN flooded ports closed (23), replying RST ACK.

SYN flooded port open:

```
05:41:36.772836 major.irc.host.6666 > spoofed.ip.one.1578:
 S 1822395560:1822395560(0) ack 674711610 win 4096 <mss 1460> (DF)
05:41:53.834459 major.irc.host.6666 > spoofed.ip.two.1578:
 S 311457256:311457256(0) ack 674711610 win 4096 <mss 1460> (DF)
05:42:00.765914 major.irc.host.6667 > spoofed.ip.three.1433:
 S 1074583123:1074583123(0) ack 674711610 win 61440 <mss 1460> (DF)
05:42:08.955560 major.irc.host.6666 > spoofed.ip.four.1433:
 S 2056091293:2056091293(0) ack 674711610 win 4096 <mss 1460> (DF)
05:43:08.425388 major.irc.host.6666 > spoofed.ip.two.1578:
 S 311457256:311457256(0) ack 674711610 win 4096 <mss 1460> (DF)
05:43:09.175868 major.irc.host.6666 > spoofed.ip.one.1578:
 S 1822395560:1822395560(0) ack 674711610 win 4096 <mss 1460> (DF)
05:43:09.816458 major.irc.host.6666 > spoofed.ip.four.1433:
 S 2056091293:2056091293(0) ack 674711610 win 4096 <mss 1460> (DF)
05:43:10.558625 major.irc.host.6667 > spoofed.ip.three.1433:
 S 1074583123:1074583123(0) ack 674711610 win 61440 <mss 1460> (DF)
```

SYN flooded port closed:

```
12:52:10.879563 auction.this.com.23 > spoofed.ip.one.1985:
 R 0:0(0) ack 674711610 win 0
12:54:37.882708 auction.this.com.23 > spoofed.ip.one.1554:
 R 0:0(0) ack 674711610 win 0
12:55:38.961649 auction.this.com.23 > spoofed.ip.one.1409:
 R 0:0(0) ack 674711610 win 0
```

Again, note the time delay between packets. This indicates not all the IPs spoofed by the attacker belonged to our monitored network. The next trace prompted a similar investigation:

```
10:20:52.097570 commercial.web.server.21 > spoofed.ip.one.1485:
 R 0:0(0) ack 674719802 win 0 (ttl 50, id 1034)
10:22:28.994103 commercial.web.server.23 > spoofed.ip.one.1485:
 R 0:0(0) ack 674719802 win 0 (ttl 50, id 38438)
10:25:43.004888 commercial.web.server.53 > spoofed.ip.one.1485:
 R 0:0(0) ack 674719802 win  (ttl 50, id 43626)

10:20:40.594667 commercial.web.server.21 > spoofed.ip.two.2104:
 R 0:0(0) ack 674719802 win 0 (ttl 45, id 14598)
10:22:17.576229 commercial.web.server.23 > spoofed.ip.two.2104:
 R 0:0(0) ack 674719802 win 0 (ttl 45, id 11298)
10:25:31.402693 commercial.web.server.53 > spoofed.ip.two.2104:
 R 0:0(0) ack 674719802  0 (ttl 45, id 33894)

10:20:31.126616 commercial.web.server.21 > spoofed.ip.three.1667:
 R 0:0(0) ack 674719802 win 0 (ttl 44, id 35589)
10:22:08.074117 commercial.web.server.23 > spoofed.ip.three.1667:
 R 0:0(0) ack 674719802 win 0 (ttl 44, id 20256)
10:25:22.038942 commercial.web.server.53 > spoofed.ip.three.1667:
 R 0:0(0) ack 674719802 win 0 (ttl 44, id 14437)
```

This source IP belonged to a commercial web site. While the three "source" ports, 21 (ftp), 23 (telnet), and 53 (dns) made little sense as true source ports, they might be good candidates as targets of a SYN flood. Sure enough, after contacting the web site, the system administrator told me a hired security consultant had tested the web server with a denial of service attack at the exact date and time indicated by my logs. Therefore, it is likely similar traces with ACK 674719802 are also the result of an attacker spoofing our IPs to SYN flood a separate victim. I do not believe these packets are generated to scan the destination IPs (here listed as spoofed.ip.xxxx). A tool called “synk4.c” creates SYN 674719801 packets, according to DDoS guru David Dittrich, who made this discovery by analyzing the following section of code:

```
 #define SEQ 0x28376839 (0x28376839 is decimal 674719801)
```

According to Dave, “synk4 takes a source address on the command line for outgoing packets, and if zero, it generates them randomly using this code”:

```
. . .
        for (i=1;i>0;i++)
          {
             srandom((time(0)+i));
             srcport = getrandom(1, max)+1000;
             for (x=lowport;x<=highport;x++)
               {
                  if ( urip == 1 )
                    {
                       a = getrandom(0, 255);
                       b = getrandom(0, 255);
                       c = getrandom(0, 255);
                       d = getrandom(0, 255);
                       sprintf(junk, "%i.%i.%i.%i", a, b, c, d);
```

```
                        me_fake = getaddr(junk);
                     }
. . .
```

Source code is here:
http://packetstorm.securify.com/spoof/unix-spoof-code/synk4.zip

As with ACK 674711610, I have found many examples of third party effects of SYN floods, where innocent victims are sending response packets to spoofed source IPs.

SYN flooded port open:

```
22:23:08.281683 biology.web.com.23 > spoofed.ip.one.1502:
 S 2894258800:2894258800(0) ack 674719802 win 8192 <mss 65512>
22:25:46.030135 biology.web.com.23 > spoofed.ip.one.2154:
 S 4154715243:4154715243(0) ack 674719802 win 8192 <mss 152>
22:26:24.456103 biology.web.com.23 > spoofed.ip.one.2026:
 S 159261598:159261598(0) ack 674719802 win 8192 <mss 32>
22:29:38.265734 biology.web.com.23 > spoofed.ip.one.1838:
 S 1866996756:1866996756(0) ack 674719802 win 8192 <mss 152>
```

SYN flooded port closed:

```
22:34:47.629194 van.smack.net.21 > spoofed.ip.two.2031:
 R 0:0(0) ack 674719802 win 0
22:36:01.282720 van.smack.net.21 > spoofed.ip.two.1071:
 R 0:0(0) ack 674719802 win 0
22:36:11.483963 van.smack.net.21 > spoofed.ip.two.2143:
 R 0:0(0) ack 674719802 win 0
```

At this time I am convinced that packets bearing ACK 674711610 and 674719802 are most likely the third party effects of a SYN flood against an innocent victim. This has been shown in my experience by contacting the sites which are the "sources" of these packets, and investigating their associated network histories.

What about reset scans? Do they exist? Presumably, the purpose of a reset scan is to determine the presence of live hosts on a network. A technique known as inverse mapping can be used to find live hosts on a network which allows its border routers or firewall to transmit ICMP error messages. If an attacker sends a RST ACK packet to a host which does not exist, the destination network's last router or firewall should send an ICMP host unreachable message. If the router/firewall is silent, we assume the target host MIGHT exist. Again, this technique relies returning ICMP error messages to source hosts. A reset scan of a network preventing outbound ICMP error messages would not yield nothing but false positive results to a reconnaissance gatherer. Reset scans can not be used to determine if ports are open on target machines. Why? Both open and closed ports should remain silent if a RST ACK packet is received. While not all vendors may implement this aspect of the RFC appropriately, most attempts to exploit these differences would be swamped by the false positive rate. Given these limiting factors, I tend not to invoke "reset scan" as an explanation for these sorts of packets.

A Final Case

I will conclude with a set of interesting traces which initially stumped me. With the help of my colleagues, and especially Mark Shaw, I pieced together the following case. Assume all the activity was registered by a single NIDS monitoring name.server.net.

```
09:22:56.960442 tester.newjersey.net.2100 > name.server.net.53:
 S 2070441966:2070442030(64) win 2048 (ttl 246, id 34960)
09:22:56.960555 tester.newjersey.net.2101 > name.server.net.53:
 S 1884680148:1884680212(64) win 2048 (ttl 246, id 8490)
09:22:56.960669 tester.newjersey.net.2102 > name.server.net.53:
 S 938156752:938156816(64) win 2048 (ttl 246, id 17966)
09:26:30.485472 tester.newjersey.net.2100 > name.server.net.53:
 S 593222604:593222668(64) win 2048 (ttl 246, id 10971)
09:26:30.485586 tester.newjersey.net.2101 > name.server.net.53:
 S 171736880:171736944(64) win 2048 (ttl 246, id 6989)
09:26:30.486219 tester.newjersey.net.2102 > name.server.net.53:
 S 1199445751:1199445815(64) win 2048 (ttl 246, id 47166)

09:24:13.867591 tester.brazil.net.2100 > name.server.net.53:
 S 795939539:795939603(64) win 2048 (ttl 241, id 53652)
09:24:13.868783 tester.brazil.net.2101 > name.server.net.53:
 S 2049322111:2049322175(64) win 2048 (ttl 241, id 13883)
09:24:13.873062 tester.brazil.net.2102 > name.server.net.53:
 S 1779866028:1779866092(64) win 2048 (ttl 241, id 14298)
09:28:04.337763 tester.brazil.net.2600 > name.server.net.53:
 S 535782194:535782258(64) win 2048 (ttl 241, id 7673)
09:28:04.339246 tester.brazil.net.2601 > name.server.net.53:
 S 1049573717:1049573781(64) win 2048 (ttl 241, id 37399)
09:28:04.339383 tester.brazil.net.2602 > name.server.net.53:
 S 148280449:148280513(64) win 2048 (ttl 241, id 25525)

09:23:26.765186 tester.argentina.net.2100 > name.server.net.53:
 S 1616673589:1616673653(64) win 2048 (ttl 241, id 21017)
09:23:26.765744 tester.argentina.net.2101 > name.server.net.53:
 S 1351385345:1351385409(64) win 2048 (ttl 241, id 9204)
09:23:26.766781 tester.argentina.net.2102 > name.server.net.53:
 S 184647009:184647073(64) win 2048 (ttl 241, id 8397)
09:24:26.275614 tester.argentina.net.2100 > name.server.net.53:
 S 1577583159:1577583223(64) win 2048 (ttl 241, id 10735)
09:24:26.276245 tester.argentina.net.2101 > name.server.net.53:
 S 1874158503:1874158567(64) win 2048 (ttl 241, id 44674)
09:24:26.276922 tester.argentina.net.2102 > name.server.net.53:
 S 1571547407:1571547471(64) win 2048 (ttl 241, id 20440)
09:25:42.915131 tester.argentina.net.2100 > name.server.net.53:
 S 988147012:988147076(64) win 2048 (ttl 241, id 41923)
09:25:42.915743 tester.argentina.net.2101 > name.server.net.53:
 S 819957179:819957243(64) win 2048 (ttl 241, id 40998)
09:25:42.916419 tester.argentina.net.2102 > name.server.net.53:
 S 1343568781:1343568845(64) win 2048 (ttl 241, id 22882)
```

Let us apply structured analysis to classify this activity.

- IPs: We see three separate machines -- tester.newjersey.net, tester.brazil.net, and tester.argentina.net -- attempting to connect to a single machine, name.server.net. You cannot determine anything more about the three initiating IPs, but name.server.net (you guessed it) is your name server.

- Ports: On the initiating side, we see a possible pattern. From each source IP, ports 2100, 2101, and 2102 are used. The tester.brazil.net box also employs 2600 (greets), 2601, and 2602. All destination ports are 53 (domain name service). Normal DNS traffic typically employs UDP, while zone transfers are done via TCP. Note BIND versions 8.2 and higher offer name queries via TCP. This process complicates our analysis and must be saved for a future paper.

- Flags: Every connection is a single SYN. This would indicate an attempt to begin the three-way handshake to exchange data, or perhaps start a scan.

- Traffic direction/activity: All traffic is sent from one of the three hosts to name.server.net. No replies are seen. Each source packet seems to contain 64 bytes of data. This differs from the very first trace we presented, showing an exchange between ftp.client.org and ftp.server.org. In the SYN packet which started that transfer, no data was passed. We can only guess at the data contained, as it was not saved with the rest of the TCP packet. For comparison's sake, observe the difference in the second line of each trace:

Case 1: No data in SYN packet:

```
14:05:27.083238 ftp.client.org.1057 > ftp.server.edu.21:
 S 1484414:1484414(0)
```

Case 2: 64 bytes in SYN packet:

```
09:22:56.960442 tester.newjersey.net.2100 > name.server.net.53:
 S 2070441966:2070442030(64)
```

- Time: All of the packets are sent between 09:22 and 09:28 on the same day. This indicates some level of coordination.

- Window size, TTL, and other features: Window size for each packet is 2048 bytes. TTLs for the two South American hosts are smaller than the New Jersey host, indicating they may have hopped through more routers on their way to your American-based name.server.net. This is to be expected if each host sets its initial TTL to the same value, such as 255.

- Bottom line: Why would three hosts all try to connect to one of our name servers, nearly simultaneously? Could they be responding to an action by one of our hosts? Is this activity malicious?

After discussing the situation with my colleagues, I formed a theory and sent emails to the points of contact listed in ARIN information for the three hosts. One of the three responded and explained the situation. The three IPs are part of a system which performs "load balancing" and dynamic redirection to a commercial web site. The process occurs as follows, using a fictitious example:

1. A web-browsing client in Chile wants to visit the web site of a major e-commerce site. She enters the URL in her browser. Her host contacts her local DNS to find the IP address associated with that hostname.

2. The local DNS server does not have the IP address in its cache, so it begins querying DNS servers until it reaches the authoritative name server of the domain owning the IP in question. This system, a "load balancing manager" (LBM), is either tied to, or serves as, the DNS for the domain.

3. The LBM checks its cache for any traffic management rules which declare how to handle requests from the client's IP address. At this stage the LBM may immediately return an IP address to the client's local DNS, or it may
proceed to step four.

4. Not finding any cached values, and choosing not to deliver a less-than-optimal IP choice to the client, the LBM queries its load balancing systems (LBS) at its three web sites, in New Jersey, Brazil, and Argentina.

5. The LBS' at the three sites conduct latency testing against the client's local DNS. These may include ICMP or TCP packets for which round trip time (RTT) is calculated, based upon responses from the client's DNS. The site whose tests result in lowest RTT is deemed "closest" (in Internet space) to the client. The IP of the "closest" site is returned to the LBM. Remember the "closest" IP could belong to a host with a very fast pipe, but very far away.

6. The LBM provides the client's local DNS with the IP of the Argentina web site.

7. The client's local DNS provides the IP of the Argentina web site to her host.

8. Her host makes contact with the web site in Argentina, displaying content.

Once the client has visited a web enterprise employing load balancing, her local DNS server may be subject to repeated and seemingly aggressive latency testing for extended periods of time. These are not malicious probes, however.

The goal of the system is to provide the quickest response time to the client while efficiently managing activity on the web server. While some in the security community view this activity as a malicious attempt to map the customer's network, I see it as a realistic attempt to serve the hundreds of thousands to millions of customers who visit the more popular web sites each day.

I found this particular load balancing system begins its tests by sending ICMP packets. If ICMP is denied by the client's routers or firewalls, the load balancer then attempts to connect to TCP port 53 on the client's name server. This explains the packets we are investigating. Since the name server in our example did not appear to respond, we can assume the load balancing program did not work out as planned, unfortunately.

What might be the next step? The network engineer responsible for these load balancers told me a final, more aggressive latency test can be made. Here the system would essentially scan the client's name server for an open port, then use the replying SYN ACK packet to test response time. Yes, this would look exactly like a multiple service port scan! For this reason, the network engineer said he has disabled this feature. Have you seen activity fitting this description against your name server?

The final trace is from another load balancing system. It uses a different packet type to do the job. Rather than SYN packets with 64 bytes of data, it sends SYN ACKs with no data. This activity was recorded after a visit to a site which employs the load balancing products. Neither the client (X) nor the web server (Y) are shown below, but four hosts involved with load balancing are included. They are:

name1.server.net:	DNS for web browsing client X
name2.server.net:	DNS for web browsing client X
mayfield.ohio.net:	Load balancer 1 for web server Y
greenbelt.maryland.net:	Load balancer 2 for web server Y

Here is the first load balancing server in action:

```
06:01:15.001304 mayfield.ohio.net.44132 > name1.server.net.53:
 S 10399587:10399587(0) ack 10399586 win 4128 <mss 556> (ttl 241, id 0)
06:01:16.999359 mayfield.ohio.net.44132 > name1.server.net.53:
 S 10399587:10399587(0) ack 10399586 win 4128 <mss 556> (ttl 241, id 0)
06:01:17.498365 mayfield.ohio.net.44133 > name2.server.net.53:
 S 10399588:10399588(0) ack 10399587 win 4128 <mss 556> (ttl 241, id 0)
06:01:18.528689 mayfield.ohio.net.44135 > name1.server.net.53:
 S 10399590:10399590(0) ack 10399589 win 4128 <mss 556> (ttl 241, id 0)
06:01:20.524742 mayfield.ohio.net.44135 > name1.server.net.53:
 S 10399590:10399590(0) ack 10399589 win 4128 <mss 556> (ttl 241, id 0)
```

... (thirteen similar packets deleted for clarity)

```
06:01:58.754918 mayfield.ohio.net.44172 > name2.server.net.53:
 S 10399627:10399627(0) ack 10399626 win 4128 <mss 556> (ttl 241, id 0)
```

Here is the second load balancing server, simultaneously testing the same two name servers:

```
06:01:14.967214 greenbelt.maryland.net.63604 > name1.server.net.53:
 S 34541003:34541003(0) ack 34541002 win 4128 <mss 556> (ttl 249, id 0)
06:01:17.461642 greenbelt.maryland.net.63607 > name2.server.net.53:
 S 34541006:34541006(0) ack 34541005 win 4128 <mss 556> (ttl 249, id 0)
06:01:18.503320 greenbelt.maryland.net.63609 > name1.server.net.53:
 S 34541008:34541008(0) ack 34541007 win 4128 <mss 556> (ttl 249, id 0)
06:01:19.464217 greenbelt.maryland.net.63607 > name2.server.net.53:
 S 34541006:34541006(0) ack 34541005 win 4128 <mss 556> (ttl 249, id 0)
06:01:20.682888 greenbelt.maryland.net.63615 > name2.server.net.53:
 S 34541014:34541014(0) ack 34541013 win 4128 <mss 556> (ttl 249, id 0)
```

... (seven similar packets deleted for clarity)

```
06:01:56.995151 greenbelt.maryland.net.63764 > name2.server.net.53:
 S 34541163:34541163(0) ack 34541162 win 4128 <mss 556> (ttl 249, id 0)
```

I reconstructed the load balancing process based upon my contacts with vendors and my understanding of load balancing operation. It is my best interpretation of the network traces, and shows how one can try to rebuild a puzzle given one or two crucial pieces.

Conclusion

In this paper, we began with a warning to know and potentially mistrust your NIDS. We introduced TCPDump, used it to look at a simple exchange of data via ftp, and discussed SYN floods. Multiple variations of SYN flood traffic was shown, and third party traffic was shown to not be "reset scans." We finished with two examples of load balancing software signatures. I hope this paper has encouraged you to take a closer look at your NIDS data, and share what you find. I look forward to hearing from you.

Appendix: Trace Excerpt with Absolute Sequence Numbers Printed

Relative sequence numbers are usually used, since we are typically interested in the amount of data passed once the initial sequence numbers are established. Plus, listing every full sequence number involves showing many distracting digits! Nevertheless, I found the following trace useful to understand whom is ACKing whom.

```
11:42:18.407029 dialup.modem.net.1052 > web.server.org.80:
 S 382137:382137(0) win 8192 <mss 536,nop,nop,sackOK> (DF)
11:42:18.582348 web.server.org.80 > dialup.modem.net.1052:
 S 1616321351:1616321351(0) ack 382138 win 9112 <nop,nop,sackOK,mss
536> (DF)
11:42:18.593124 dialup.modem.net.1052 > web.server.org.80:
 . ack 1616321352 win 8576 (DF)
11:42:18.659933 dialup.modem.net.1052 > web.server.org.80:
 . 382138:382674(536) ack 1616321352 win 8576 (DF)
11:42:18.664698 dialup.modem.net.1052 > web.server.org.80:
 P 382674:382684(10) ack 1616321352 win 8576 (DF)
11:42:18.884944 web.server.org.80 > dialup.modem.net.1052:
 . ack 382674 win 9112 (DF)
11:42:18.949336 web.server.org.80 > dialup.modem.net.1052:
 . ack 382684 win 9112 (DF)
11:42:19.106286 web.server.org.80 > dialup.modem.net.1052:
 P 1616321352:1616321766(414) ack 382684 win 9112 (DF)
11:42:19.232579 dialup.modem.net.1052 > web.server.org.80:
 . ack 1616321766 win 8162 (DF)
11:42:19.320803 web.server.org.80 > dialup.modem.net.1052:
 P 1616321766:1616321774(8) ack 382684 win 9112 (DF)
11:42:19.359277 web.server.org.80 > dialup.modem.net.1052:
 P 1616321774:1616321854(80) ack 382684 win 9112 (DF)
11:42:19.366198 dialup.modem.net.1052 > web.server.org.80:
 . ack 1616321854 win 8074 (DF)
```

Notice one sequence number is used by each side before any data is passed. web.server.org ACKs 382138 (showing 382137 was "used"), and dialup.modem.net ACKs 1616321352 (showing 1616321351 was "used"). Knowing these ACK numbers, we know the first byte of data passed from dialup.modem.net will be 382138, and the first byte passed by web.server.net will be 1616321352. Sure enough, the fourth packet,

```
11:42:18.659933 dialup.modem.net.1052 > web.server.org.80:
 . 382138:382674(536) ack 1616321352 win 8576 (DF)
```

and the eighth packet,

```
11:42:19.106286 web.server.org.80 > dialup.modem.net.1052:
 P 1616321352:1616321766(414) ack 382684 win 9112 (DF)
```

confirm this understanding of sequence numbers. Check the format again to be sure:

sequence number of first byte in packet:sequence number of first byte in NEXT packet (data)

Armed with this knowledge, the relative sequence numbers should make sense as well.

References

Daemon9, a.k.a. Route. "Project Neptune." (Phrack 48, Article 13, 1996)

Dietrich, Sven, Long, Neil, and Dittrich, David. “An Analysis of the ‘Shaft’ Distributed Denial of Service Tool.” http://packetstorm.securify.com/distributed/shaft_analysis.txt

Irwin, Vicki and Pomeranz, Hal. "Advanced Intrusion Detection and Packet Filtering." (SANS Network Security 99, 1999)

Newsham, Tim, and Ptacek, Tom. "Insertion, Evasion, and Denial of Service: Eluding Network Intrusion Detection." (Secure Networks, Inc., 1998)

Northcutt, Stephen. Network Intrusion Detection: An Analyst's Handbook. (Indianapolis, Indiana: New Riders, 1999)

Postel, Jon (ed.). "RFC 793: Transmission Control Protocol." (Defense Advanced Research Projects Agency, 1981)

Stevens, W. Richard. TCP/IP Illustrated, Volume 1: The Protocols. (Reading, Massachusetts: Addison-Wesley, 1994)

Acknowledgements

I would like to thank the following people for reading and commenting upon this paper, or giving me guidance prior to writing: Dave Dittrich, Chad Renfro, Bamm Visscher, Mark Shaw, Chuck Port, Cheryl Knecht, Sam Adams, John Green, Dustin Childs, Judy Novak, and all members of the Intrusion Detection Flight!

Revision History

v1.0 Draft only, unpublished
v2.0 Corrected errors, added confirmed case studies
v2.1 Corrected minor error regarding DF flag (thanks Judy Novak!)
v2.5 Reformatted and reorganized for 12th FIRST Conference; improved explanation of MSS; deleted discussion of using Snoop formatted data
v2.6 Corrected interpretation of FIN scan, thanks to presentation by John Green at SANS
v2.7 Changed personal email and home page URL
v2.8 Incorporated information on “shaft” and “synk4.c” from Dave Dittrich.