

# 6

## COMMON LOWER-LAYER PROTOCOLS



Whether troubleshooting latency issues, identifying malfunctioning applications, or zeroing in on security threats in order to be able to spot abnormal traffic, you must first understand normal traffic. In the next couple of chapters, you'll learn how normal network traffic works at the packet level.

We'll look at the most common protocols, including the workhorses TCP, UDP, and IP, and more commonly used application-layer protocols such as HTTP, DHCP, and DNS. Each protocol section has at least one associated capture file, which you can download and work with directly. This chapter will specifically focus on the lower-layer protocols found in reference to layers 1 through 4 of the OSI model.

These are arguably the most important chapters in this book. Skipping the discussion would be like cooking Sunday supper without cornbread. Even if you already have a good grasp of how each protocol functions, give these chapters at least a quick read in order to review the packet structure of each.

## Address Resolution Protocol

Both logical and physical addresses are used for communication on a network. The use of logical addresses allows for communication between multiple networks and indirectly connected devices. The use of physical addresses facilitates communication on a single network segment for devices that are directly connected to each other with a switch. In most cases, these two types of addressing must work together in order for communication to occur.

Consider a scenario where you wish to communicate with a device on your network. This device may be a server of some sort or just another workstation you need to share files with. The application you are using to initiate the communication is already aware of the IP address of the remote host (via DNS, covered in Chapter 7), meaning the system should have all it needs to build the layer 3 through 7 information of the packet it wants to transmit. The only piece of information it needs at this point is the layer 2 data link data containing the MAC address of the target host.

MAC addresses are needed because a switch that interconnects devices on a network uses a *Content Addressable Memory (CAM) table*, which lists the MAC addresses of all devices plugged into each of its ports. When the switch receives traffic destined for a particular MAC address, it uses this table to know through which port to send the traffic. If the destination MAC address is unknown, the transmitting device will first check for the address in its cache; if it is not there, then it must be resolved through additional communication on the network.

The resolution process that TCP/IP networking (with IPv4) uses to resolve an IP address to a MAC address is called the *Address Resolution Protocol (ARP)*, which is defined in RFC 826. The ARP resolution process uses only two packets: an ARP request and an ARP response (see Figure 6-1).

**NOTE** *An RFC, or Request for Comments, is the official document that defines the implementation standards for protocols. You can search for RFC documentation at the RFC Editor home page, <http://www.rfc-editor.org/>.*

The transmitting computer sends out an ARP request that basically asks, “Howdy everybody, my IP address is XX.XX.XX.XX, and my MAC address is XX:XX:XX:XX:XX:XX. I need to send something to whoever has the IP address XX.XX.XX.XX, but I don’t know its hardware address. Will whoever has this IP address please respond back with your MAC address?”

This packet is broadcast to every device on the network segment. Any device that does not have this IP address simply discards the packet. The device that does have this IP address sends an ARP reply with an answer like, “Hey, transmitting device, I’m who you are looking for with the IP address of XX.XX.XX.XX. My MAC address is XX:XX:XX:XX:XX:XX.”

Once this resolution process is completed, the transmitting device updates its cache with the MAC-to-IP address association of this device, and it can begin sending data.

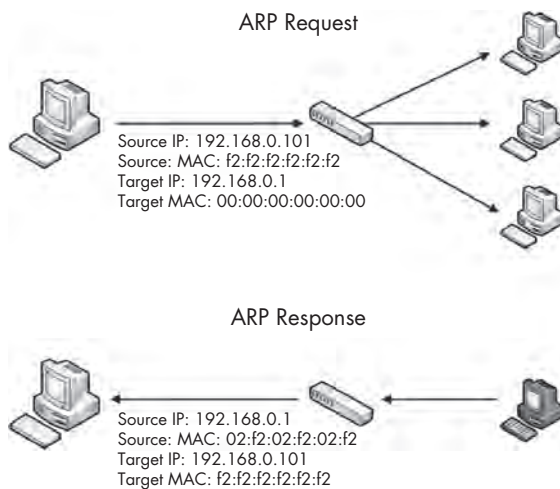


Figure 6-1: The ARP resolution process

**NOTE** You can view the ARP table of a Windows host by typing `arp -a` from a command prompt.

Seeing this process in action will help you to understand how it works. But before we look at some examples, let's examine the ARP packet header.

### The ARP Header

As shown in Figure 6-2, the ARP header includes the following fields:

- Hardware Type** The layer 2 type used. In most cases, this is Ethernet (type 1).
- Protocol Type** The higher-layer protocol for which the ARP request is being used.
- Hardware Address Length** The length (in octets/bytes) of the hardware address in use (6 for Ethernet).
- Protocol Address Length** The length (in octets/bytes) of the logical address of the specified protocol type.
- Operation** The function of the ARP packet: either 1 for a request or 2 for a reply.
- Sender Hardware Address** The hardware address of the sender.
- Sender Protocol Address** The sender's upper-layer protocol address.
- Target Hardware Address** The intended receiver's hardware address (zeroed in ARP requests).
- Target Protocol Address** The intended receiver's upper-layer protocol address.

Address Resolution Protocol		
Bit Offset	0-7	8-15
0	Hardware Type	
16	Protocol Type	
32	Hardware Address Length	Protocol Address Length
48	Operation	
64	Sender Hardware Address (1st 16 Bits)	
80	Sender Hardware Address (2nd 16 Bits)	
96	Sender Hardware Address (3rd 16 Bits)	
112	Sender Protocol Address (1st 16 Bits)	
128	Sender Protocol Address (2nd 16 Bits)	
144	Target Hardware Address (1st 16 Bits)	
160	Target Hardware Address (2nd 16 Bits)	
176	Target Hardware Address (3rd 16 Bits)	
192	Target Protocol Address (1st 16 Bits)	
208	Target Protocol Address (2nd 16 Bits)	

Figure 6-2: The ARP packet structure

Now open the file *arp\_resolution.pcap* to see this resolution process in action. We'll focus on each packet individually as we walk through this process.

### Packet 1: ARP Request

*arp\_resolution.pcap*

The first packet is the ARP request, as shown in Figure 6-3. We can confirm that this packet is a true broadcast packet by examining the Ethernet header in Wireshark's Packet Details pane. The packet's destination address is `ff:ff:ff:ff:ff:ff` ❶. This is the Ethernet broadcast address, and anything sent to it will be broadcast to all devices on the current network segment. The source address of this packet in the Ethernet header is listed as our MAC address ❷.

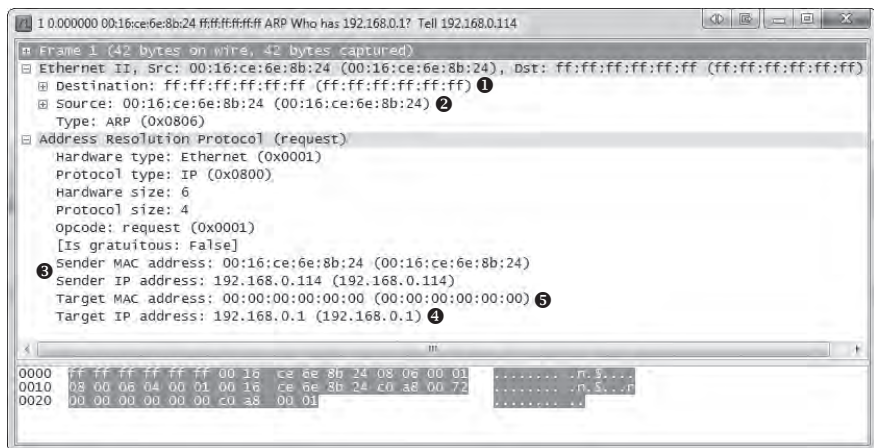


Figure 6-3: An ARP request packet

Given this structure, we can discern that this is indeed an ARP request on an Ethernet network using IP. The sender’s IP address (192.168.0.114) and MAC address (00:16:ce:6e:8b:24) are listed ❸, as is the IP address of the target (192.168.0.1) ❹. The MAC address of the target—the information we are trying to get—is unknown, so the target MAC is listed as 00:00:00:00:00:00 ❺.

## Packet 2: ARP Response

In our response to the initial request (see Figure 6-4), the Ethernet header now has a destination address of the source MAC address from the first packet. The ARP header looks similar to that of the ARP request, with a few changes:

- The packet’s operation code (opcode) is now 0x0002 ❶, indicating a reply rather than a request.
- The addressing information is reversed—the sender MAC address and IP address are now the target MAC address and IP address ❷.
- Most important, all of the information is present, meaning we now have the MAC address (00:13:46:0b:22:ba) ❸ of our host at 192.168.0.1.

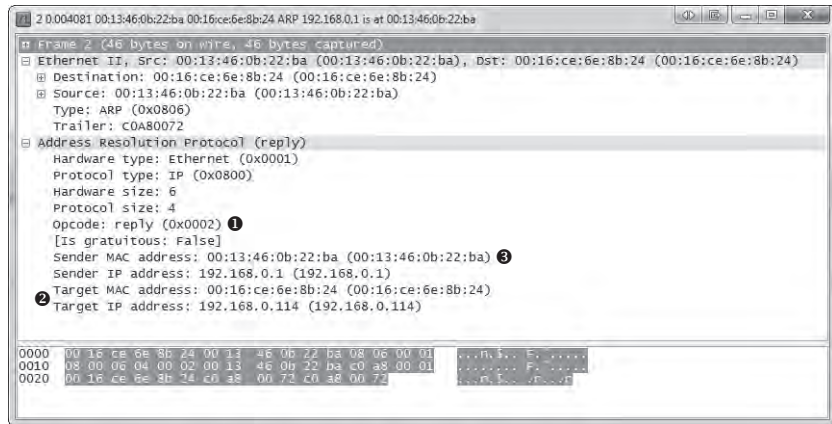


Figure 6-4: An ARP reply packet

## Gratuitous ARP

*arp\_gratuitous.pcap*

Where I come from, when something is done “gratuitously,” that usually carries a negative connotation. A *gratuitous ARP*, however, is actually a good thing.

In many cases, a device’s IP address can change. When this happens, the IP-to-MAC address mappings that hosts on the network have in their caches will be invalid. To prevent this from causing communication errors, a gratuitous ARP packet is transmitted on the network to force any device that receives it to update its cache with the new IP-to-MAC address mapping (see Figure 6-5).

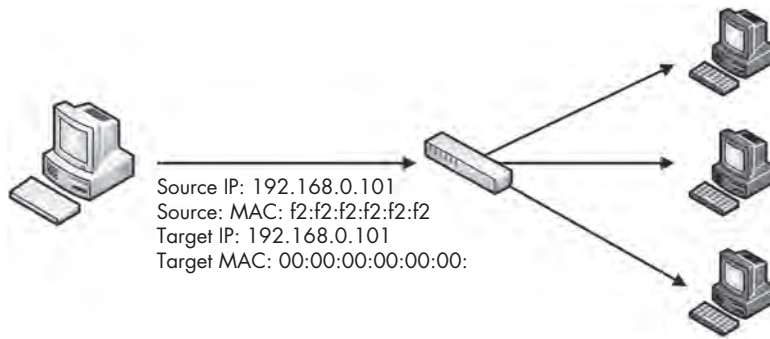


Figure 6-5: The gratuitous ARP process

A few different scenarios can spawn a gratuitous ARP packet. One of the most common is the changing of an IP address. Open the capture file *arp\_gratuitous.pcap*, and you'll see this in action. This file contains only a single packet (see Figure 6-6) because that's all that's involved in gratuitous ARP.

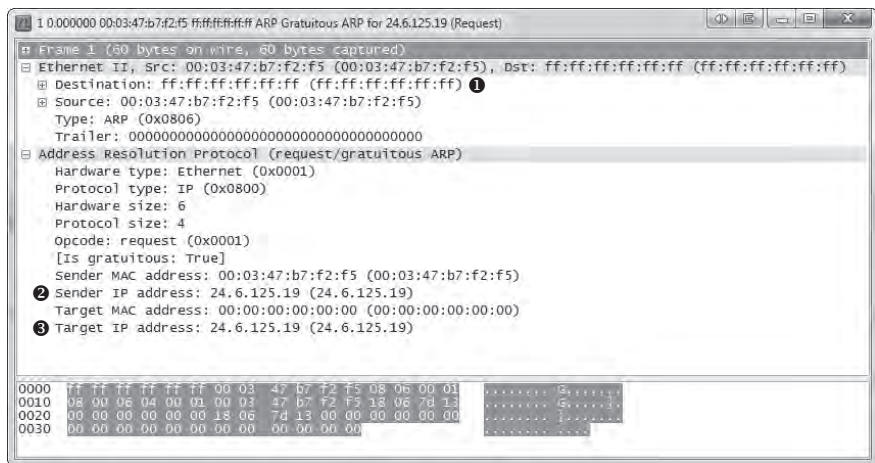


Figure 6-6: A gratuitous ARP packet

Examining the Ethernet header, you can see that this packet is sent as a broadcast so that all hosts on the network receive it ❶. The ARP header looks like an ARP request, except that the sender IP address ❷ and the target IP address ❸ are the same. When received by other hosts on the network, this packet will cause them to update their ARP tables with the new IP-to-MAC address association. Because this ARP packet is unsolicited but results in a client updating its ARP cache, the packet is considered gratuitous.

You will notice gratuitous ARP packets in a few different situations. As mentioned, changing a device's IP address will generate one. Also, some operating systems will perform a gratuitous ARP on startup. Additionally, you may notice gratuitous ARP packets on systems that use them for load-balancing of incoming traffic.

## Internet Protocol

The primary purpose of protocols at layer 3 of the OSI model is to allow for communication between networks. As you just saw, MAC addresses are used for communication on a single network at layer 2. In much the same fashion, layer 3 is responsible for addresses for internetwork communication. A few protocols can do this, but the most common is the *Internet Protocol (IP)*. Here, we'll examine IP version 4 (IPv4), which is defined in RFC 791.

In order to understand the functionality of IPv4, you need to know how traffic flows between networks. IPv4 is the workhorse of the communication process and is ultimately responsible for carrying data between devices, regardless of where the communication endpoints are located.

A simple network in which all devices are connected via hubs or switches is called a *local area network (LAN)*. When you want to connect two LANs together, you can do so with a router. Complex networks can consist of thousands of LANs connected through thousands of routers worldwide. The Internet itself is a collection of millions of LANs and routers.

### IP Addresses

*IP addresses* are 32-bit addresses used to uniquely identify devices connected to a network. It is a bit much to expect someone to remember a sequence of ones and zeros that is 32 characters long, so IP addresses are written in *dotted-quad notation*.

In dotted-quad notation, each of the four sets of ones and zeros that make up an IP address is converted to base 10 and represented as a number between 0 and 255 in the format *A.B.C.D* (see Figure 6-7). For example, consider the IP address 11000000 10101000 00000000 00000001. This value is obviously a bit much to remember or notate. Fortunately, using dotted-quad notation, we can represent it as 192.168.0.1.

IP addresses are divided into four distinct parts for a reason. An IP address consists of two parts: a *network address* and a *host address*. The network address identifies the LAN the device is connected to, and the host address identifies the device itself on that network. The determination of which part of the IP address belongs to the network or host address is not always the same. This is actually determined by another set of addressing information called the *network mask (netmask)*, sometimes also referred to as a *subnet mask*.

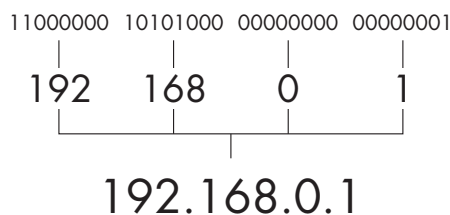


Figure 6-7: Dotted-quad IPv4 address notation

The netmask identifies which portion of the IP address belongs to the network address and which part belongs to the host address. The netmask number is also 32 bits long, and every bit that is set to a 1 identifies the portion of the IP address that is reserved for the network address. The remaining bits set to 0 identify the host address.

For example, consider the IP address 10.10.1.22, represented in binary as 00001010 00001010 00000001 00010110. In order to determine the allocation of each section of the IP address, we can apply our netmask. In this case, our netmask is 11111111 11111111 00000000 00000000. This means that the first half of the IP address is reserved for the network address (10.10 or 00001010 00001010) and the last half of the IP address identifies the individual host on this network (.1.22 or 00000001 00010110), as shown in Figure 6-8.

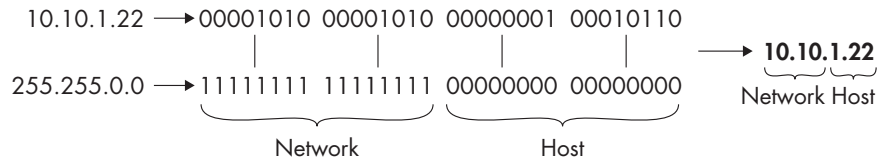


Figure 6-8: The netmask determines the allocation of the bits in an IP address.

Netmasks can also be written in dotted-quad notation. For example, the netmask 11111111 11111111 00000000 00000000 is written as 255.255.0.0.

IP addresses and netmasks are commonly written in *Classless Inter-Domain Routing (CIDR) notation* for shorthand. In this form, an IP address is written in full, followed by a forward slash (/) and the number of bits that represent the network portion of the IP address. For example, an IP address of 10.10.1.22 and a netmask of 255.255.0.0 would be written in CIDR notation as 10.10.1.22/16.

## The IPv4 Header

The source and destination IP addresses are the crucial components of the IPv4 packet header, but that's not all of the IP information you will find within a packet. The IP header is actually quite complex compared with the ARP packet we just examined. It includes a lot of extra functionality that helps IP do its job.

As shown in Figure 6-9, the IPv4 header has the following fields:

**Version** The version of IP being used

**Header Length** The length of the IP header

**Type of Service** A precedence flag and type of service flag, which are used by routers to prioritize traffic

**Total Length** The length of the IP header and the data included in the packet

**Identification** A unique identification number used to identify a packet or sequence of fragmented packets



**Flags** Used to identify whether or not a packet is part of a sequence of fragmented packets

**Fragment Offset** If a packet is a fragment, the value of this field is used to reassemble the packets in the correct order.

**Time to Live** Defines the lifetime of the packet, measured in hops/seconds through routers

**Protocol** Used to identify the type of packet coming next in the sequence of packets

**Header Checksum** An error-detection mechanism used to verify the contents of the IP header are not damaged or corrupted

**Source IP Address** The IP address of the host that sent the packet

**Destination IP Address** The IP address of the packet's destination

**Options** Reserved for additional IP options. It includes options for source routing and timestamps.

**Data** The actual data being transmitted with IP

Internet Protocol					
Bit Offset	0-3	4-7	8-15	16-18	19-31
0	Version	HDR Length	Type of Service	Total Length	
32	Identification			Flags	Fragment Offset
64	Time to Live	Protocol		Header Checksum	
96	Source IP Address				
128	Destination IP Address				
160	Options				
160 or 192+	Data				

Figure 6-9: The IPv4 packet structure

## Time to Live

`ip_ttl_source.pcap`  
`ip_ttl_dest.pcap`

The *Time to Live (TTL)* value defines a period of time that can be elapsed or a maximum number of routers a packet can traverse before the packet is discarded. A TTL is defined when a packet is created, and generally is decremented by 1 every time the packet is forwarded by a router. For example, if a packet has a TTL of 2, the first router it reaches will decrement the TTL to 1 and forward it to the second router. This router will then decrement the TTL to 0, and if the final destination of the packet is not on that network, the packet will be discarded (see Figure 6-10). Since the TTL value is technically time-based, a very busy router could decrement the TTL value by more than 1, but generally, it's safe to assume that one routing device will decrement a TTL by only 1 most of the time.

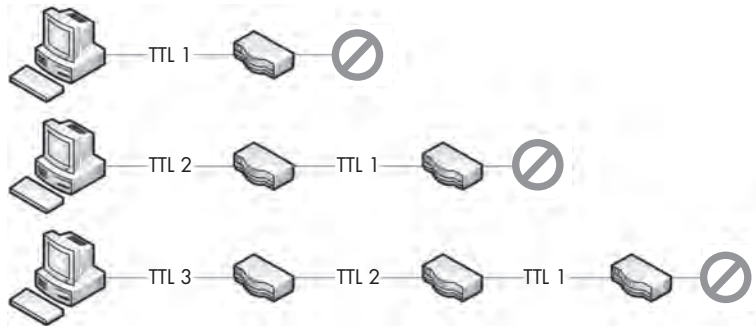


Figure 6-10: The TTL of a packet decreases every time it traverses a router.

Why is the TTL value important? Typically, we are concerned about the lifetime of a packet only in terms of the time that it takes to travel from its source to its destination. However, consider a packet that must travel to a host across the Internet while traversing dozens of routers. At some point in that packet's path, it could encounter a misconfigured router and lose the path to its final destination. In such a case, the router could do a number of things, one of which could result in the packet being forwarded around a network in a never-ending loop.

If you have any programming background at all, you know that a loop that never ends can cause all sorts of issues, typically resulting in a program or an entire operating system crashing. Theoretically, the same thing could occur with packets on a network. The packets would keep looping between routers. As the number of looping packets increased, the available bandwidth on the network would deplete until a DoS condition occurred. To prevent this potential problem, the TTL field of the IP header was created.

Let's look at an example of this in Wireshark. The file *ip\_ttl\_source.pcap* contains two ICMP packets. ICMP (discussed later in this chapter) utilizes IP to deliver packets, as we can see by expanding the IP header section in the Packet Details pane (see Figure 6-11).

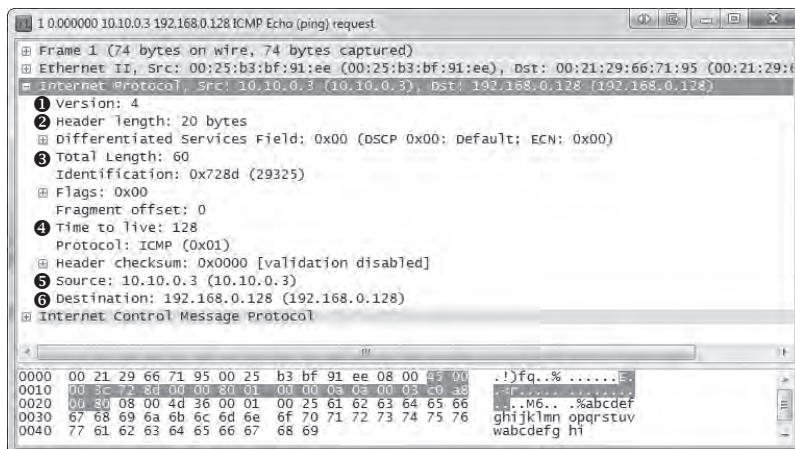


Figure 6-11: The IP header of the source packet

You can see that the version of IP being used is version 4 ❶, the IP header length is 20 bytes ❷, the total length of the header and payload is 60 bytes ❸, and the value of the TTL field is 128 ❹.

The primary purpose of an ICMP ping is to test communication between devices. Data is sent from one host to another as a request, and the receiving host should send that data back as a reply. In this file, we have one device with the address of 10.10.0.3 ❺ sending an ICMP request to a device with the address 192.168.0.128 ❻. This initial capture file was created at the source host, 10.10.0.3.

Now open the file *ip\_ttl\_dest.pcap*. In this file, the data was captured at the destination host, 192.168.0.128. Expand the IP header of the first packet in this capture to examine its TTL value (see Figure 6-12).

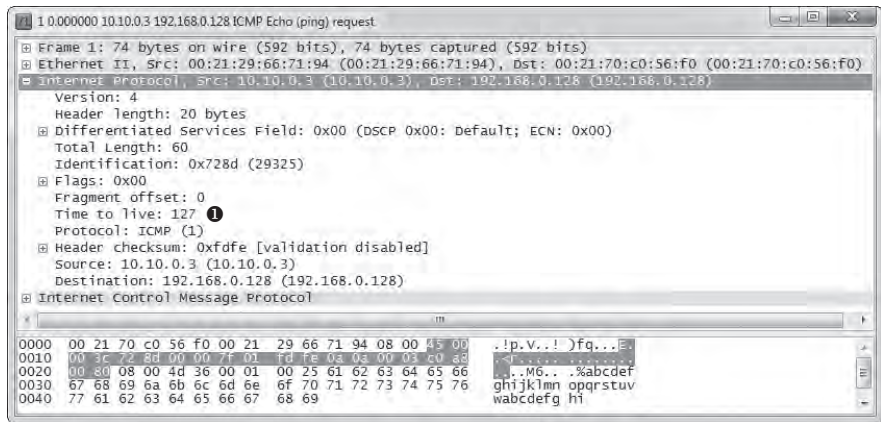


Figure 6-12: The IP header tells us that the TTL has been lowered by 1.

You should immediately notice that the TTL value is 127 ❶, one less than the original TTL of 128. Without even knowing the architecture of the network, we can conclude that these two devices are separated by one router and that the passage through that router reduced the TTL value by one.

## IP Fragmentation

*ip\_frag\_source.pcap*

*Packet fragmentation* is a feature of IP that permits reliable delivery of data across varying types of networks by splitting a data stream into smaller fragments.

The fragmentation of a packet is based on the *maximum transmission unit (MTU)* size of the layer 2 data link protocol in use and the configuration of the devices using these layer 2 protocols. In most cases, the layer 2 data link protocol in use is Ethernet. Ethernet has a default MTU of 1500, which means that the maximum packet size that can be transmitted over an Ethernet network is 1,500 bytes (not including the 14-byte Ethernet header itself).

**NOTE** *Although there are standard MTU settings, the MTU of a device can be reconfigured manually in most cases. An MTU setting is assigned on a per-interface basis and can be modified on Windows and Linux systems, as well as on the interfaces of managed routers.*

When a device prepares to transmit an IP packet, it determines whether it must fragment the packets by comparing the packet's data size to the MTU of the network interface from which the packet will be transmitted. If the data size is greater than the MTU, the packet will be fragmented. Fragmenting a packet involves the following steps:

1. The device splits the data into the number of packets required for successful data transmission.
2. The Total Length field of each IP header is set to the segment size of each fragment.
3. The More Fragments flag is set to 1 on all packets in the data stream, except for the last one.
4. The Fragment Offset field is set in the IP header of the fragments.
5. The packets are transmitted.

The file *ip\_frag\_source.pcap* was taken from a computer with the address 10.10.0.3, transmitting a ping request to a device with address 192.168.0.128. Notice that the Info column of the Packet List pane lists two fragmented IP packets, followed by the ICMP (ping) request.

Begin by examining the IP header of packet 1 (see Figure 6-13).

You can see that this packet is part of a fragment based on the More Fragments and Fragment Offset fields. Packets that are fragments either will have a positive Fragment Offset value or will have the More Fragments flag set. In the first packet, the More Fragments flag is set **1**, indicating that the receiving device should expect to receive another packet in this sequence. The Fragment Offset is set to 0 **2**, indicating this packet is the first in a series of fragments.

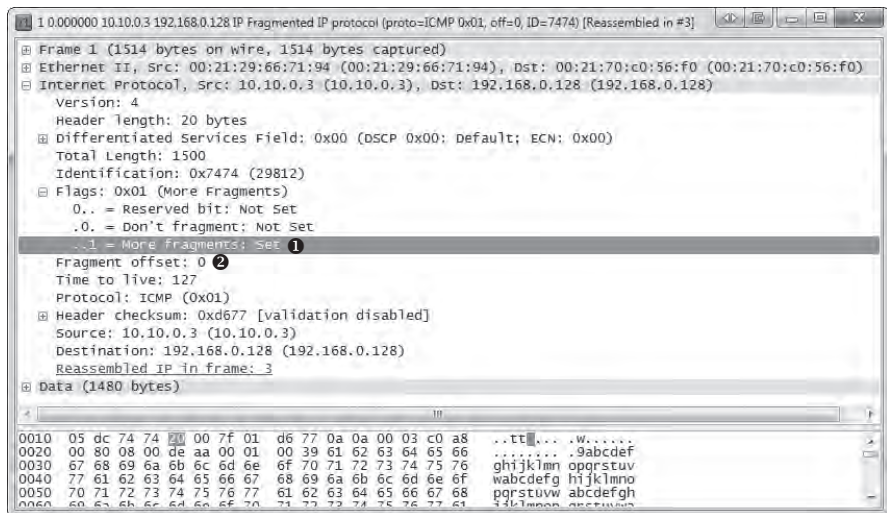


Figure 6-13: More fragments and fragment offset values can indicate a fragmented packet.

The IP header of the second packet (see Figure 6-14) also has the More Fragments flag set ❶, but in this case, the fragment offset value is 1480 ❷. This is indicative of the 1,500-byte MTU, minus 20 bytes for the IP header.

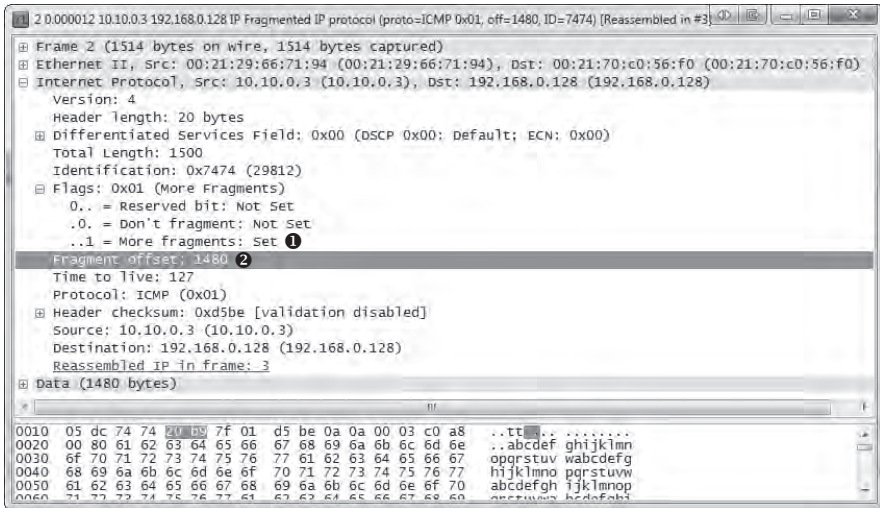


Figure 6-14: The Fragment Offset value increases based on the size of the packets.

The third packet (see Figure 6-15) does not have the More Fragments flag set ❶, which marks it as the last fragment in the data stream, and the Fragment Offset is set to 2960 ❷, the result of  $1480 + (1500 - 20)$ . These fragments can all be identified as part of the same series of data because they have the same values in the Identification field of the IP header ❸.

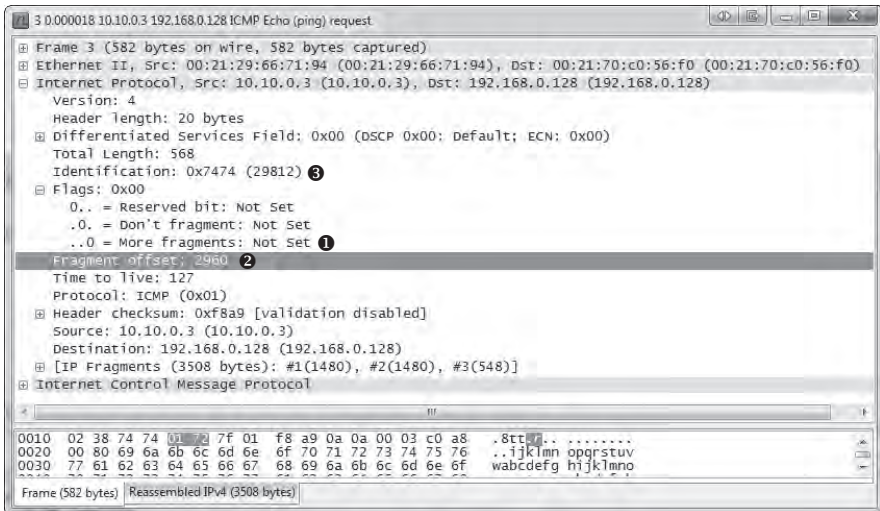


Figure 6-15: More Fragments is not set, indicating the last fragment.

# Transmission Control Protocol

The ultimate goal of the *Transmission Control Protocol (TCP)* is to provide end-to-end reliability for the delivery of data. TCP, which is defined in RFC 793, operates at layer 4 of the OSI model. It handles data sequencing and error recovery, and ultimately ensures that data gets where it is supposed to go. A lot of commonly used application-layer protocols rely on TCP and IP to deliver packets to their final destination.

## The TCP Header

TCP provides a great deal of functionality, as reflected in the complexity of its header. As shown in Figure 6-16, the following are the TCP header fields:

**Source Port** The port used to transmit the packet.

**Destination Port** The port to which the packet will be transmitted.

**Sequence Number** The number used to identify a TCP segment. This field is used to ensure that parts of a data stream are not missing.

**Acknowledgment Number** The sequence number that is to be expected in the next packet from the other device taking part in the communication.

**Flags** The URG, ACK, PSH, RST, SYN, and FIN flags for identifying the type of TCP packet being transmitted.

**Window Size** The size of the TCP receiver buffer in bytes.

**Checksum** Used to ensure the contents of the TCP header and data are intact upon arrival.

**Urgent Pointer** If the URG flag is set, this field is examined for additional instructions for where the CPU should begin reading the data within the packet.

**Options** Various optional fields that can be specified in a TCP packet.

Transmission Control Protocol				
Bit Offset	0-3	4-7	8-15	16-31
0	Source Port		Destination Port	
32	Sequence Number			
64	Acknowledgment Number			
96	Data Offset	Reserved	Flags	Window Size
128	Checksum		Urgent Pointer	
160	Options			

Figure 6-16: The TCP header

## TCP Ports

All TCP communication takes place using source and destination *ports*, which can be found in every TCP header. A port is like the jack on an old telephone switchboard. A switchboard operator would monitor a board of lights and plugs. When a light lit up, he would connect with the caller, ask who she wanted to talk to, and then connect her to her destination by plugging in a cable. Every call needed to have a source port (the caller) and a destination port (the recipient). TCP ports work in much the same fashion.

In order to transmit data to a particular application on a remote server or device, a TCP packet must know the port the remote service is listening on. If you try to access an application on a port other than the one configured for use, the communication will fail.

The source port in this sequence is not incredibly important and can be selected randomly. The remote server will simply determine the port to communicate with from the original packet it is sent (see Figure 6-17).

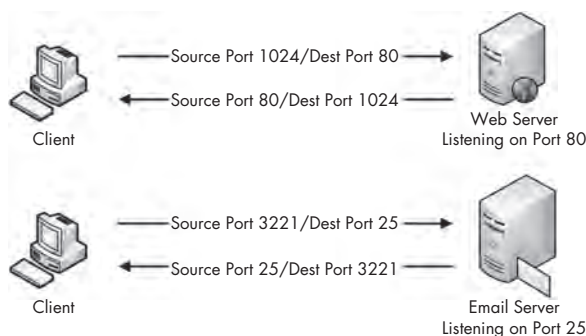



Figure 6-17: TCP uses ports to transmit data.

There are 65,535 ports available for use when communicating with TCP. We typically divide these into two groups:

- The *standard port group* is from 1 through 1023 (ignoring 0 because it is reserved). Particular services use standard ports, which generally lie within the standard port grouping.
- The *ephemeral port group* is from 1024 through 65535 (although some operating systems have different definitions for this). Only one service can communicate on a port at any given time, so modern operating systems select source ports randomly in an effort to make communications unique. These source ports are typically located in the ephemeral range.

Let's examine a couple of different TCP packets and identify the port numbers they are using by opening the file *tcp\_ports.pcap*. In this file, we have the HTTP communication of a client browsing to two websites. As mentioned previously, HTTP uses TCP for communication, which makes it a great example of standard TCP traffic.

In the first packet in this file (see Figure 6-18), the first two values represent the packet's source port and destination port. This packet is being sent from 172.16.16.128 to 212.58.226.142. The source port is 2826 , an ephemeral

port. (Remember that source ports are chosen at random by the operating system, although they can increment from that random selection.) The destination port is a standard port, port 80 ❷, the standard port used for web servers using HTTP.

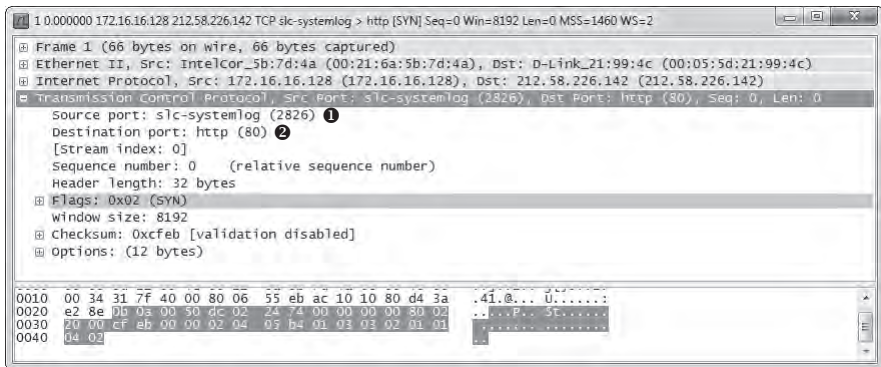


Figure 6-18: The source and destination ports can be found in the TCP header.

Notice that Wireshark labels these ports as `slc-systemlog (2826)` and `http (80)`. Wireshark maintains a list of ports and their most common uses. Although these are primarily standard ports, many ephemeral ports have commonly used services associated with them. The labeling of these ports can be quite confusing, so it's typically best to disable it by turning off transport name resolution. To do so, choose **Edit ▶ Preferences ▶ Name Resolution**, and then remove the check mark next to **Enable Transport Name Resolution**. If you wish to leave this functionality enabled but want to change how Wireshark identifies a certain port, you can do so by modifying the `Services` file located in the Wireshark program directory, which is based on the Internet Assigned Numbers Authority (IANA) common ports listing.

The second packet is being sent back from 212.58.226.142 to 172.16.16.128 (see Figure 6-19). As with the IP addresses, the source and destination ports are now also switched ❶.

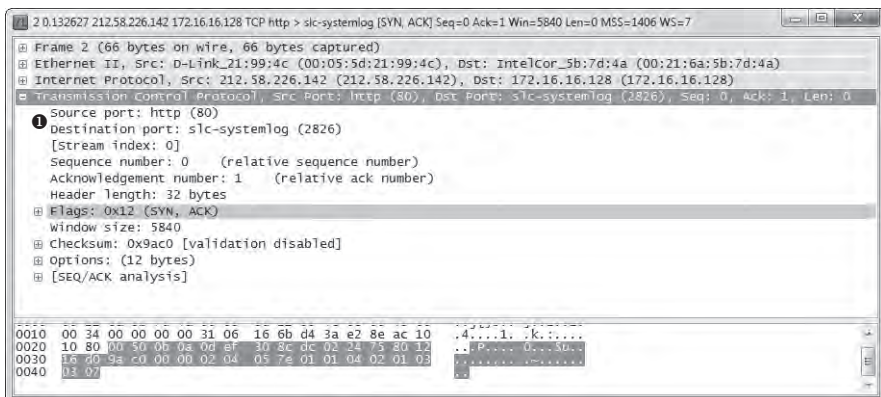


Figure 6-19: The source and destination port numbers are switched for reverse communication.



All TCP-based communication works the same way: a random source port is chosen to communicate to a known destination port. Once this initial packet is sent, the remote device communicates with the source device using the established ports.

There is one more communication stream included in this sample capture file. See if you can locate the port numbers it uses for communication.

**NOTE** *As we progress through this book, you will learn more about the ports associated with common protocols and services. Eventually, you will be able to profile services and devices by the ports they use. For a thorough list of common ports, see <http://www.iana.org/assignments/port-numbers/>.*

## The TCP Three-Way Handshake

*tcp\_handshake.pcap*

All TCP-based communication must begin with a *handshake* between two hosts. This handshake process serves a few different purposes:

- It allows the transmitting host to ensure that the destination host is up and able to communicate.
- It lets the transmitting host check that it is listening on the port on which the source is attempting to communicate.
- It allows the transmitting host to send its starting sequence number to the recipient so that both hosts can keep the stream of packets in proper sequence.

The TCP handshake occurs in three separate steps, as shown in Figure 6-20. In the first step, the device that wants to communicate (host A) sends a TCP packet to its target (host B). This initial packet contains no data other than the lower-layer protocol headers. The TCP header in this packet has the SYN flag set and includes the initial sequence number and maximum segment size (MSS) that will be used for the communication process. Host B responds to this packet by sending a similar packet with the SYN and ACK flags set, along with its initial sequence number. Finally, host A sends one last packet to host B with only the ACK flag set. Once this process is completed, both devices should have all of the information they need to begin communicating properly.

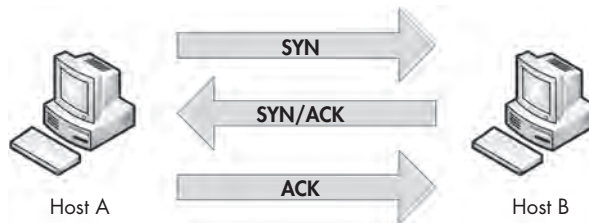


Figure 6-20: The TCP three-way handshake

**NOTE** *TCP packets are often referred to by the flags they have set. For example, rather than refer to a packet as a TCP packet with the SYN flag set, we call that packet a SYN packet. As such, the packets used in the TCP handshake process are referred to as SYN, SYN/ACK, and ACK.*

To see this process in action, open *tcp\_handshake.pcap*. Wireshark includes a feature that replaces the sequence numbers of TCP packets with relative numbers for easier analysis. For our purposes, we'll disable this feature in order to see the actual sequence numbers. To disable it, choose **Edit ▶ Preferences**, expand the **Protocols** heading, and choose **TCP**. In the window, uncheck the box next to Relative Sequence Numbers and Window Scaling, and then click **OK**.

The first packet in this capture represents our initial SYN packet (see Figure 6-21). The packet is transmitted from 172.16.16.128 on port 2826 to 212.58.226.142 on port 80. We can see here that the sequence number transmitted is 3691127924 ❶.



Figure 6-21: The initial SYN packet

The second packet in the handshake is the SYN/ACK response from 212.58.226.142 (see Figure 6-22). This packet also contains this host's initial sequence number (233779340) ❶ and an acknowledgment number (3691127925) ❷. The acknowledgment number shown here is one more than the sequence number included in the previous packet, because this field is used to specify the next sequence number the host expects to receive.

```

2 0132627 212.58.226.142 172.16.16.128 TCP http -> s1c-systemlog [SYN, ACK] Seq=233779340 Ack=3691127925 Win=5040 Len=0 MSS=1406 WS=7
+-----+
+ Frame 2 (66 bytes on wire, 66 bytes captured)
+ Ethernet II, Src: D-Link_21:199:4c (00:13:03:5d:21:199:4c), Dst: IntelCor_3b:7d:4a (00:21:6a:3b:7d:4a)
+ Internet Protocol, Src: 212.58.226.142 (212.58.226.142), Dst: 172.16.16.128 (172.16.16.128)
+ Transmission Control Protocol, Src Port: http (80), Dst Port: s1c-systemlog (2826), Seq: 233779340, Ack: 3691127925, Len: 0
+ Source port: http (80)
+ Destination port: s1c-systemlog (2826)
+ [Stream index: 0]
+ Sequence number: 233779340 ①
+ Acknowledgement number: 3691127925 ②
+ Header length: 32 bytes
+ Flags: 0x12 [SYN, ACK]
+ 0... .. = Congestion window reduced (cwr): not set
+ 0... .. = ECN-echo: not set
+ ..0... .. = Urgent: Not set
+ ...1... .. = Acknowledgement: Set
+ ....0... .. = Push: Not set
+ ....0... .. = Reset: Not set
+ ... .. = SYN: Set
+ .....0 = FIN: Not set
+ Window size: 5040
+ Checksum: 0x540d [validation disabled]
+ Options: (12 bytes)
+ Maximum segment size: 1406 bytes
+ NOP
+ SACK permitted
+ NOP
+ window scale: 7 (multiply by 128)
+ [Seq/Ack analysis]
+-----+
0010 00 34 00 00 00 31 06 16 6b d4 7a e2 8e ac 10 4...1...K...
0020 10 80 00 00 00 00 00 00 00 00 00 00 00 00 00 1...0...S...
0030 10 00 28 20 00 00 00 04 25 00 00 31 04 07 31 07 1...0...S...
0040 00 00

```

Figure 6-22: The SYN/ACK response

The final packet is the ACK packet sent from 172.16.16.128 (see Figure 6-23). This packet, as expected, contains the sequence number 3691127925 ① as defined in the previous packet’s Acknowledgment Number field.

```

2 0132768 172.16.16.128 212.58.226.142 TCP s1c-systemlog -> http [ACK] Seq=3691127925 Ack=233779341 Win=4218 Len=0
+-----+
+ Frame 3 (54 bytes on wire, 54 bytes captured)
+ Ethernet II, Src: IntelCor_3b:7d:4a (00:21:6a:3b:7d:4a), Dst: D-Link_21:199:4c (00:13:03:5d:21:199:4c)
+ Internet Protocol, Src: 172.16.16.128 (172.16.16.128), Dst: 212.58.226.142 (212.58.226.142)
+ Transmission Control Protocol, Src Port: s1c-systemlog (2826), Dst Port: http (80), Seq: 3691127925, Ack: 233779341, Len: 0
+ Source port: s1c-systemlog (2826)
+ Destination port: http (80)
+ [Stream index: 0]
+ Sequence number: 3691127925 ①
+ Acknowledgement number: 233779341
+ Header length: 20 bytes
+ Flags: 0x10 [ACK]
+ 0... .. = Congestion window reduced (cwr): not set
+ 0... .. = ECN-echo: not set
+ ..0... .. = Urgent: Not set
+ ...1... .. = Acknowledgement: Set
+ ....0... .. = Push: Not set
+ ....0... .. = Reset: Not set
+ ... .. = SYN: Not set
+ .....0 = FIN: Not set
+ Window size: 4218
+ Checksum: 0x42b2 [validation disabled]
+ [Seq/Ack analysis]
+-----+
0000 00 01 5d 21 99 4c 00 21 6a 3b 7d 4a 08 00 45 00 ...1...1...K...
0010 00 28 31 80 40 00 80 06 15 fb ac 10 10 80 d4 58 ..1...0...S...
0020 e2 8e 00 00 00 00 00 00 00 00 00 00 00 00 00 1...0...S...
0030 10 00 28 20 00 00 00 04 25 00 00 31 04 07 31 07 1...0...S...
0040 00 00

```

Figure 6-23: The final ACK

A handshake occurs before every TCP communication sequence. When sorting through a busy capture file in search of the beginning of a communication sequence, the sequence of SYN-SYN/ACK-ACK is a great marker.

### TCP Teardown

tcp\_teardown.pcap

Most greetings eventually have a good-bye, and in the case of TCP, every handshake has a teardown. The *TCP teardown* is used to gracefully end a connection between two devices after they have finished communicating. This process involves four packets, and it utilizes the FIN flag to signify the end of a connection.

In a teardown sequence, host A tells host B that it is finished communicating by sending a TCP packet with the FIN and ACK flags set. Host B responds with an ACK packet, and transmits its own FIN/ACK packet. Host A responds with an ACK packet, ending the communication process. This process is illustrated in Figure 6-24.

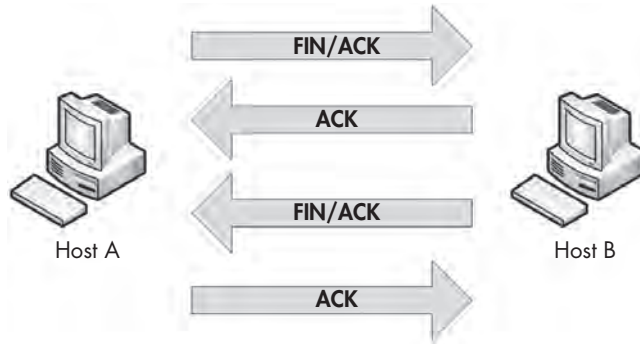


Figure 6-24: The TCP teardown process

To view this process in Wireshark, open the file `tcp_teardown.pcap`. Beginning with the first packet in the sequence, (see Figure 6-25), you can see that the device at 67.228.110.120 initiates the teardown sequence by sending a packet with the FIN and ACK flags set ❶.

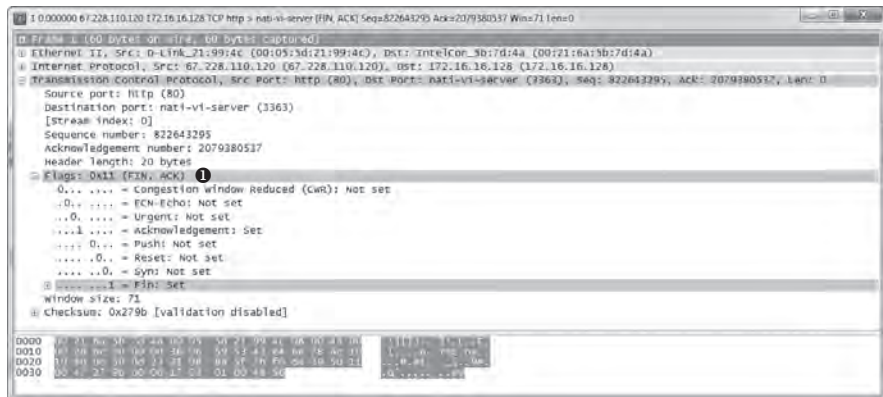


Figure 6-25: The FIN/ACK initiates the teardown process.

Once this packet is sent, 172.16.16.128 responds with an ACK packet to acknowledge receipt of the first packet, and it sends a FIN/ACK packet. The process is complete when 67.228.110.120 sends a final ACK. At this point, the communication between the two devices ends, and they must complete a new TCP handshake in order to begin communicating again.

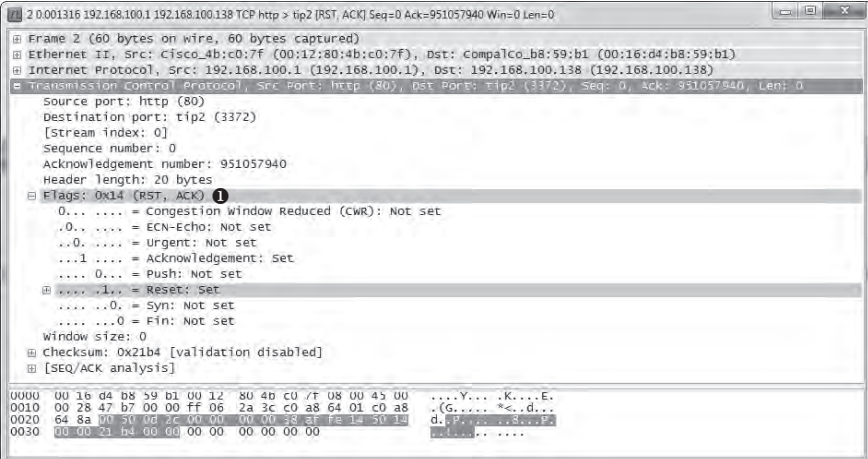
tcp\_  
refuseconnection  
.pcap

## TCP Resets

In an ideal world, every connection would end gracefully with a TCP tear-down. In reality, connections often end abruptly. For example, this may occur due to a potential attacker performing a port scan or simply a misconfigured host. In these cases, a TCP packet with the RST flag set is used. The RST flag is used to indicate a connection was closed abruptly or to refuse a connection attempt.

The file `tcp_refuseconnection.pcap` displays an example of network traffic that includes a RST packet. The first packet in this file is from the host at 192.168.100.138, which is attempting to communicate with 192.168.100.1 on port 80. What this host doesn't know is that 192.168.100.1 isn't listening on port 80 because it is a Cisco router, with no web interface configured, meaning that no service is listening for connections on port 80. In response to this attempted communication, 192.168.100.1 sends a packet to 192.168.100.138, telling it that communication won't be possible over port 80. Figure 6-26 shows the abrupt end to this attempted communication in the TCP header of the second packet. The RST packet contains nothing other than RST and ACK flags **1**, and no further communication follows.

An RST packet ends communication whether it arrives at the beginning of an attempted communication sequence, as in this example, or is sent in the middle of the communication between hosts.



```
2.0001316 192.168.100.1 192.168.100.138 TCP http > tip2 [RST, ACK] Seq=0 Ack=951057940 Win=0 Len=0
  Frame 2 (60 bytes on wire (60 bytes captured))
  Ethernet II, Src: Cisco_4b:c0:7f (00:12:80:4b:c0:7f), Dst: compalco_b8:59:b1 (00:16:d4:b8:59:b1)
  Internet Protocol, Src: 192.168.100.1 (192.168.100.1), Dst: 192.168.100.138 (192.168.100.138)
  Transmission Control Protocol, Src Port: http (80), Dst Port: tip2 (3372), Seq: 0, Ack: 951057940, Len: 0
    source port: http (80)
    Destination port: tip2 (3372)
    [Stream index: 0]
    Sequence number: 0
    Acknowledgement number: 951057940
    Header length: 20 bytes
    Flags: 0x14 (RST, ACK) 1
      0... .. = Congestion Window Reduced (CWR): Not set
      .0... .. = ECN-Echo: Not set
      ..0... .. = Urgent: Not set
      ...1... .. = Acknowledgement: Set
      ....0... .. = Push: Not set
      [....]1... .. = Reset: Set
      ....0... .. = Syn: Not set
      ....0... .. = Fin: Not set
    window size: 0
    Checksum: 0x21b4 [validation disabled]
    [Seq/ACK analysis]
0000 00 1b d4 b8 59 b1 00 12 80 4b c0 7f 08 00 45 00  ....Y... .K...E.
0010 00 28 47 b7 00 00 ff 06 2a 3c c0 a8 64 01 c0 a8  .(G.... *c..d..
0020 64 8a 00 00 00 00 00 00 00 00 00 00 00 00 00  d.....
0030 00 00 21 b4 00 00 00 00 00 00 00 00 00 00 00  d.....
```

Figure 6-26: The RST and ACK flags signify the end of communication.

## User Datagram Protocol

The *User Datagram Protocol (UDP)* is the other layer 4 protocol commonly used on modern networks. While TCP is designed for reliable data delivery with built-in error checking, UDP aims to provide speedy transmission. For this reason, UDP is a best-effort service, commonly referred to as a *connectionless*

*protocol*. A connectionless protocol does not formally establish and terminate a connection between hosts, unlike TCP with its handshake and teardown processes.

With a connectionless protocol, which doesn't provide reliable services, it would seem that UDP traffic would be flaky at best. That would be true, except that the protocols that rely on UDP typically have their own built-in reliability services, or use certain features of ICMP to make the connection somewhat more reliable. For example, the application-layer protocols DNS and DHCP, which are highly dependent on the speed of packet transmission across a network, use UDP as their transport layer protocol, but they handle error checking and retransmission timers themselves.

## The UDP Header

*udp\_dnsrequest.pcap*

The UDP header is much smaller and simpler than the TCP header. As shown in Figure 6-27, the following are the UDP header fields:

**Source Port** The port used to transmit the packet

**Destination Port** The port to which the packet will be transmitted

**Packet Length** The length of the packet in bytes

**Checksum** Used to ensure that the contents of the UDP header and data are intact upon arrival

User Datagram Protocol		
Bit Offset	0–15	16–31
0	Source Port	Destination Port
32	Packet Length	Checksum

Figure 6-27: The UDP header

The file *udp\_dnsrequest.pcap* contains one packet. This packet represents a DNS request, which uses UDP. When you expand the packet's UDP header, you'll see four fields (see Figure 6-28).

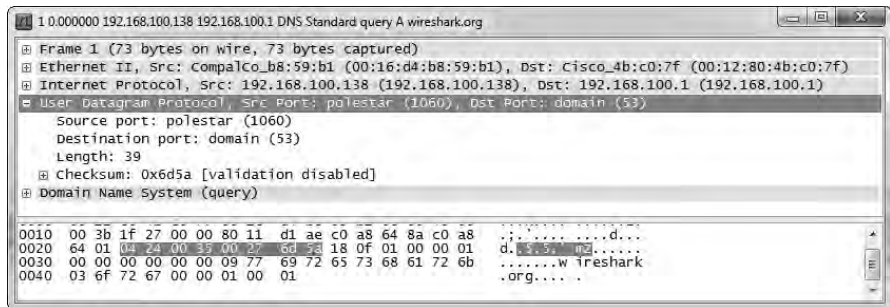


Figure 6-28: The contents of a UDP packet are very simple.

The key point to remember is that UDP does not care about reliable delivery. Therefore, any application that uses UDP must take special steps to ensure reliable delivery, if it is necessary.

## Internet Control Message Protocol

*Internet Control Message Protocol (ICMP)* is the utility protocol of TCP/IP, responsible for providing information regarding the availability of devices, services, or routes on a TCP/IP network. Most network troubleshooting techniques and tools center around common ICMP message types. ICMP is defined in RFC 792.

### **The ICMP Header**

ICMP is part of IP, and it relies on IP to transmit its messages. ICMP contains a relatively small header that changes depending on its purpose. As shown in Figure 6-29, the ICMP header contains the following fields:

**Type** The type or classification of the ICMP message, based on the RFC specification

**Code** The subclassification of the ICMP message, based on the RFC specification

**Checksum** Used to ensure that the contents of the ICMP header and data are intact upon arrival

**Variable** A portion that depends on the Type and Code fields

Internet Control Message Protocol			
Bit Offset	0–15		16–31
0	Type	Code	Checksum
32	Variable		

Figure 6-29: The ICMP header

### **ICMP Types and Messages**

As noted, the structure of an ICMP packet depends on its purpose, as defined by the values in the *Type* and *Code* fields.

You might consider the ICMP Type field as the packet’s classification and the Code field as its subclass. For example, a Type field value of 3 indicates “Destination Unreachable.” While this information alone might not be enough to troubleshoot a problem, if that packet were to also specify a Code field value of 3, indicating “Port Unreachable,” you could conclude that there is an issue with the port with which you are attempting to communicate.

**NOTE** For a full list of available ICMP types and codes, see <http://www.iana.org/assignments/icmp-parameters>.

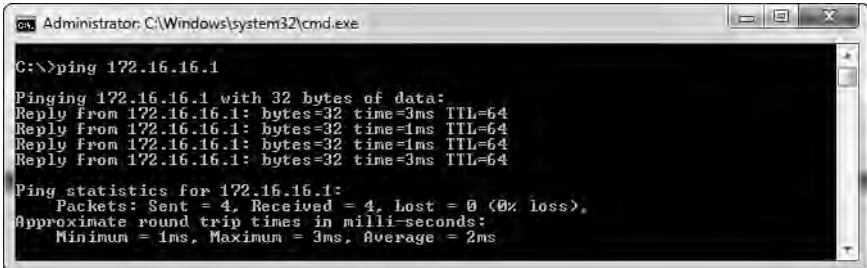
## Echo Requests and Responses

*icmp\_echo.pcap*

ICMP's biggest claim to fame is thanks to the ping utility. Ping is used to test for connectivity to a device. Most information technology (IT) professionals are familiar with ping.

To use ping, enter **ping <ip address>** at the command prompt, replacing <ip address> with the actual IP address of a device on your network. If the target device is turned on, your computer has a communication route to it, and there is no firewall blocking that communication, you should see replies to your ping command.

The example in Figure 6-30 shows four successful replies that display their size, RTT, and TTL used. The Windows utility also provides a summary detailing how many packets were sent, received, and lost. If communication fails, you should see a message telling you why.



```
Administrator: C:\Windows\system32\cmd.exe
C:\>ping 172.16.16.1
Pinging 172.16.16.1 with 32 bytes of data:
Reply from 172.16.16.1: bytes=32 time=3ms TTL=64
Reply from 172.16.16.1: bytes=32 time=1ms TTL=64
Reply from 172.16.16.1: bytes=32 time=1ms TTL=64
Reply from 172.16.16.1: bytes=32 time=3ms TTL=64
Ping statistics for 172.16.16.1:
    Packets: Sent = 4, Received = 4, Lost = 0 (0% loss),
    Approximate round trip times in milli-seconds:
        Minimum = 1ms, Maximum = 3ms, Average = 2ms
```

Figure 6-30: The ping command being used to test connectivity

Basically, the ping command sends one packet at a time to a device and listens for a reply to determine if there is connectivity to that device, as shown in Figure 6-31.

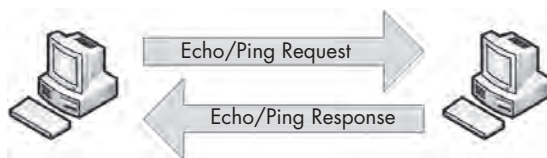


Figure 6-31: The ping command involves only two steps.

**NOTE** Although ping has long been the bread and butter of IT, its results can be a bit deceiving as host-based firewalls are deployed. Many of today's firewalls limit the ability of a device to respond to ICMP packets. This is great for security, because potential attackers using ping to determine if a host is accessible might be deterred, but troubleshooting is also made more difficult—it can be frustrating to ping a device to test for connectivity and not receive a reply when you know you can communicate with that device.

The ping utility in action is a great example of simple ICMP communication. The packets in the file *icmp\_echo.pcap* demonstrate what happens when you run ping.



The first packet (see Figure 6-32) shows that host 192.168.100.138 is sending a packet to 192.168.100.1 ❶. When you expand the ICMP portion of this packet, you can determine the ICMP packet type by looking at the Type and Code fields. In this case, the packet is type 8 ❷, code 0 ❸, indicating an echo request. (Wireshark should tell you what the type/code being displayed actually is.) This echo (ping) request is the first half of the equation. It is a simple ICMP packet, sent using IP, that contains a small amount of data. Along with the type and code designations and the checksum, we also have a sequence number that is used to pair requests with replies, and a random text string in the variable portion of the ICMP packet.

**NOTE** *The terms echo and ping are often used interchangeably, but just remember that ping is actually the name of a tool. The ping tool is used to send ICMP echo request packets.*

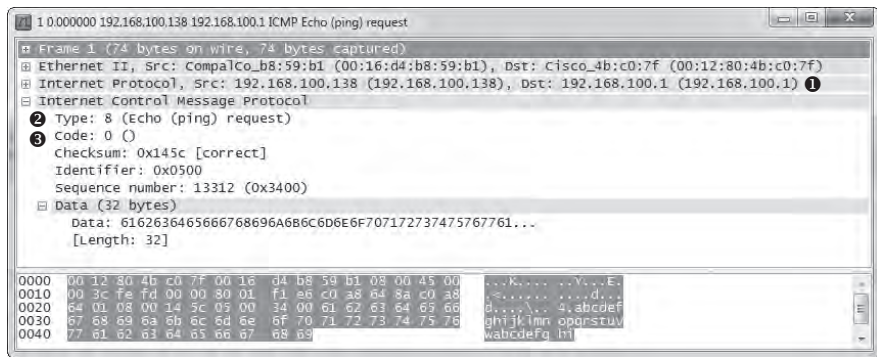


Figure 6-32: The ICMP echo request packet

The second packet in this sequence is the reply to our request (see Figure 6-33). The ICMP portion of the packet is type 0 ❶, code 0 ❷, indicating that this is an echo reply. Because the sequence number in the second packet matches that of the first ❸, we know that this echo reply matches the echo request in the previous packet. This reply packet also contains the same 32-byte string of data that was transmitted with the initial request ❹. Once this second packet has been received by 192.168.100.138, ping will report success (see Figure 6-30, shown earlier).

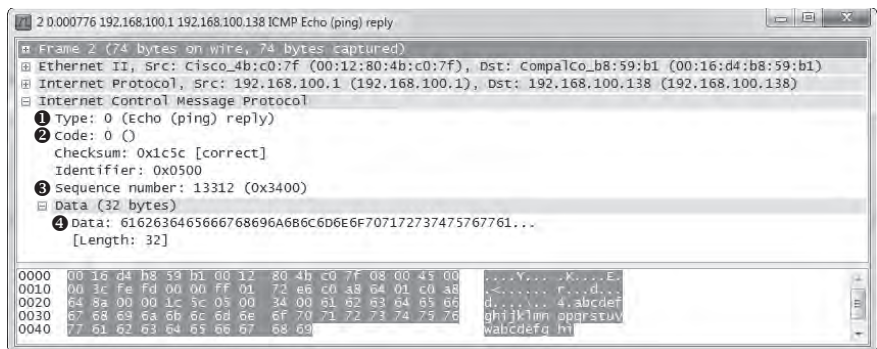


Figure 6-33: The ICMP echo reply packet

Note that you can use variations of ping to increase the size of the data padding, which forces packets to be fragmented for various types of network troubleshooting. This may be required when you're troubleshooting networks that require a smaller fragment size.

**NOTE** *The random text used in an ICMP echo request can be of great interest to a potential attacker. Attackers can use the information in this padding to profile the operating system used on a device. Additionally, attackers can place small bits of data in this field as a method of covert communication.*

## Traceroute

*icmp\_traceroute.pcap*

The traceroute utility is used to identify the path from one device to another. On a simple network, a path may go through only a single router or no router at all. On a complex network, however, a packet may need to go through dozens of routers to reach its final destination, which is why it's crucial to be able to trace the exact path a packet takes from one destination to another in order to troubleshoot communication.

By using ICMP (with a little help from IP), traceroute can map the path packets take. For example, the first packet in the file *icmp\_traceroute.pcap* is pretty similar to the echo request we looked at in the previous section (see Figure 6-34).

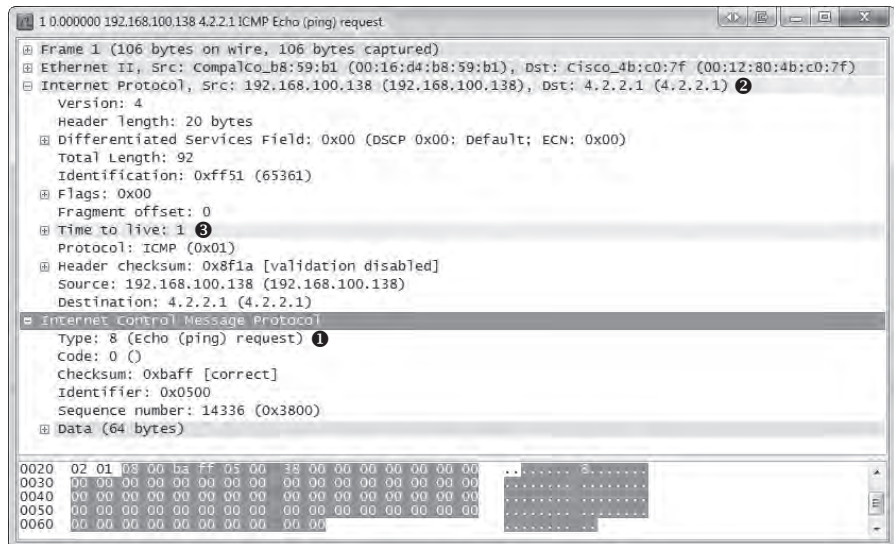


Figure 6-34: An ICMP echo request packet with a TTL value of 1

At first glance, this packet appears to be a simple echo request 1 from 192.168.100.138 to 4.2.2.1 2, and everything in the ICMP portion of the packet is identical to the formatting of an echo request packet. However, when you expand the IP header of this packet, you'll notice one odd value: The packet's TTL value is set to 1 3, which means that the packet will be dropped at the first router that it hits. Because the destination 4.2.2.1

address is an Internet address, we know that there must be at least one router between our source and destination devices, so there is no way this packet will reach its destination. That's good for us, because traceroute relies on the fact that this packet will make it to only the first router it traverses.

The second packet is, as expected, a reply from the first router we reached along the path to our destination (see Figure 6-35). This packet reached this device at 192.168.100.1, its TTL was decremented to 0, and the packet could not be transmitted further, so the router replied with an ICMP response. This packet's type 11 ❶, code 0 ❷ tells us that the destination was unreachable because the packet's TTL was exceeded during transit.

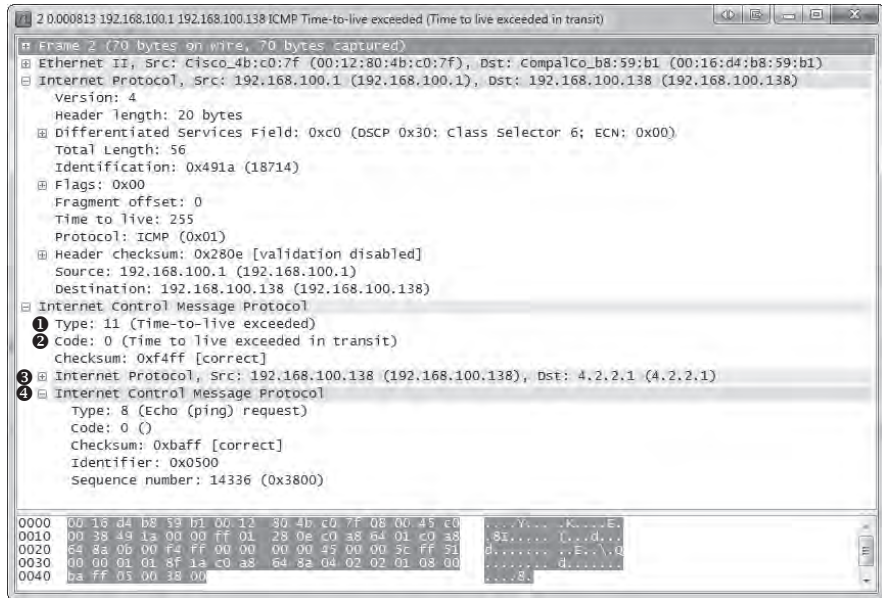


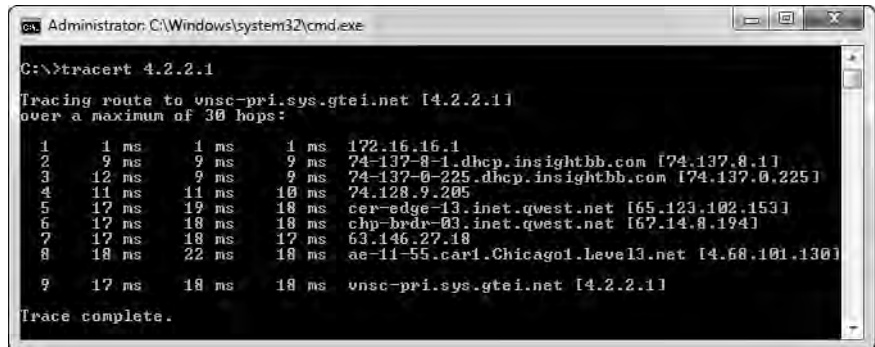
Figure 6-35: An ICMP response from the first router along the path

This ICMP packet is sometimes called a *double-headed packet*, because the tail end of its ICMP portion contains a copy of the IP header ❸ and ICMP data ❹ that was sent in the original echo request. This information can prove to be very useful for troubleshooting.

This process of sending packets with incremented TTL values occurs two more times before we get to packet 7. Here, you see the same thing you saw in the first packet, except that this time, the TTL value in the IP header is set to 2, which ensures the packet will make it to the second hop router before it is dropped. As expected, we receive a reply from the next hop router, 12.180.241.1, with the same ICMP destination unreachable and TTL exceeded messages. This process continues with the TTL value increasing by one until the destination 4.2.2.1 is reached.

To sum up, this traceroute process has communicated with each router along the path, building a map of the route to the destination. This map is shown in Figure 6-36.

**NOTE** *The discussion here on traceroute is generally Windows-focused because it uses ICMP exclusively. The traceroute utility on Linux is a bit more versatile and can utilize other protocols in order to perform route path tracing.*



```
Administrator: C:\Windows\system32\cmd.exe
C:\>tracert 4.2.2.1
Tracing route to vns-c-pri.sys.gtei.net [4.2.2.1]
over a maximum of 30 hops:
  0  0 ms  0 ms  0 ms  172.16.16.1
  1  9 ms  9 ms  9 ms  74-137-8-1.dhcp.insightbb.com [74.137.8.1]
  2 12 ms  9 ms  9 ms  74-137-0-225.dhcp.insightbb.com [74.137.0.225]
  3 11 ms 11 ms 10 ms 74.128.9.205
  4 17 ms 19 ms 18 ms cer-edge-13.inet.qwest.net [65.123.102.153]
  5 17 ms 18 ms 18 ms chp-brdr-03.inet.qwest.net [67.14.8.194]
  6 17 ms 18 ms 17 ms 63.146.27.18
  7 18 ms 22 ms 18 ms ae-11-55.car1.Chicago1.Level3.net [4.68.101.130]
  8 17 ms 18 ms 18 ms vns-c-pri.sys.gtei.net [4.2.2.1]
Trace complete.
```

Figure 6-36: A sample output from the traceroute utility

As you'll see throughout this book, ICMP has many different functions. We'll use ICMP frequently as we analyze more scenarios.

This chapter has introduced you to a few of the most important protocols you will examine in the process of packet analysis. IP, TCP, UDP, and ICMP are at the foundation of all network communications, and they are critical to just about every daily task you perform. In the next chapter, we will look at a grouping of common application-layer protocols.