

概要

11/06/27 10:55:29

文書間に違いがあります。

新文書：

[2nd_08_ato](#)

123 ページ (6.72 MB)

11/06/27 10:53:43

旧文書：

[1st_08_ato](#)

66 ページ (3.33 MB)


11/06/27 10:53:35

結果の表示に使用します。


[最初の変更箇所が 1 ページ目にあります。](#)


削除されたページはありません

このレポートの読み方

 は、変更箇所を示します。

 は、削除された内容を示します。

 は、ページが変更されたことを示します。

 は、ページが移動されたことを示します。

8

BASIC REAL-WORLD SCENARIOS



Beginning with this chapter, we'll dig into the meat of packet analysis, as we use Wireshark to analyze real-world network problems. In the first part, we'll analyze scenarios that you might encounter day to day as a network engineer, help desk technician, or application developer—all derived from my real-world experiences and those of my colleagues. We'll use Wireshark to examine traffic from Twitter, Facebook, and ESPN.com to see how these common services work.

The second part of this chapter introduces a series of real-world problems. For each, I describe the situation surrounding each problem and offer the information that was available to the analyst at the time. Having laid the groundwork, we'll turn to analysis, as I describe the method used to capture the appropriate packets and step you through the analysis process. Once analysis is complete, I offer a full solution to the problem or point you to potential solutions, along with an overview of lessons learned.

Throughout, remember that analysis is a very dynamic process, and the methods I use to analyze each scenario may not be the same ones that you might use. Everyone analyzes in different ways. The most important thing is

that the end result of the analysis solves a problem or provides a learning experience. In addition, most problems discussed in this chapter can probably be solved without a packet sniffer. When I was first learning how to analyze packets I found it helpful to examine typical problems in atypical ways by using packet analysis techniques, which is why I present these scenarios to you.

Social Networking at the Packet Level

First, we'll look at the traffic of two popular social networking websites: Twitter and Facebook. We'll examine the authentication process associated with each service and see how the two very similar functions use different methods to perform the same task. We'll also look at how some of the primary functions of each service work in order to gain a better understanding of the traffic we generate in our normal daily activities.

Capturing Twitter Traffic

`twitter_login.pcap` Whether you use Twitter to stay up-to-date on news in the tech community or just to complain about your girlfriend, it's one of the more commonly used services on the Internet. For this scenario, you'll find a capture of Twitter traffic in the file `twitter_login.pcap`.

NOTE Websites change their code frequently. As a result, if you try to re-create the captures in the next few sections you may find that your results differ from what is shown here.

The Twitter Login Process

When I teach packet analysis, one of the first things I have my students do is log in to a website they normally use and capture the traffic from the login process. This serves a dual purpose: It exposes the students to more packets in general, and it allows them to discover insecurities in their daily activities by looking for plaintext passwords traversing the wire.

Fortunately, the Twitter authentication process is not completely insecure. As you can see in Figure 8-1, these first three packets constitute the *TCP handshake* between our local device (172.16.16.128) ❶ and a remote server (168.143.162.68) ❷. The remote server is listening for our connection on port 443 ❸, which is typically associated with SSL over HTTP, commonly referred to as *HTTPS*, a secure form of data transfer. Based on these alone, we can assume that this is SSL traffic.

No.	Time	Source	❶ Destination	❷ Protocol	Info	❸
1	0.000000	172.16.16.128	168.143.162.68	TCP	4669 → 443 [SYN] Seq=4164864060 win=8192 Len=0 MSS=1460	
2	0.072728	168.143.162.68	172.16.16.128	TCP	443 → 4669 [SYN, ACK] Seq=1150193371 Ack=4164864061 win=18200 Len=0 MSS=1460	
3	0.000101	172.16.16.128	168.143.162.68	TCP	4669 → 443 [ACK] Seq=4164864061 Ack=1150193372 Win=16872 Len=0	

Figure 8-1: Handshake connecting to port 443

The packets that follow the handshake are part of the SSL *encrypted handshake*. SSL relies on *keys*—strings of characters used to encrypt and decrypt communication between two parties. The handshake process is the formal transmission of these keys between hosts, as well as the negotiation of various connection and encryption characteristics. Once this handshake is completed, secure data transfer begins.

In order to find the encrypted packets that handle the exchange of data, look for the packets that are identified as Application Data in the Info column of the Packet Details pane. Expanding the SSL portion of any of these packets will display the Encrypted Application Data field, containing the unreadable encrypted data ❶, as shown in Figure 8-2. This shows the transfer of the username and password during login.

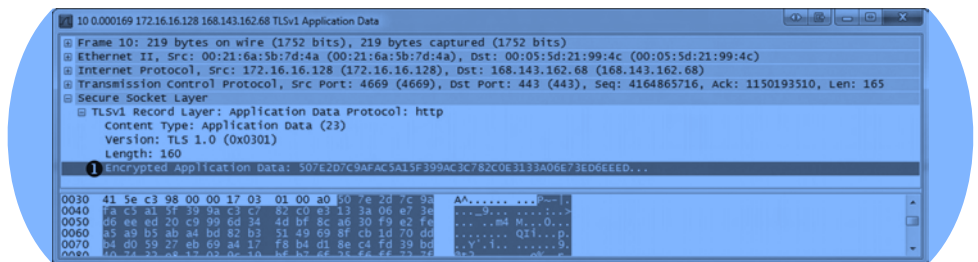


Figure 8-2: Encrypted credentials being transmitted

The authentication continues briefly until the connection begins its tear-down process with a FIN/ACK at packet 16. Following authentication, we would expect our browser to be redirected to our Twitter home page, which is exactly what happens. As you can see in Figure 8-3, packets 19, 21, and 22 are part of the handshake process that sets up a new connection to the same remote server (168.143.162.68) but on port 80 instead of 443 ❶. Following the completed handshake, we see the HTTP GET request in packet 23 for the root directory of the web server (/) ❷. The server acknowledges the request in packet 24 ❸ and begins transmitting data over the next several packets. The contents of packet 41 marks the completion of the data transmission related to the GET request.

No.	Time	Source	Destination	Protocol	Info
19	0.001117	172.16.16.128	168.143.162.68	TCP	4670 > 80 [SYN] Seq=3871493748 win=8192 Len=0 MSS=1460
21	0.000063	168.143.162.68	172.16.16.128	TCP	80 > 4670 [SYN, ACK] Seq=2866679388 Ack=3871493749 win=18200 Len=0 MSS=1406
22	0.000063	172.16.16.128	168.143.162.68	TCP	4670 > 80 [ACK] Seq=3871493749 Ack=2866679389 win=16872 Len=0
❷ 23	0.000371	172.16.16.128	168.143.162.68	HTTP	GET / HTTP/1.1
❸ 24	0.080775	168.143.162.68	172.16.16.128	TCP	80 > 4670 [ACK] Seq=2866679389 Ack=3871495149 win=8400 Len=0

Figure 8-3: The GET request for the root directory of our Twitter home page (/) once authentication has completed

Several more GET requests are made in the remainder of the capture file in order to retrieve the images and other files linked to the home page.

Sending Data with a Tweet

twitter_tweet.pcap

Once logged in, the next step is to tell the world what's on your mind. Because I'm in the middle of writing a book, I'll tweet, "This is a tweet for Practical Packet Analysis, second edition" and capture the traffic from posting that tweet in the file *twitter_tweet.pcap*.

This capture file starts as soon as the tweet is submitted. It begins with a handshake between our local workstation 172.16.16.134 and the remote address 168.143.162.100. The fourth and fifth packets in the capture comprise an HTTP packet sent from the client to the server. Wireshark has combined the data in these two packets, and placed it in the Packet Details pane of packet 5 for ease of viewing.

To examine this HTTP header, expand the HTTP section in the Packet Details pane of the fifth packet, as shown in Figure 8-4. You will see that the POST method is used with the URL `/status/update` ❶. We know that this is indeed a packet from the tweet, because the Host field contains the value `twitter.com` ❷.

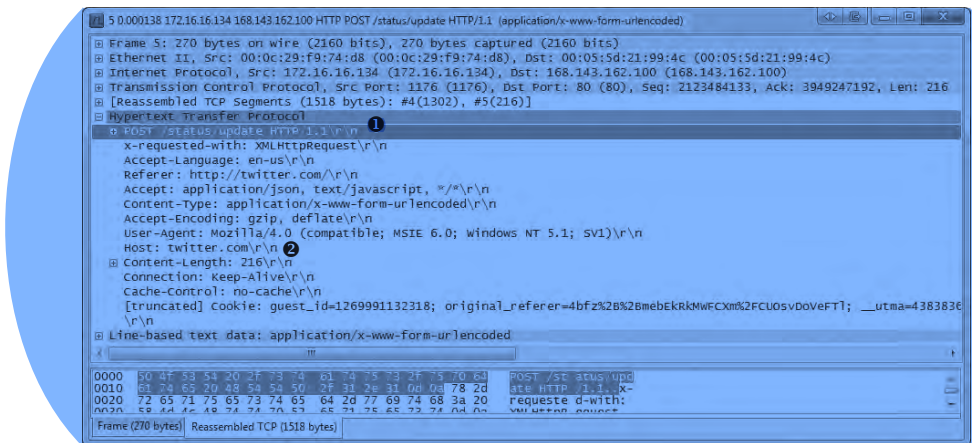


Figure 8-4: The HTTP POST for a Twitter update

Notice the header contained in the packet's Line-based Text Data field ❶ in Figure 8-5. When you analyze this data, you will see a field named Authenticity Token, followed by a `status` field in a URL containing this value:

This+is+a+tweet+for+practical+packet+analysis%2c+second+edition

The value of the `status` field is the tweet I've submitted in unencrypted plaintext.

There is a slight security concern here, because some people protect their tweets and don't intend for them to be seen by just anyone. This doesn't mean that just anybody could read the tweet, but a user on the same network could intercept this traffic and see the contents of the tweet clearly.

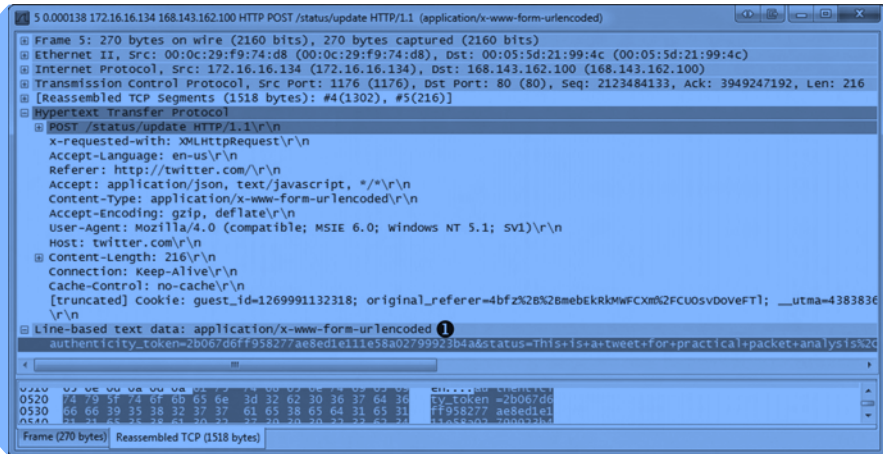


Figure 8-5: The tweet in plaintext

Twitter Direct Messaging

twitter_dm.pcap

Now we'll consider a scenario with some security implications: Twitter direct messaging, which allows users to share presumably private messages. The file *twitter_dm.pcap* is a packet capture of a Twitter direct message. As you can see in Figure 8-6, direct messages aren't exactly private.



Figure 8-6: A direct message in the clear

The display of packet 7 in Figure 8-6 shows that content is still sent in plaintext. This is evident in the same Line-based Text Data field 1 that we viewed in the previous capture.

The knowledge that we gain here about Twitter isn't necessarily earth-shattering, but it may make you reconsider sending sensitive data via private Twitter messages over untrusted networks.

Capturing Facebook Traffic

Once I've finished reading my tweets, I like to log in to Facebook to see what my friends are up to, so that I can live vicariously through them. Now let's use Wireshark to capture and analyze Facebook traffic.

The Facebook Login Process

facebook_login.pcap



We'll begin with the login process captured in *facebook_login.pcap*. The capture begins as soon as credentials are submitted, as shown in Figure 8-7. Similar to the Twitter login process, we see a TCP handshake over port 443 ❶. Our workstation at 172.16.0.122 ❷ is initiating communication with 69.63.180.173 ❸, the server handling the Facebook authentication process. Once the handshake completes, the SSL handshake occurs ❹, and login credentials are submitted.

No.	Time	Source	Destination	Protocol	Info
1	0.000000	172.16.0.122	69.63.180.173	TCP	54595 → 443 [RST] Seq=3017405622 Win=5840 Len=0 TSval=3479125856 TSecr=3479125856
2	0.000000	69.63.180.173	172.16.0.122	TCP	443 → 54595 [ACK] Seq=2917405623 Ack=2894038505 Win=92 Len=0 TSval=3479125856 TSecr=3479125856
3	0.000033	172.16.0.122	69.63.180.173	TLSv1	Client Hello ❹
4	0.000522	69.63.180.173	172.16.0.122	TLSv1	Server Hello, Certificate, Server Hello Done
5	0.000631	172.16.0.122	69.63.180.173	TLSv1	Client Key Exchange, Change Cipher Spec, Encrypted Handshake Message
6	0.000648	69.63.180.173	172.16.0.122	TLSv1	Change Cipher Spec, Encrypted Handshake Message
7	0.000648	69.63.180.173	172.16.0.122	TCP	443 → 54595 [ACK] Seq=2894039263 Ack=2917406956 Win=5473 Len=0 TSval=3479126142 TSecr=3479125856
8	0.000448	69.63.180.173	172.16.0.122	TLSv1	Application data
9	0.000533	172.16.0.122	69.63.180.173	TLSv1	Application data
10	0.180618	69.63.180.173	172.16.0.122	TCP	443 → 54595 [ACK] Seq=2894039263 Ack=2917406956 Win=5473 Len=0 TSval=3479126142 TSecr=3479125856
11	0.72207	69.63.180.173	172.16.0.122	TLSv1	Application data

Figure 8-7: Login credentials are transmitted securely with HTTPS.

One difference between the Facebook login process and the Twitter one is that we don't immediately see the authentication connection teardown following the transmission of login credentials. Instead, we see a GET request for */home.php* in the HTTP header of packet 12 ❶, as highlighted in Figure 8-8.

12 0.011407 172.16.0.122 69.63.180.173 HTTP GET /home.php? HTTP/1.1	
Frame 12: 693 bytes on wire (5544 bits), 693 bytes captured (5544 bits) on interface 0 Ethernet II, Src: 00:12:17:0c:00:00, Dst: 00:12:17:0c:00:00, Seq: 0, Len: 693 Internet Protocol Version 4, Src: 172.16.0.122, Dst: 69.63.180.173 Transmission Control Protocol, Src Port: 58637, Dst Port: 80, Seq: 1272383368, Len: 627 Hypertext Transfer Protocol	
Host: www.facebook.com/r/n User-Agent: Mozilla/5.0 (X11; U; Linux i686; en-US; rv:1.9.1.8) Gecko/20100214 Ubuntu/9.10 (karmic) Firefox/3.5.8/r/n Accept: text/html,application/xhtml+xml,application/xml;q=0.9,*/*;q=0.8/r/n Accept-Language: en-US,en;q=0.5/r/n Accept-Encoding: gzip,deflate/r/n Accept-Charset: ISO-8859-1,utf-8;q=0.7,*/*;q=0.7/r/n Keep-Alive: 300/r/n Connection: keep-alive/r/n Referer: http://www.facebook.com/r/n Cookie: datr=1270494458-fa0d48b0ba777054edaf01bfb8a07ee6ddea85f4a3d253cbbd; tsd=rlv1; test_cookie=1; c_user=100000943187386; lo=812nj4zrP5uptr/r/n	

Figure 8-8: After authentication, the GET request for */home.php* takes place.

The connection used for authentication is torn down after the contents of *home.php* is delivered, as seen in packet 64 ❶ at the end of the capture file in Figure 8-9. First, the HTTP connection over port 80 is torn down (packet 62) ❷, and then the HTTPS connection over port 443 is torn down.

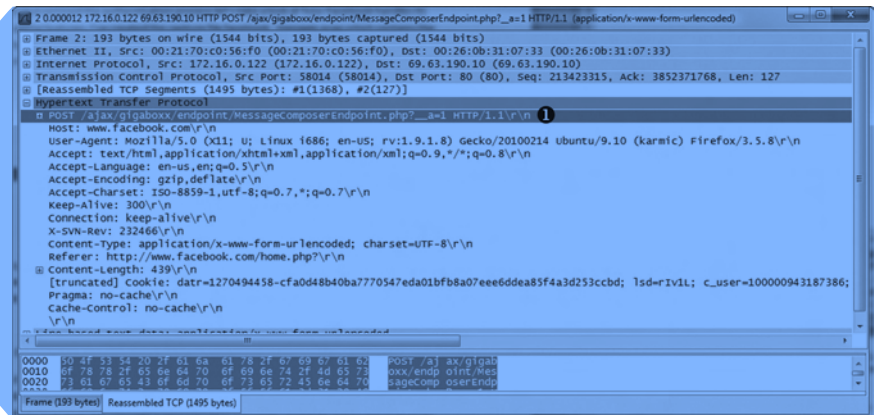
No.	Time	Source	Destination	Protocol	Info
62	0.000000	172.16.0.122	69.63.180.173	TCP	58637 → 80 [FIN, ACK] Seq=1272384963 Ack=2937930926 Win=1002 Len=0 TSval=302065222 TSecr=3479125856
63	0.000033	172.16.0.122	69.63.180.173	TLSv1	Encrypted Alert
64	0.000022	172.16.0.122	69.63.180.173	TCP	54595 → 443 [FIN, ACK] Seq=2917406980 Ack=2894040467 Win=158 Len=0 TSval=302065222 TSecr=3479125856
65	0.036439	69.63.180.173	172.16.0.122	TCP	80 → 58637 [ACK] Seq=2937930926 Ack=1272384964 Win=7233 Len=0 TSval=3479459532 TSecr=302065222
66	0.000082	69.63.180.173	172.16.0.122	TCP	80 → 58637 [FIN, ACK] Seq=2937930926 Ack=1272384964 Win=7233 Len=0 TSval=3479459532 TSecr=302065222
67	0.000023	172.16.0.122	69.63.180.173	TCP	58637 → 80 [ACK] Seq=1272384964 Ack=2937930927 Win=1002 Len=0 TSval=302065232 TSecr=3479459532
68	0.000033	69.63.180.173	172.16.0.122	TCP	443 → 54595 [ACK] Seq=2894040467 Ack=2917406980 Win=5496 Len=0 TSval=302065232 TSecr=3479459532
69	0.000078	172.16.0.122	69.63.180.173	TCP	54595 → 443 [ACK] Seq=2917406980 Ack=2894040467 Win=158 Len=0 TSval=302065285 TSecr=3479459532
70	0.459711	172.16.0.122	69.63.180.173	TCP	54595 → 443 [FIN, ACK] Seq=2917406979 Ack=2894040467 Win=158 Len=0 TSval=302065280 TSecr=3479459532
71	0.086948	69.63.180.173	172.16.0.122	TCP	443 → 54595 [ACK] Seq=2894040467 Ack=2917406980 Win=5496 Len=0 TSval=302065360 TSecr=302065360

Figure 8-9: The HTTP connection is torn down and is followed by the HTTPS connection.

facebook_
message.pcap

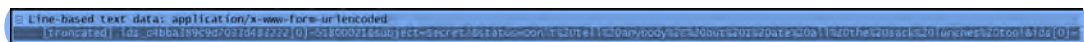
Now that we've examined Facebook's login authentication process, let's see how it handles private messaging. The file *facebook_message.pcap* contains the packets captured while sending a message from my account to another Facebook account. When you open this file, you may be surprised by the few packets it contains.

The first two packets comprise the HTTP traffic responsible for sending the message itself. When you expand the HTTP header of packet 2, as shown in Figure 8-10, you will see the POST method is used with a rather long URL string ①. As you can see, the string includes a reference to AJAX.



Asynchronous JavaScript and XML (AJAX) is a client-side approach to creating interactive web applications that retrieve information from a server in the background. While you might expect that after the private message is sent to the client's browser, the session would be redirected to another page (as with the Twitter direct message), that doesn't happen. In this case, the use of AJAX probably means that the message is sent from some type of interactive pop-up, rather than from an individual page, which means that no redirection or reloading of content is necessary. This is one of the benefits of some AJAX implementations.

You can examine the content of this private message by expanding the Line-based Text Data portion of packet 2, as shown in Figure 8-11. As with Twitter, it appears as though Facebook's private messages are sent unencrypted.



Comparing Twitter vs. Facebook Methods

You've now seen the authentication and messaging methods of two web services: Twitter and Facebook. Each takes a different approach. Programmers might argue that the Twitter method of authentication is better than Facebook's because it can be faster and more efficient. Security researchers might argue the Facebook method is better because it ensures that all content has been delivered. Also, no additional authentication is required before the authentication connection closes, which may in turn make man-in-the-middle attacks more difficult to achieve. (*Man-in-the-middle attacks* are attacks where malicious users intercept traffic between two communicating parties.) In reality, the differences between the authentication methods of the two websites are minimal, but they do demonstrate that differences can occur when two programmers set out to develop a routine that performs the same task.

Although it's interesting, the point of this analysis was not to find out exactly how Twitter and Facebook work but simply to expose you to traffic that you can compare and contrast. This baseline should provide a good framework if you need to examine why similar services aren't operating as they should or are just operating slowly.

Capturing ESPN.com Traffic

`http_espn.pcap`

Having completed my social network stalking for the morning, my next task is to check up on the latest news headlines and sports scores. Certain sites always make for interesting analysis, and <http://www.espn.com/> is one of those. I've captured the traffic of a computer browsing to the ESPN website in the file `http_espn.pcap`.

This capture file contains many packets—956 to be exact. This is simply too much data for us to manually scroll through the entire file in an attempt to discern individual connections and anomalies, so we'll use some of Wireshark's analysis features to make the process easier.

Using the Conversations Window

The ESPN home page includes a lot of bells and whistles, so it's easy to understand why it would take nearly a thousand packets to transfer that data to us. Whenever you have a large data transfer, it's useful to know the source of that data, and more important, whether it's from one or multiple sources. We can find out by using Wireshark's Conversations window (Statistics ► Conversations).

Figure 8-12 shows an example with 14 unique IP conversations, 25 unique TCP connections, and 14 unique UDP conversations—all displayed in detail in the main Conversations window. There's a lot going on here, especially for just one site.

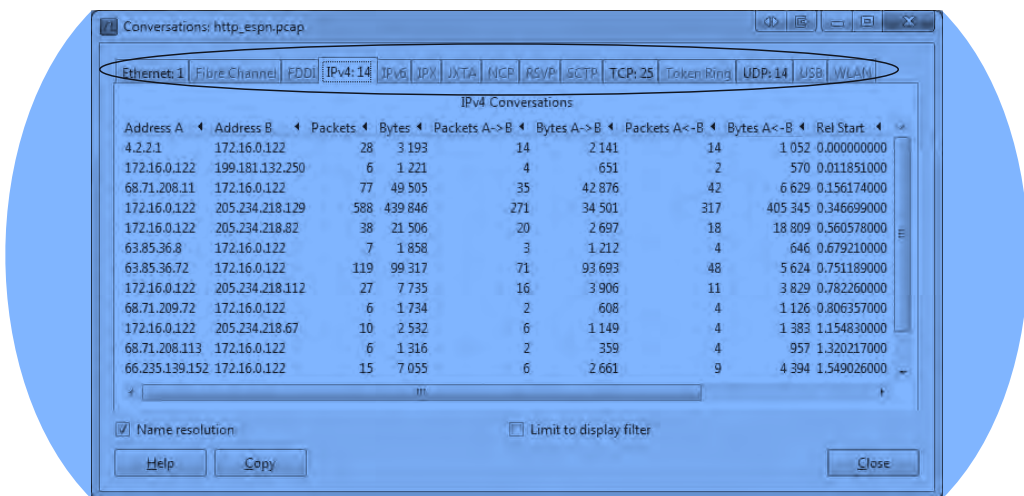


Figure 8-12: The Conversations window shows several unique connections.

Using the Protocol Hierarchy Statistics Window

For a better view of the situation, we can see the application layer protocols used with these TCP and UDP connections. Open the Protocol Hierarchy Statistics window (Statistics ► Protocol Hierarchy), as shown in Figure 8-13.

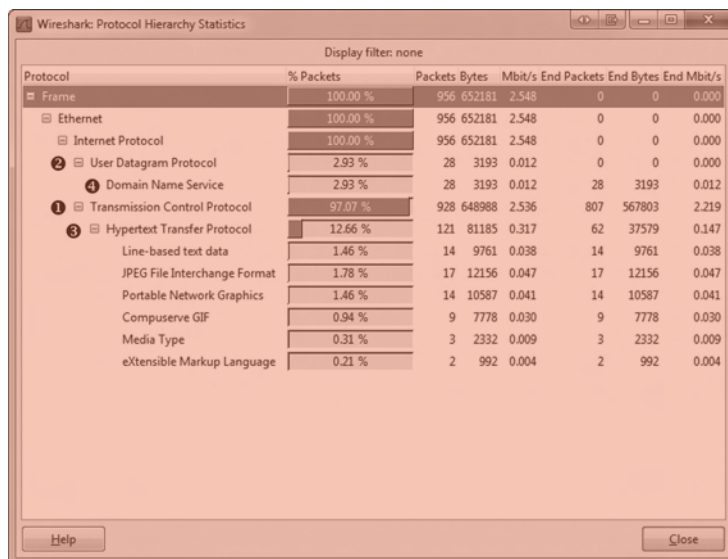


Figure 8-13: The Protocol Hierarchy Statistics window shows the distribution of protocols in this capture.

As you can see, TCP accounts for 97.07 percent of the packets in the capture ❶, and UDP accounts for the remaining 2.93 percent ❷. As expected, the TCP traffic is all HTTP ❸, which is broken down even further into the file types transferred over HTTP.

It may seem confusing when I say that all of the TCP traffic is HTTP when Wireshark shows only 12.66 percent as being HTTP, but that's because the other 84.41 percent is pure TCP traffic (data transfer and control packets). All of the UDP traffic shown is DNS, based on the entry under the UDP heading ❹.

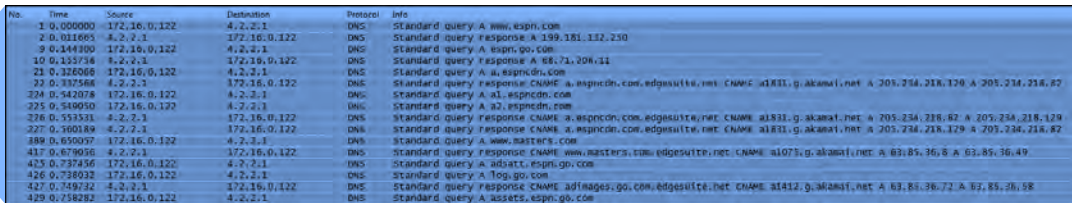
Based on this information alone, we can draw a few conclusions. For one, we know from previous examples that DNS transactions are quite small, so the fact there are 28 DNS packets (as listed in the Packets column next to the Domain Name Service entry in Figure 8-13) means that we could have as many as 14 DNS transactions. We derive this number by dividing the total number of packets by two, which represents pairs of requests and responses. If you look under the UDP heading of the Conversations window it will show that there are indeed 14 conversations, which accounts for each DNS transaction and confirms our assumption.

DNS queries don't happen on their own though, and the only other traffic in the capture is HTTP traffic. This tells us that it's likely that the HTML code within the ESPN website references other domains or subdomains by DNS name, thus forcing multiple queries to be executed.

Let's see if we can find some evidence to substantiate our theories.

Viewing DNS Traffic

One simple way to view DNS traffic is to create a filter. Entering `dns` into the filter section of the Wireshark window shows all of the DNS traffic, as shown in Figure 8-14.



No.	Time	Source	Destination	Protocol	Info
1	0.000000	172.16.0.122	4.2.2.1	DNS	Standard query A www.espn.com
2	0.011695	4.2.2.1	172.16.0.122	DNS	Standard query response A 199.181.132.230
9	0.144300	172.16.0.122	4.2.2.1	DNS	Standard query A espn.go.com
10	0.155734	4.2.2.1	172.16.0.122	DNS	Standard query response A 68.71.204.11
21	0.326008	172.16.0.122	4.2.2.1	DNS	Standard query A a.espn.com
22	0.337568	4.2.2.1	172.16.0.122	DNS	Standard query response CNAME a.espn.com.edgesuite.net CNAME a1831.g.akamai.net A 205.234.218.129 A 205.234.218.82
224	0.542078	172.16.0.122	4.2.2.1	DNS	Standard query A a1.espn.com
225	0.549010	172.16.0.122	4.2.2.1	DNS	Standard query A a2.espn.com
226	0.553531	4.2.2.1	172.16.0.122	DNS	Standard query response CNAME a.espn.com.edgesuite.net CNAME a1831.g.akamai.net A 205.234.218.82 A 205.234.218.129
227	0.560189	4.2.2.1	172.16.0.122	DNS	Standard query response CNAME a.espn.com.edgesuite.net CNAME a1831.g.akamai.net A 205.234.218.129 A 205.234.218.82
289	0.650097	172.16.0.122	4.2.2.1	DNS	Standard query A www.masters.com
417	0.879036	4.2.2.1	172.16.0.122	DNS	Standard query response CNAME www.masters.com.edgesuite.net CNAME a1073.g.akamai.net A 63.85.36.6 A 63.85.36.49
425	0.737456	172.16.0.122	4.2.2.1	DNS	Standard query A adsatt.espn.go.com
426	0.738032	172.16.0.122	4.2.2.1	DNS	Standard query A log.go.com
427	0.749222	4.2.2.1	172.16.0.122	DNS	Standard query response CNAME adimages.go.com.edgesuite.net CNAME a1412.g.akamai.net A 63.85.36.72 A 63.85.36.58
429	0.758282	172.16.0.122	4.2.2.1	DNS	Standard query A assets.espn.go.com

Figure 8-14: The DNS traffic appears to be standard queries and responses.

This DNS traffic shown in Figure 8-14 appears to all be queries and responses. For a better view of the DNS names being queried, create a filter that displays only the queries. To create this filter, select a query in the Packet List pane and expand the packet's DNS header in the Packet Details pane. Then right-click the Flags: 0x0100 (Standard query) field, hover over **Apply as Filter**, and choose **Selected**.

This should activate the filter `dns.flags == 0x0100`, which shows only the queries and makes it much easier to read the records we're analyzing. And, as you can see in Figure 8-14, there are indeed 14 individual queries (each packet represents a query), and all of the domain names seem to be associated with ESPN or the content displayed on its home page.

Viewing HTTP Requests

Finally, we can verify the source of these queries by examining the HTTP requests. To do so, select **Statistics ▸ HTTP**, select **Requests**, and click **Create Stat.** (Make sure the filter you just created is cleared before doing this.)

Figure 8-15 shows the HTTP Requests window. The 14 connections shown here (each line represents a connection to a particular domain) account for all of the domains represented by the DNS queries.

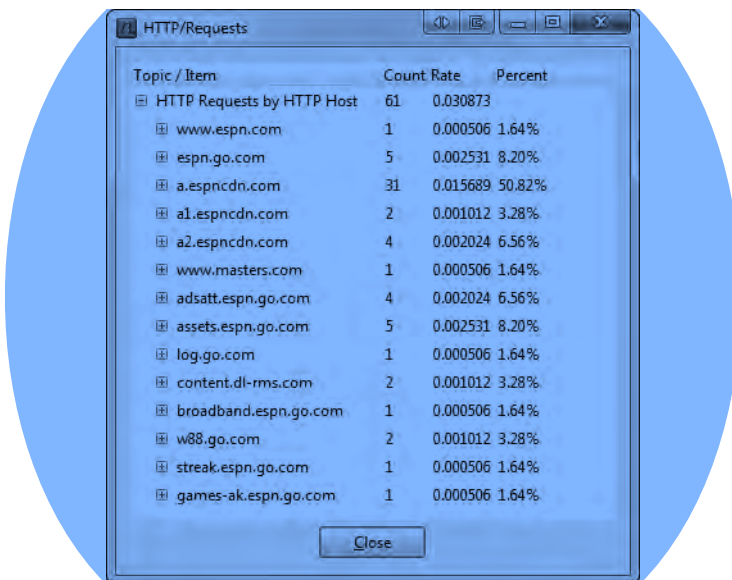


Figure 8-15: All HTTP requests are summarized in this window, which shows the domains accessed.

With this many connections occurring, it may be in our best interest to check whether this highly involved process is taking place in a timely manner. The easiest way to do this is to view a summary of the traffic. To do so, choose **Statistics ▸ Summary**. The Summary window, shown in Figure 8-16, shows that the entire process occurs in about 2 seconds, which is perfectly acceptable.

It's odd to think that our simple request to view a web page broke into requests for 14 separate domains and subdomains, touching a variety of different servers, and that this whole process took place in only 2 seconds. Capturing traffic while visiting your favorite websites and breaking it down as we have here is an interesting exercise. You never know quite where your data is coming from until you start looking at packets.

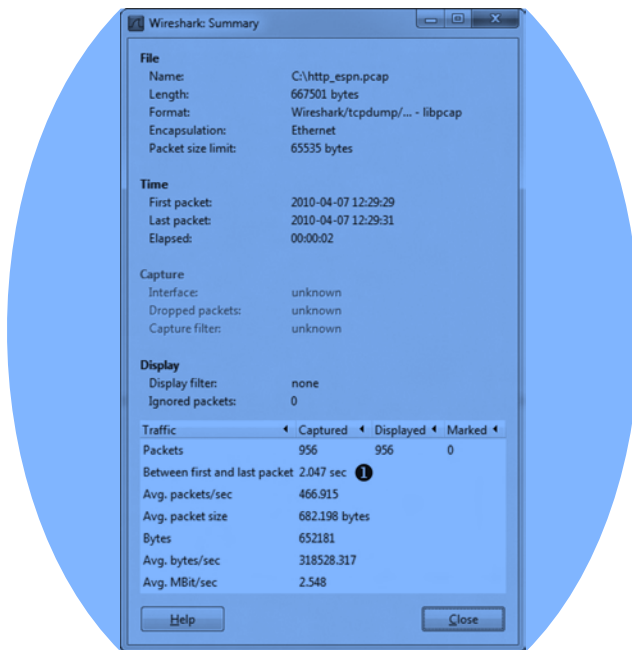


Figure 8-16: The Summary window for the file shows that this entire process occurs in two seconds.

Real-World Problems

We'll now shift to some examples of problematic traffic. Let's look at various Internet access problems, as well as typical problems like an unreliable printer and a connectivity issue from a branch office.

No Internet Access: Configuration Problems

The first problem scenario is rather simple: A user cannot access the Internet. We have verified that the user can access all of the internal resources of the network, including shares on other workstations and applications hosted on local servers.

The network architecture is fairly straightforward, as all clients and servers connect to a series of simple switches. Internet access is handled through a single router serving as the default gateway, and IP addressing information is provided by DHCP. This is a very common scenario in small offices.

Tapping into the Wire

nowebaccess1.pcap

In order to determine the cause of the issue, we can have the user attempt to browse the Internet while our sniffer is listening on the wire. We use the information from "Sniffer Placement in Practice" on page 31 (see Figure 2-15)

to determine the most appropriate method for placing our sniffer.

The switches on our network do not support port mirroring. We already have to interrupt the user in order to conduct our test, so we can assume that it is okay to take him offline once again. (That said, using a tap would be the most appropriate way to tap into the wire.) The resulting file is *nowebaccess1.pcap*.

Analysis

The traffic capture begins with an ARP request and reply, as shown in Figure 8-17. In packet 1, the user's computer, with MAC address 00:25:b3:bf:91:ee and IP address 172.16.0.8, sends an ARP broadcast packet to all computers on the network segment in an attempt to find the MAC address associated with the IP address of its default gateway, 172.16.0.10.

No.	Time	Source	Destination	Protocol	Info
1	0.000000	00:25:b3:bf:91:ee	ff:ff:ff:ff:ff:ff	ARP	who has 172.16.0.10? Tell 172.16.0.8
2	0.000090	00:24:81:a1:f6:79	00:25:b3:bf:91:ee	ARP	172.16.0.10 is at 00:24:81:a1:f6:79

Figure 8-17: ARP request and reply for the computer's default gateway

A response is received in packet 2, and the user's computer learns that 172.16.0.10 is at 00:24:81:a1:f6:79. Once this reply is received, the computer now has a route to a gateway that should be able to direct it to the Internet.

Following the ARP reply, the computer must attempt to resolve the DNS name of the website to an IP address using DNS in packet 3. As shown in Figure 8-18, the computer does this by sending a DNS query packet to its primary DNS server, 4.2.2.2.



Figure 8-18: A DNS query sent to 4.2.2.2

Under normal circumstances, a DNS server would respond to a DNS query very quickly, but that's not the case here. Rather than a response, we see the same DNS query sent a second time to a different destination address. As shown in Figure 8-19, in packet 4, the second DNS query is sent to the secondary DNS server configured on the computer, which is 4.2.2.1.



Figure 8-19: A second DNS query sent to 4.2.2.1

Again, no reply is received from the DNS server, and the query is sent again one second later to 4.2.2.2. This process repeats itself, alternating the destination packets between the primary ❶ and secondary ❷ configured DNS servers over the next several seconds, as shown in Figure 8-20. The entire process takes around 8 seconds ❸, which is how long it takes before the user's Internet browser reports that a website is inaccessible.

No.	Time	Source	Destination	Protocol	Info
1	0.000000	00:25:b3:bf:91:ee	ff:ff:ff:ff:ff:ff	ARP	who has 172.16.0.10? Tell 172.16.0.8
2	0.000090	00:24:81:a1:f6:79	00:25:b3:bf:91:ee	ARP	172.16.0.10 is at 00:24:81:a1:f6:79
3	0.000105	172.16.0.8	4.2.2.2 ❶	DNS	Standard query A www.google.com
4	0.999280	172.16.0.8	4.2.2.1 ❷	DNS	Standard query A www.google.com
5	1.999279	172.16.0.8	4.2.2.2	DNS	Standard query A www.google.com
6	3.999372	172.16.0.8	4.2.2.1	DNS	Standard query A www.google.com
7	3.999393	172.16.0.8	4.2.2.2	DNS	Standard query A www.google.com
8	7.999627	172.16.0.8	4.2.2.1	DNS	Standard query A www.google.com
9	7.999648	172.16.0.8	4.2.2.2	DNS	Standard query A www.google.com

Figure 8-20: DNS queries repeated until communication stops

Based on the packets we've seen, we can begin pinpointing the source of the problem. First, we see a successful ARP request to what we believe is the default gateway router for the network, so we know that device is online and communicating. We also know that the user's computer is actually transmitting packets on the network, so we can assume there isn't an issue with the protocol stack on the computer itself. The problem clearly begins to occur when the DNS request is made.

In the case of this network, DNS queries are resolved by an external server on the Internet (4.2.2.2 or 4.2.2.1). This means that in order for resolution to take place correctly, the router responsible for routing packets to the Internet must successfully forward the DNS queries to the server, and the server must respond. This all must happen before HTTP can be used to request the web page itself.

We know that no other users are having issues connecting to the Internet, which tells us that the network router and remote DNS server are probably not the source of the problem. The only thing remaining as the potential source of the problem is the user's computer itself.

Upon deeper examination of the affected computer, we find that rather than receiving a DHCP-assigned address, the computer has manually assigned addressing information, and the default gateway address is actually set incorrectly. The address set as the default gateway is not a router and cannot forward the DNS query packets outside the network.

Lessons Learned

The problem in this scenario resulted from a misconfigured client. While the problem itself turned out to be quite simple, it significantly impacted the user. Troubleshooting a simple misconfiguration like this one could take quite some time for someone lacking knowledge of the network or the ability to perform a quick packet analysis as we've done here. As you can see, packet analysis is not limited to large and complex problems.

Notice that because we didn't enter the scenario knowing the IP address of the network's gateway router, Wireshark didn't identify the problem exactly, but it did tell us where to look, saving valuable time. Rather than examining the gateway router, contacting our ISP, or trying to find the resources to troubleshoot the remote DNS server, we were able to focus our troubleshooting efforts on the computer itself, which was, in fact, the source of the problem.

NOTE *Had we been more familiar with this particular network's IP addressing scheme, analysis could have been even faster. The problem could have been identified immediately once we noticed that the ARP request was sent to an IP address different from that of the gateway router. These simple misconfigurations are often the source of network problems and can typically be resolved quickly with a bit of packet analysis.*

No Internet Access: Unwanted Redirection

In this scenario, we once again have a user who is unable to access the Internet from her workstation. However, unlike the user in the previous scenario, this user can access the Internet, but she cannot access her home page, <http://www.google.com/>. When the user attempts to reach any domain hosted by Google, she is directed to a browser page that says "Internet Explorer cannot display the web page." This issue is affecting only this particular user.

As with the previous scenario, this is a small network with a few simple switches and a single router serving as the default gateway.

Tapping into the Wire

`nowebaccess2.pcap`

To begin our analysis, we have the user attempt to browse to <http://www.google.com/> while we listen to the traffic that is generated using a tap. The resulting file is `nowebaccess2.pcap`.

Analysis

The capture begins with an ARP request and reply, as shown in Figure 8-21. In packet 1, the user's computer, with MAC address 00:25:b3:bf:91:ee and IP address of 172.16.0.8, sends an ARP broadcast packet to all computers on the network segment in an attempt to find the MAC address associated with the host's IP address 172.16.0.102. We don't immediately recognize this address.

No.	Time	Source	Destination	Protocol	Info
1	0.000000	00:25:b3:bf:91:ee	ff:ff:ff:ff:ff:ff	ARP	who has 172.16.0.102? Tell 172.16.0.8
2	0.000334	00:21:70:c0:56:f0	00:25:b3:bf:91:ee	ARP	172.16.0.102 is at 00:21:70:c0:56:f0

Figure 8-21: ARP request and reply for another device on the network

In packet 2, the user's computer learns that the IP address 172.16.0.102 is at 00:21:70:c0:56:f0. Based on the previous scenario, we might assume that this is the gateway router's address, and that address is used so that packets can once again be forwarded to the external DNS server. However, as shown in Figure 8-22, the next packet is not a DNS request, but a TCP packet from 172.16.0.8 to 172.16.0.102. It has the SYN flag set, indicating that this is the first packet in the handshake for a new TCP-based connection between the two hosts ①.

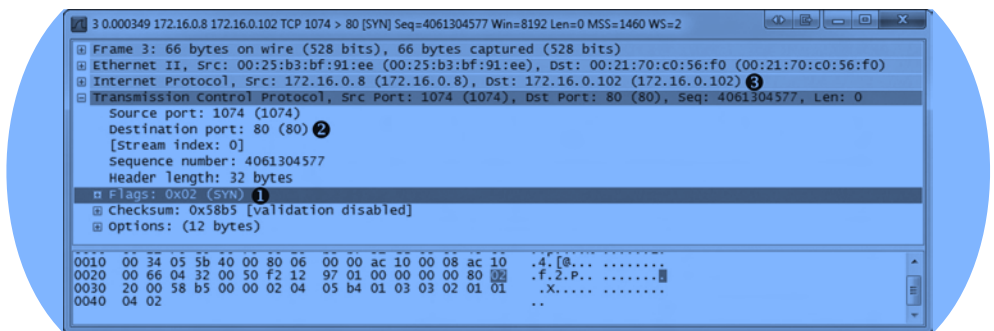


Figure 8-22: TCP SYN packet sent from one internal host to another

Notably, the TCP connection attempt is to port 80 ② on 172.16.0.102 ③, which is typically associated with HTTP traffic. As shown in Figure 8-23, this connection attempt is abruptly halted when the host 172.16.0.102 sends a TCP packet in response (packet 4) with the RST and ACK flags set ①.

Recall from Chapter 6 that a packet with the RST flag set is used to terminate a TCP connection. However, in this scenario, the host at 172.16.0.8 attempted to establish a TCP connection to the host at 172.16.0.102 on port 80. Unfortunately, because that host has no services configured to listen to requests on port 80, the TCP RST packet is sent to terminate the connection. This process repeats twice. A SYN is sent from the user's computer and replied to with a RST, before communication finally ends, as shown in Figure 8-24. At this point, the user receives a message in her browser saying that the page cannot be displayed.

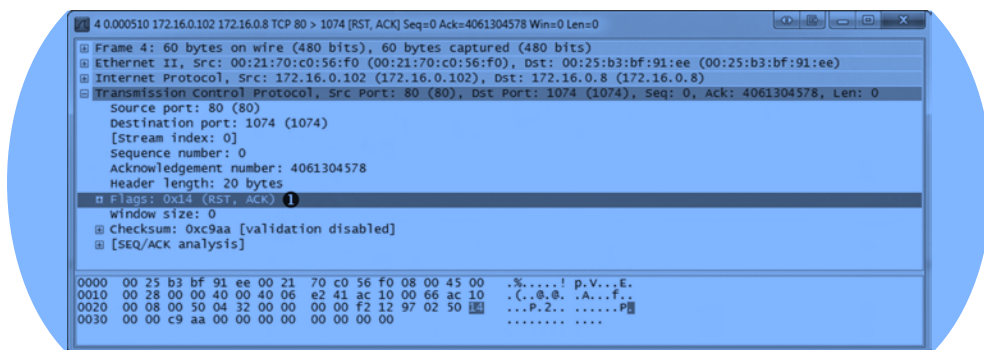


Figure 8-23: TCP RST packet sent in response to the TCP SYN

No.	Time	Source	Destination	Protocol	Info
3	0.000349	172.16.0.8	172.16.0.102	TCP	1074 > 80 [SYN] Seq=4061304577 win=8192 Len=0 MSS=1460 WS=2
4	0.000510	172.16.0.102	172.16.0.8	TCP	80 > 1074 [RST, ACK] Seq=0 Ack=4061304578 win=0 Len=0
5	0.499162	172.16.0.8	172.16.0.102	TCP	1074 > 80 [SYN] Seq=4061304577 win=8192 Len=0 MSS=1460 WS=2
6	0.499362	172.16.0.102	172.16.0.8	TCP	80 > 1074 [RST, ACK] Seq=0 Ack=4061304578 win=0 Len=0
7	0.999190	172.16.0.8	172.16.0.102	TCP	1074 > 80 [SYN] Seq=4061304577 win=8192 Len=0 MSS=1460
8	0.999507	172.16.0.102	172.16.0.8	TCP	80 > 1074 [RST, ACK] Seq=0 Ack=4061304578 win=0 Len=0

Figure 8-24: The TCP SYN and RST packets are seen three times in total.

After examining the configuration of another network device that is working correctly, the ARP request and reply in packets 1 and 2 concern us because the ARP request is not for the gateway router's actual MAC address, but some unknown device. Following the ARP request and reply, we would expect to see a DNS query sent to our configured DNS server in order to find the IP address associated with *www.google.com*, but we don't. There are two conditions that could prevent a DNS query from being made:

- The device initiating the connection already has the DNS name-to-IP address mapping in its DNS cache.
- The device connecting to the DNS name already has the DNS name-to-IP address mapping specified in its *hosts* file.

Upon further examination of the client computer, we find that the computer's *hosts* file has an entry for *www.google.com* associated with the internal IP address 172.16.0.102. This erroneous entry is the source of our user's problems.

A computer will typically use its *hosts* file as the authoritative source for DNS name-to-IP address mappings, and will check that file before querying an outside source. In this scenario, the user's computer checked its *hosts* file, found the entry for *www.google.com*, and decided that *www.google.com* was actually on its own local network segment. Next, it sent an ARP request to the host, received a response, and attempted to initiate a TCP connection to 172.16.0.102 on port 80. However, because the remote system was not configured as a web server, it would not accept the connection attempts.

Once the *hosts* file entry was removed, the user's computer began communicating correctly and was able to access *www.google.com*.

NOTE To examine your *hosts* file on a Windows system, open `C:\Windows\System32\drivers\etc\hosts`. On Linux, view `/etc/hosts`.

This scenario is actually very common. It's one that malware has been using for years to redirect users to websites hosting malicious code. Imagine if an attacker were to modify your *hosts* file so that every time you went to do your online banking, it actually redirected you to a fake site designed to steal your account credentials!

Lessons Learned

As you continue to analyze traffic, you will learn both how the various protocols work and how to break them. In this scenario, the DNS query was not sent because the client was misconfigured, not because of any external limitations or misconfigurations.

By examining this problem at the packet level, we were able to quickly spot an IP address that was unknown and also to determine that DNS, a key component of this communication process, was missing. Using this information, we were able to identify the client as the source of the problem.

No Internet Access: Upstream Problems

As with the previous two scenarios, in this scenario, a user complains of no Internet access from his workstation. This user has narrowed the issue down to a single website, `http://www.google.com/`. Upon further investigation, we find that this issue is affecting everyone in the organization—no one can access Google domains.

The network is configured as in the two prior scenarios, with a few simple switches and a single router connecting the network to the Internet.

Tapping into the Wire

In order to troubleshoot this issue, we first browse to `http://www.google.com/` to generate traffic. Because this issue is network wide—meaning it's also affecting your computer, and it could be the result of a massive malware infection—you shouldn't sniff directly from your device. When you find yourself in a situation like this, a tap is the best solution, because it allows you to be completely passive after a brief interruption of service. The file resulting from the capture via a tap is `nowebaccess3.pcap`.

Analysis

This packet capture begins with DNS traffic instead of the ARP traffic we are used to seeing. Because the first packet in the capture is to an external address, and packet 2 contains a reply from that address, we can assume that the ARP process has already happened and the MAC-to-IP address mapping for our gateway router already exists in the host's ARP cache at 172.16.0.8.

As shown in Figure 8-25, the first packet in the capture is from the host 172.16.0.8 to address 4.2.2.1 ❶, and it's a DNS packet ❷. Examining the contents of the packet, we see that it is a query for the A record for `www.google.com` ❸.

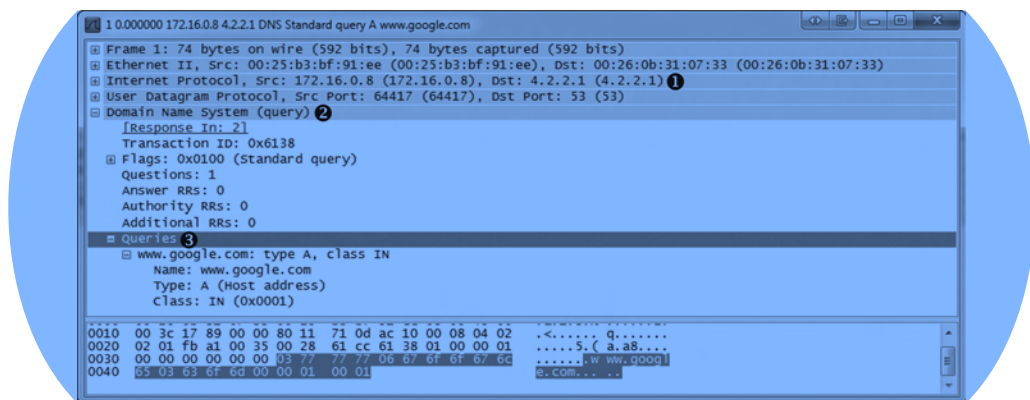


Figure 8-25: DNS query for www.google.com A record

The response to the query from 4.2.2.1 is the second packet in the capture file, as shown in Figure 8-26. Examining the Packet Details pane, we see that the name server that responded to this request provided multiple answers to the query ❶. At this point, all looks well, and communication is occurring as it should.

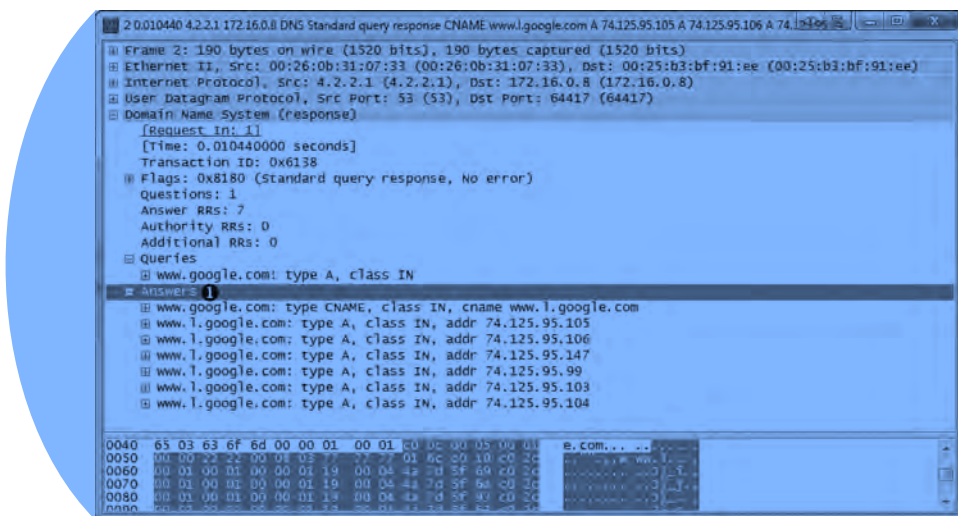


Figure 8-26: DNS reply with multiple A records

Now that the user's computer has determined the web server's IP address, it can attempt to communicate with the server. As shown in Figure 8-27, this process is initiated in packet 3, with a TCP packet sent from 172.16.0.8 to 74.125.95.105 ❶. This destination address comes from the first A record provided in the DNS query response seen in packet 2. The TCP packet has the SYN flag set ❷, and it's attempting to communicate with the remote server on port 80 ❸.

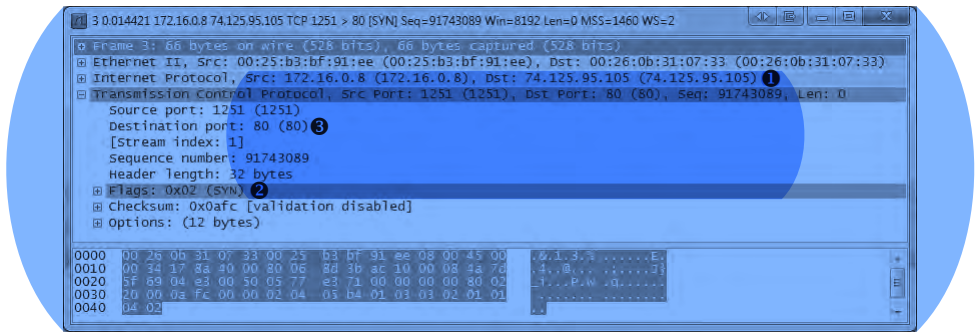


Figure 8-27: SYN packet attempting to initiate a connection on port 80

Because this is a TCP handshake process, we know that we should see a TCP SYN/ACK packet sent in response, but instead, after a short time, another SYN packet is sent from the source to the destination. This process occurs once more after approximately a second, as shown in Figure 8-28, at which point communication stops and the browser reports that the website could not be found.

No.	Time	Source	Destination	Protocol	Info
3	0.014421	172.16.0.8	74.125.95.105	TCP	1251 > 80 [SYN] Seq=91743089 Win=8192 Len=0 MSS=1460 WS=2
4	0.019417	172.16.0.8	74.125.95.105	TCP	1251 > 80 [SYN] Seq=91743089 Win=8192 Len=0 MSS=1460 WS=2
5	1.016531	172.16.0.8	74.125.95.105	TCP	1251 > 80 [SYN] Seq=91743089 Win=8192 Len=0 MSS=1460 WS=2

Figure 8-28: The TCP SYN packet is attempted three times with no response received.

As we troubleshoot this scenario, we consider that we know that the workstation within our network can connect to the outside world because the DNS query to our external DNS server at 4.2.2.1 is successful. The DNS server responds with what appears to be a valid address, and our hosts attempt to connect to one of those addresses. Also, the local workstation we are attempting to connect from appears to be functioning.

The problem is that the remote server simply isn't responding to our connection requests; a TCP RST packet is not sent. This might occur for several reasons: a misconfigured web server, a corrupted protocol stack on the web server, or a packet-filtering device on the remote network (such as a firewall). Assuming there is no local packet filtering device in place, all of the other potential solutions are on the remote network and beyond our control. In this case, the web server was not functioning correctly, and no attempt to access it succeeded. Once the problem was corrected on Google's end, communication was able to proceed.

Lessons Learned

In this scenario, the problem was not one that we could correct. Our analysis determined that the issue was not with the hosts on our network, our router, or the external DNS server providing us with name resolution services. The issue lay outside our network infrastructure.

Sometimes discovering that a problem isn't really ours can not only relieve stress, but also save face when management comes knocking. I have fought

with many ISPs, vendors, and software companies who claim that an issue is not their fault, but as you've just seen, packets don't lie.

Inconsistent Printer

Our IT help desk administrator is having trouble resolving a printing issue. Users in the sales department are reporting that the high-volume sales printer is malfunctioning. When a user sends a large print job to the printer, it will print several pages and then stop printing before the job is done. Multiple driver configuration changes have been attempted but have been unsuccessful. The help desk staff would like you to ensure that this isn't a network problem.

Tapping into the Wire

inconsistent_
printer.pcap

The common thread in relation to this problem is the printer, so we want to begin by placing our sniffer as close to the printer as we can. While we can't install Wireshark on the printer itself, the switches used in this network are advanced layer 3 switches, so we can use port mirroring. We'll mirror the port to which the printer is connected to an empty port, and connect a laptop with Wireshark installed into this port. Once this setup is complete, we'll have a user send a large print job to the printer, and we'll monitor the output. The resulting capture file is *inconsistent_printer.pcap*.

Analysis

As shown in Figure 8-29, a TCP handshake between the network workstation sending the print job (172.16.0.8) and the printer (172.16.0.253) initiates the connection at the start of the capture file. Following the handshake, a TCP data packet 1,460 bytes in size is sent to the printer ❶. The amount of data can be seen in the far right side of the Info column in the Packet List pane or at the bottom of the TCP header information in the Packet Details pane.

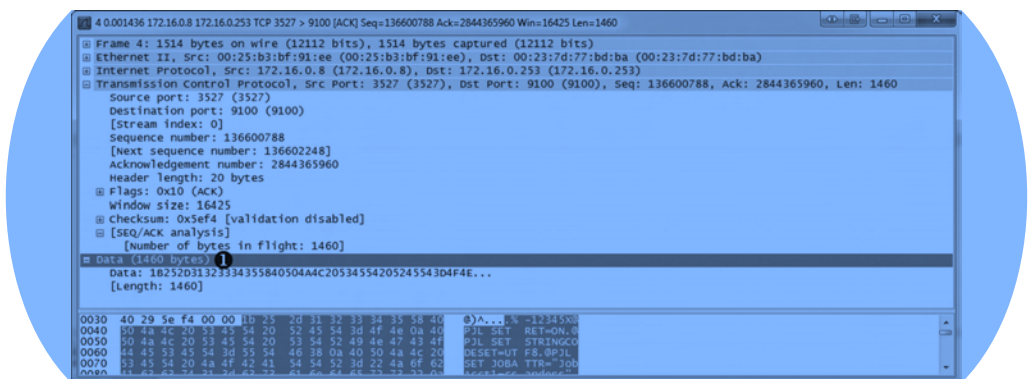


Figure 8-29: Data being transmitted to the printer over TCP

Following packet 4, another data packet is sent containing 1,460 bytes of data ❶, as you can see in Figure 8-30. This data is acknowledged by the printer ❷.

No.	Time	Source	Destination	Protocol	Info
3	0.000035	172.16.0.8	172.16.0.253	TCP	3527 > 9100 [ACK] Seq=136600788 Ack=2844365960 Win=16425 Len=0
4	0.001436	172.16.0.8	172.16.0.253	TCP	3527 > 9100 [ACK] Seq=136600788 Ack=2844365960 Win=16425 Len=1460
5	0.000009	172.16.0.8	172.16.0.253	TCP	3527 > 9100 [ACK] Seq=136602248 Ack=2844365960 Win=16425 Len=1460 ❶
6	0.003847	172.16.0.253	172.16.0.8	TCP	9100 > 3527 [PSH, ACK] Seq=2844365960 Ack=136603708 Win=7888 Len=106
7	0.000068	172.16.0.8	172.16.0.253	TCP	3527 > 9100 [ACK] Seq=136603708 Ack=2844366066 Win=16398 Len=1460
8	0.000010	172.16.0.8	172.16.0.253	TCP	3527 > 9100 [ACK] Seq=136605168 Ack=2844366066 Win=16398 Len=1460
9	0.000007	172.16.0.8	172.16.0.253	TCP	3527 > 9100 [ACK] Seq=136606628 Ack=2844366066 Win=16398 Len=1460
10	0.000007	172.16.0.8	172.16.0.253	TCP	3527 > 9100 [ACK] Seq=136608088 Ack=2844366066 Win=16398 Len=1460
11	0.027984	172.16.0.253	172.16.0.8	TCP	9100 > 3527 [ACK] Seq=2844366066 Ack=136609548 Win=6144 Len=0
12	0.000057	172.16.0.8	172.16.0.253	TCP	3527 > 9100 [ACK] Seq=136609548 Ack=2844366066 Win=16398 Len=1460
13	0.000014	172.16.0.8	172.16.0.253	TCP	3527 > 9100 [ACK] Seq=136611008 Ack=2844366066 Win=16398 Len=1460
14	0.000009	172.16.0.8	172.16.0.253	TCP	3527 > 9100 [ACK] Seq=136612468 Ack=2844366066 Win=16398 Len=1460
15	0.000009	172.16.0.8	172.16.0.253	TCP	3527 > 9100 [ACK] Seq=136613928 Ack=2844366066 Win=16398 Len=1460
16	0.064656	172.16.0.253	172.16.0.8	TCP	9100 > 3527 [ACK] Seq=2844366066 Ack=136615388 Win=4400 Len=0

Figure 8-30: Normal data transmission and TCP acknowledgments

The flow of data continues until the last two packets in the capture. Packet 121 is a TCP retransmission packet, and our first sign of trouble, as shown in Figure 8-31.

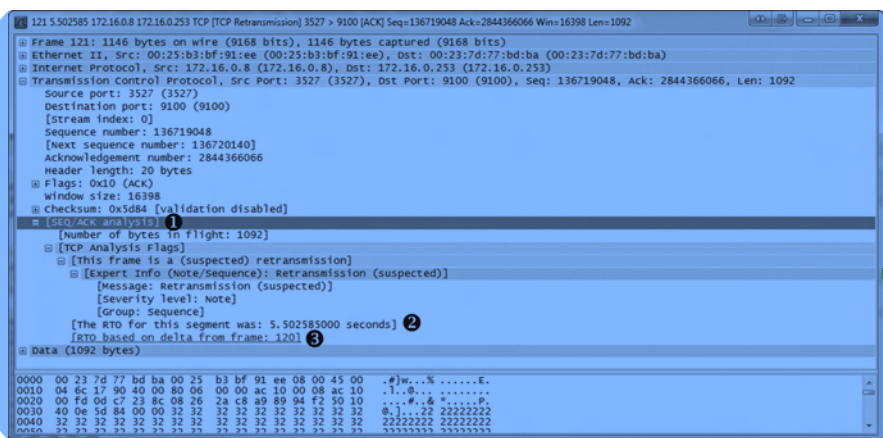


Figure 8-31: These TCP retransmission packets are a sign of a potential problem.

A TCP retransmission packet is sent when one device sends a TCP packet to a remote device, and the remote device doesn't acknowledge the transmission. Once a retransmission threshold is reached, the sending device assumes that the remote device did not receive the data, and it retransmits the packet. This process is repeated a few times before communication effectively stops.

In this scenario, the retransmission is sent from the client workstation to the printer because the printer failed to acknowledge the transmitted data. If you expand the SEQ/ACK analysis portion of the TCP header along with the additional information beneath it, as shown in Figure 8-31 ❶, you can view the details of why this is a retransmission. According to the details processed by Wireshark, packet 121 is a retransmission of packet 120 ❷. Additionally, the retransmission timeout (RTO) for the retransmitted packet was around 5.5 seconds ❸.

When analyzing the delay between packets, you can change the time display format to suit your situation. In this case, because we want to see how long the retransmissions occurred after the previous packet was sent, change this option by selecting **View ▶ Time Display Format** and select **Seconds Since**

Previous Captured Packet. Then, as shown in Figure 8-32, you can clearly see that the retransmission in packet 121 occurs 5.5 seconds after the original packet (packet 120) is sent ❶.

No.	Time	Source	Destination	Protocol	Info
121	5.502389	172.16.0.8	172.16.0.253	TCP	[TCP Retransmission] 3527 → 9100 [ACK] Seq=136719048 Ack=2844366066 win=16398 Len=1092
122	5.600089	172.16.0.8	172.16.0.253	TCP	[TCP Retransmission] 3527 → 9100 [ACK] Seq=136719048 Ack=2844366066 win=16398 Len=1092

Figure 8-32: Viewing the time between packets is useful for troubleshooting.

The next packet is another retransmission of packet 120. The RTO of this packet is 11.10 seconds, which includes the 5.5 seconds from the RTO of the previous packet. A look at the Time column of the Packet List pane tells us that this retransmission was sent 5.6 seconds after the previous retransmission. This appears to be the last packet in the capture file and, coincidentally, the printer stops printing at approximately this time.

In this analysis scenario, we have the benefit of dealing with only two devices inside our own network, so we just need to determine whether the client workstation or the printer is to blame. We can see that data is flowing correctly for quite some time, and then at some point, the printer simply stops responding to the workstation. The workstation gives its best effort to get the data to its destination, as evidenced by the retransmissions, but the printer simply stops responding. This issue is reproducible and happens regardless of which computer sends a print job, so we assume the printer is the source of the problem.

After further analysis, we find that the printer's RAM is malfunctioning. When large print jobs are sent to the printer, it prints only a certain number of pages, likely until certain regions of memory are accessed. At that point, the memory issue causes the printer to be unable to accept any new data, and it ceases communication with the host transmitting the print job.

Lessons Learned

Although this printer problem was not the result of a network issue, we were able to use Wireshark to pinpoint the problem. Unlike previous scenarios, this one centered solely on TCP traffic. Luckily, TCP often leaves us with useful information when two devices simply stop communicating.

In this case, when communication abruptly stopped, we were able to pinpoint the exact location of the problem based on nothing more than TCP's built-in retransmission functionality. As we continue through our scenarios, we will often rely on functionality like this to troubleshoot more complex issues.

Stranded in a Branch Office

In this scenario, we have a company with a central headquarters and newly deployed remote branch offices. The company's IT infrastructure is mostly contained within the central office using a Windows server-based domain and a secondary domain controller. The domain controller is responsible for handling DNS and authentication requests for users at the branch office.

The domain controller is a secondary DNS server that should receive its resource record information from the upstream DNS servers at the corporate headquarters.

The deployment team is rolling out the new infrastructure to the branch office when it finds that no one can access the intranet web application servers on the network. These servers are located at the main office and are accessed through the wide area network (WAN). This issue affects all users at the branch office, and is limited to just these internal servers. All users can access the Internet and other resources within the branch.

Figure 8-33 shows the components to consider in this scenario, which involves multiple sites.

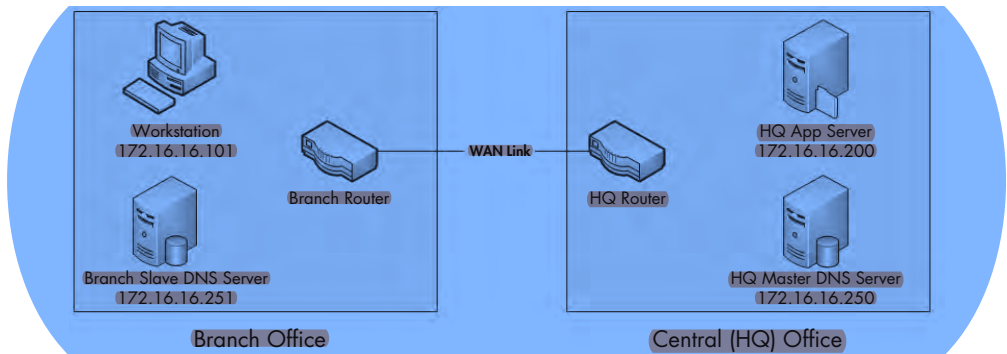


Figure 8-33: The relevant components for the stranded branch office issue

Tapping into the Wire

`stranded_
clientside.pcap`

Because the problem lies in communication between the main and branch offices, there are a couple of places we could collect data to start tracking down the problem. The problem could be with the clients inside the branch office, so we'll start by port mirroring one of those computers to check what it sees on the wire. Once we've collected that information, we can use it to point toward other collection locations that might help solve the problem. The initial capture file obtained from one of the clients is `stranded_clientside.pcap`.

Analysis

`stranded_
branchdns.pcap`

As shown in Figure 8-34, our first capture file begins when the user at the workstation address 172.16.16.101 attempts to access an application hosted on the headquarters app server, 172.16.16.200. This capture contains only two packets. It appears as though a DNS request is sent to 172.16.16.251 ❶ for the A record ❷ for appserver ❸ in the first packet. This is the DNS name for the server at 172.16.16.200 in the central office.

As you can see in Figure 8-35, the response to this packet is a server failure ❶, which indicates that something is preventing the DNS query from completing successfully. Notice that this packet does not answer the query ❷ since it is an error (server failure).

We now know that the communication problem is related to some DNS issue. Because the DNS queries at the branch office are resolved by the DNS server at 172.16.16.251, that's our next stop.

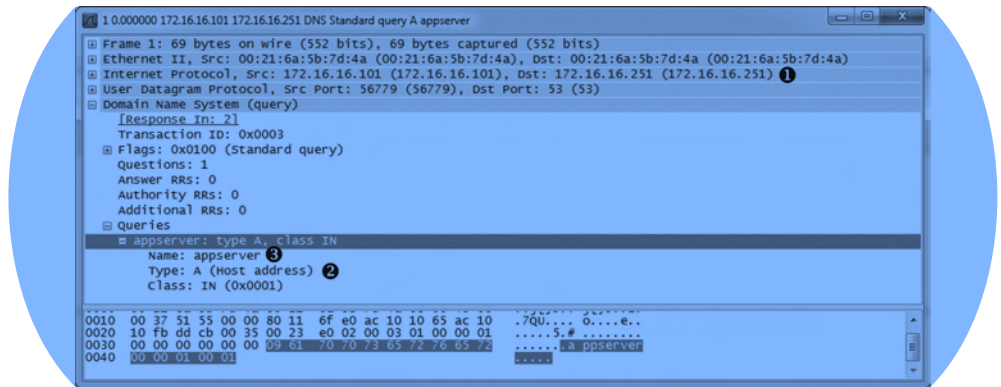


Figure 8-34: Communication begins with a DNS query for the appserver A record.

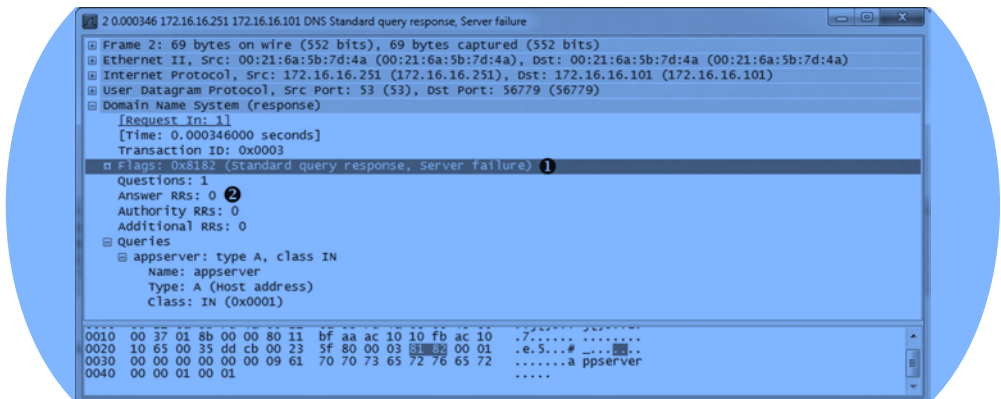


Figure 8-35: The query response indicates a problem upstream.

In order to capture the appropriate traffic from the branch DNS server, we'll leave our sniffer in place and simply change the port-mirroring assignment so that the server's traffic, rather than the workstation's traffic, is now mirrored to our sniffer. The result is the file *stranded_branchdns.pcap*.

As shown in Figure 8-36, this capture begins with the query and response we saw earlier, along with one additional packet. This additional packet looks a bit odd because it is attempting to communicate with the primary DNS server at the central office (172.16.16.250) ① on the standard DNS server port 53 ②, but it is not the UDP ③ we're used to seeing.

In order to figure out the purpose of this packet, recall our discussion of DNS in Chapter 7. DNS usually uses UDP, but it uses TCP when the response to a query exceeds a certain size. In that case, we'll see some initial UDP traffic that triggers the TCP traffic. TCP is also used for DNS during a zone transfer, when resource records are transferred between DNS servers, which is likely the case here.

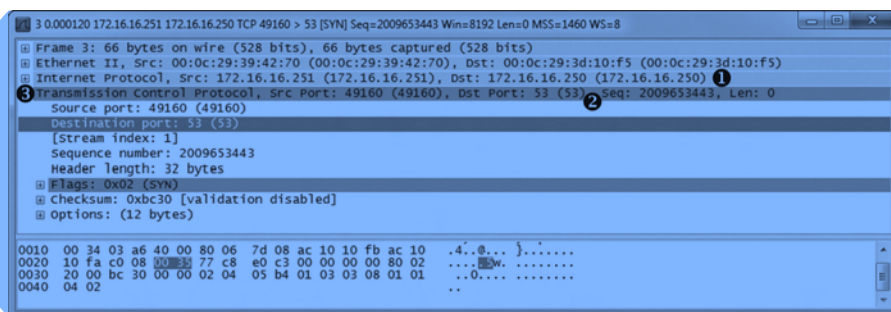


Figure 8-36: This SYN packet uses port 53 but is not UDP.

The DNS server at the branch office location is a slave to the DNS server at the central office, meaning that it relies on it in order to receive resource records. The application server that users in the branch office are trying to access is located inside the central office, which means that the central office DNS server is authoritative for that server. In order for the branch office server to be able to resolve a DNS request for the application server, the DNS resource record for that server must be transferred from the central office DNS server to the branch office DNS server. This is likely the source of the SYN packet in this capture file.

The lack of response to this SYN packet tells us that the DNS problem here is the result of a failed zone transfer between the branch and central office DNS servers. Now we can go one step further by figuring out why the zone transfer is failing. The possible culprits for the issue can be narrowed down to the routers between the offices or the central office DNS server itself. In order to figure this out, we can sniff the traffic of the central office DNS server to see if the SYN packet is making it to the server.

I have not included a capture file for the central office DNS server traffic because there was none. The SYN packet never reached the server. Upon dispatching technicians to review the configuration of the routers connecting the two offices, it was found that the central office router was configured to allow UDP traffic inbound only on port 53 and block TCP traffic inbound on port 53. This simple misconfiguration prevented zone transfers from occurring between servers, which prevented clients within the branch office from resolving queries for devices in the central office.

Lessons Learned

You can learn a lot about investigating network communications issues by watching crime dramas. When a crime occurs, the detectives begin by interviewing those most affected. Leads that result from that examination are pursued, and the process continues until a culprit is found.

In this scenario, we began by examining the victim (the workstation) and established leads by finding the DNS communication issue. Our leads led us to the branch DNS server, then to the central DNS server, and finally to the router, which was the source of the problem.

When performing analysis, try thinking of packets as clues. The clues don't always tell you who committed the crime, but they often take you to the culprit eventually.

Ticked-Off Developer

Some of the most frequent arguments in IT are between developers and system administrators. Developers always blame shoddy network setup and malfunctioning equipment for program malfunctions. System administrators tend to blame bad code for network errors and slow communication.

In this scenario, a programmer has developed an application for tracking the sales at multiple stores and reporting back to a central database. In an effort to save bandwidth during normal business hours, this is not a real-time application. Reporting data is accumulated throughout the day and is transmitted at night as a comma-separated value (CSV) file to be inserted into the central database.

This newly developed application is not functioning correctly. The files sent from the stores are being received by the server, but the data being inserted into the database is not correct. Sections are missing, data is in the wrong place, and some portions of the data are missing. Much to the dismay of the system administrator, the programmer blames the network for the issue. He is convinced that the files are becoming corrupted while in transit from the stores to the central data repository. Our goal is to prove him wrong.

Tapping into the Wire

`tickedoffdeveloper.pcap`

In order to collect the data we need, we can capture packets at one of the stores or at the central office. Because the issue is affecting all of the stores, it would seem that if the issue is network-related, it would occur at the central office—that is the only common thread among all stores.

The network switches support port mirroring, so we'll mirror the port the server is plugged into and sniff its traffic. The traffic capture will be isolated to a single instance of a store uploading its CSV file to the collection server. This result is the capture file `tickedoffdeveloper.pcap`.

Analysis

We know nothing about the application the programmer has developed, other than the basic flow of information on the network. The capture file appears to start with some FTP traffic, so we'll investigate that to see if it is indeed the mechanism that is transporting this file. This is a good place to examine the communication flow graph for a nice, clean summary of the communication that is occurring. To do so, select **Statistics ▶ Flow Graph**, and then click **OK**. Figure 8-37 shows the resulting graph.

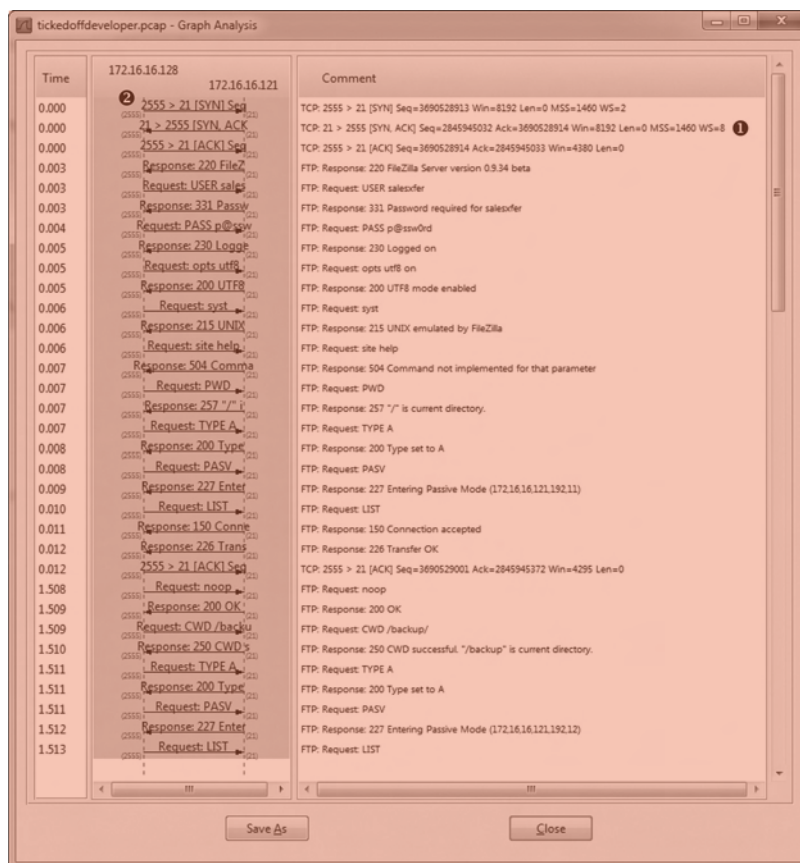


Figure 8-37: The flow graph gives a quick view of the FTP communication.

Based on this flow graph, we see that a basic FTP connection is set up between 172.16.16.128 and 172.16.16.121. Since 172.16.16.128 is initiating the connection, we can assume that it is the client, and that 172.16.16.121 is the server that compiles and processes the data. The flow graph confirms that this traffic is exclusively using the FTP protocol.

We know that some transfer of data should be happening here, so we can use our knowledge of FTP to locate the packet where this transfer begins. The FTP connection and data transfer are initiated by the client, so we should be looking for the FTP STOR command, which is used to upload data to an FTP server. The easiest way to find this is to build a filter.

Because this capture file is littered with FTP request commands, rather than sorting through the hundreds of protocols and options in the expression builder, we can build the filter we need directly from the Packet List pane. In order to do this, we first need to select a packet with an FTP request command present. We will choose packet 5, since it's near the top of our list. Then expand the FTP section in the Packet Details pane and expand the USER section. Right-click the Request Command: USER field and select Prepare a Filter. Finally, choose Selected.

This will prepare a filter for all packets that contain the FTP USER request command and put it in the filter dialog. Next, as shown in Figure 8-38, edit the filter by replacing the word USER with the word **STOR** ❶.

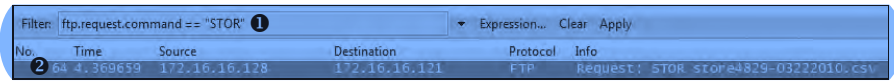


Figure 8-38: This filter helps identify where data transfer begins.

Now apply this filter by pressing ENTER, and you'll see that only one instance of the STOR command exists in the capture file, at packet 64 ❷.

Now that we know where data transfer begins, clear the filter by clicking the **Clear** button above the Packet List pane.

Examining the capture file beginning with packet 64, we see that this packet specifies the transfer of the file *store4829-03222010.csv* ❶, as shown in Figure 8-39.

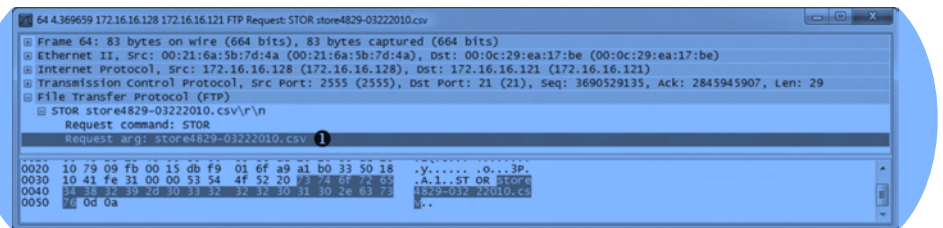


Figure 8-39: The CSV file is being transferred using FTP.

The packets following the STOR command use a different port, but are identified as part of an FTP-DATA transmission. We've verified that data is being transferred, but we have yet to prove the programmer wrong. In order to do that, we need to show that the contents of the file are sound after traversing the network by extracting the transferred file from the captured packets.

When a file is transferred across a network in an unencrypted format, it is broken down into segments and reassembled at its destination. In this scenario, we captured packets as they reached their destination but before they were reassembled. The data is all there, we simply need to reassemble it by extracting the file as a data stream. To perform the reassembly, select any of the packets in the FTP-DATA stream (such as packet 66) and click **Follow TCP Stream**. The results are displayed in the TCP stream, as shown in Figure 8-40.

The data appears because it is being transferred in plaintext over FTP, but we can't be sure that the file is intact based on the stream alone. In order to reassemble the data so that we can extract it in its original format, click the **Save As** button and specify the name of the file as displayed in packet 64, as shown in Figure 8-41. Then click **Save**.

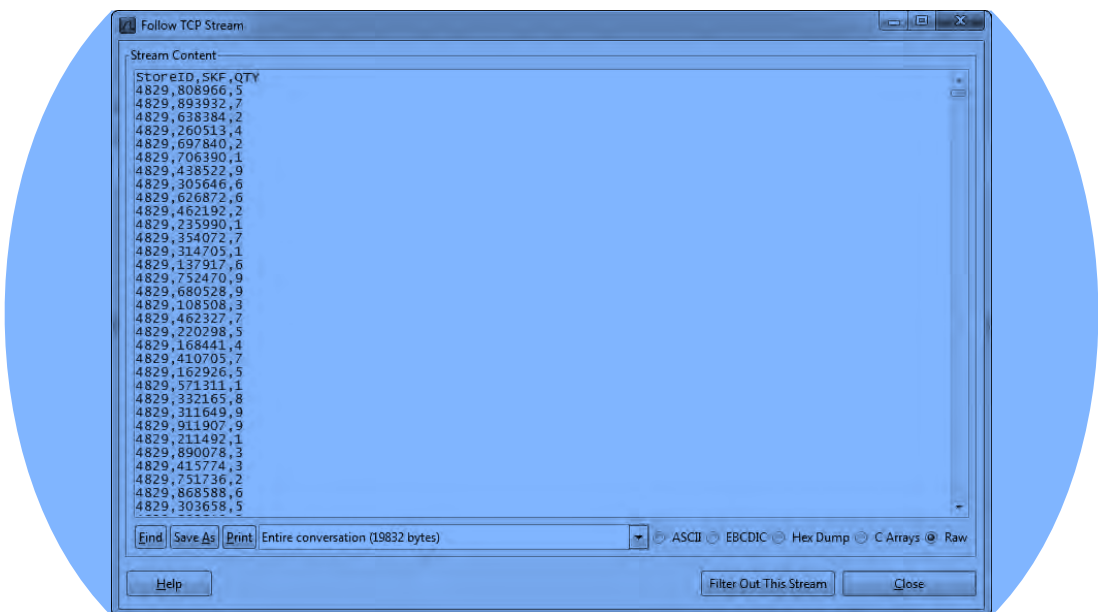


Figure 8-40: The TCP stream shows what appears to be the data being transferred.

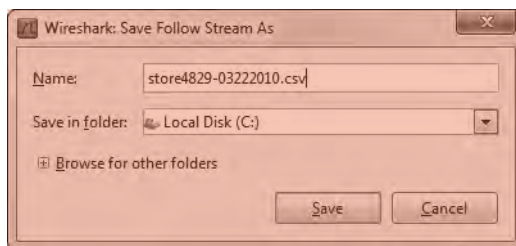


Figure 8-41: Saving the stream as the original filename

The result of this save operation should be a CSV file that is an exact byte-level copy of the file originally transferred from the store system. The file can be verified by comparing the MD5 hash of the original file with that of the extracted file. The MD5 hashes should be the same, as shown in Figure 8-42.

Once the files are compared, we can prove that the network is not to blame for the database corruption occurring within the application. The file transferred from the store to the collection server is intact when it reaches the server, so any corruption must be occurring when the file is processed by the application.

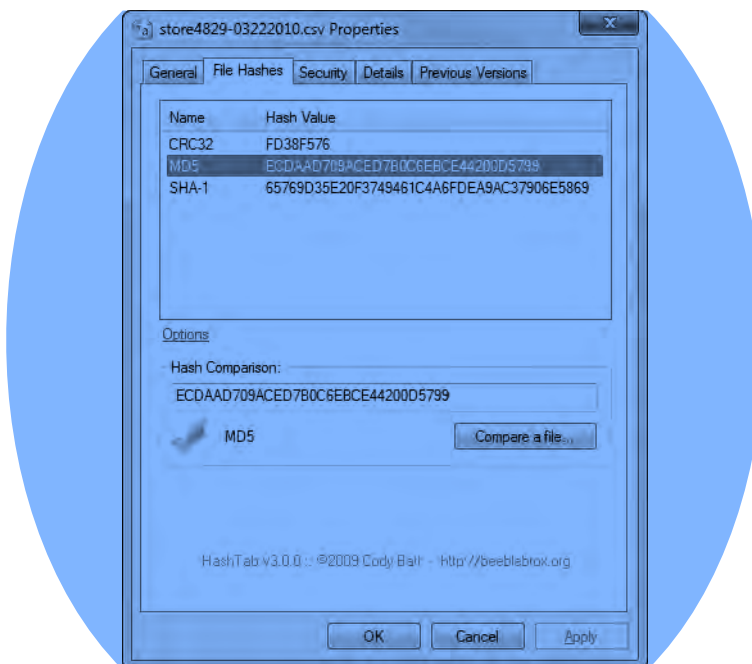


Figure 8-42: The MD5 hashes of the original file and the extracted file are equivalent.

Lessons Learned

One great thing about packet-level analysis is that you don't need to deal with the clutter of applications. Poorly coded applications greatly outnumber the good ones, but at the packet level, none of that matters. In this case, the programmer was concerned about all of the mysterious components his application was dependent upon, but at the end of the day, his complicated data transfer that took hundreds of lines of code is still no more than FTP, TCP, and IP. Using what we know about these basic protocols, we were able to ensure the communication process flowed correctly and even extract files to prove the soundness of the network. It's crucial to remember that no matter how complex the issue at hand, it's still just packets.

Final Thoughts

In this chapter, we've covered several basic scenarios where packet analysis allowed us to gain a better understanding of problematic communication. Using basic analysis of common protocols, we were able to track down and solve network problems in a timely manner. While you may not encounter exactly the same scenarios on your network, the analysis techniques presented here should prove useful to you as you analyze your own unique problems.

9

FIGHTING A SLOW NETWORK



As a network administrator, much of your time will be spent fixing computers and services that are running slower than they should be. But just because someone says that the network is running slowly does not mean that the network is to blame.

Before you begin to tackle a slow network problem, you first need to determine whether the network is in fact running slowly. You'll learn how to do that in this chapter.

We will begin by discussing the error-recovery and flow-control features of TCP. Then we will explore how to detect the source of slowness on a network. Finally, we will look at approaches for baselining networks and the devices and services that run on them. Once you have completed this chapter, you should be much better equipped to identify, diagnose, and troubleshoot slow networks.

NOTE Multiple techniques can be used to troubleshoot slow networks. I've chosen to focus this chapter primarily on TCP because most of the time it is all that you will have to work with. TCP allows you to perform passive retrospective analysis rather than generate additional traffic (as with ICMP).

TCP Error-Recovery Features

TCP's error-recovery features are our best tools for locating, diagnosing, and eventually repairing high latency on a network. In terms of computer networking, *latency* is a measure of delay between a packet's transmission and its receipt.

Latency can be measured as one-way (from a single source to a destination) or as round-trip (from a source to a destination and back to the original source). When communication between devices is fast, and the amount of time it takes for a packet to get from one point to another is low, the communication is said to have *low latency*. Conversely, when packets take a significant amount of time to travel between a source and destination, the communication is said to have *high latency*. High latency is the number one enemy of all network administrators who value their sanity (and their job).

In Chapter 6, we discussed how TCP uses sequence and acknowledgment numbers to ensure the reliable delivery of packets. In this chapter, we'll look at sequence and acknowledgment numbers again to see how TCP responds when high latency causes these numbers to be received out of sequence (or not received at all).

TCP Retransmissions

tcp_retransmissions
.pcap

The ability of a host to retransmit packets is one of TCP's most fundamental error-recovery features. It is designed to combat packet loss.

There are many possible causes for packet loss, including malfunctioning applications, routers under a heavy traffic load, or a temporary service outage. Things move fast at the packet level, and often the packet loss is temporary, so it's crucial for TCP to be able to detect and recover from packet loss.

The primary mechanism for determining whether the retransmission of a packet is necessary is called the *retransmission timer*. This timer is responsible for maintaining a value called the *retransmission timeout (RTO)*. Whenever a packet is transmitted using TCP, the retransmission timer starts. This timer stops when an ACK for that packet is received. The time between the packet transmission and receipt of the ACK packet is called the *round-trip time (RTT)*. Several of these times are averaged, and that average is used to determine the final RTO value.

Until an RTO value is actually determined, the transmitting operating system relies on its default configured RTT setting. This setting is issued for the *initial* communication between hosts and is adjusted based on the RTT from received packets in order to form the actual RTO.

Once the RTO value has been determined, the retransmission timer is used on every transmitted packet to determine whether packet loss has occurred. Figure 9-1 illustrates the TCP retransmission process.

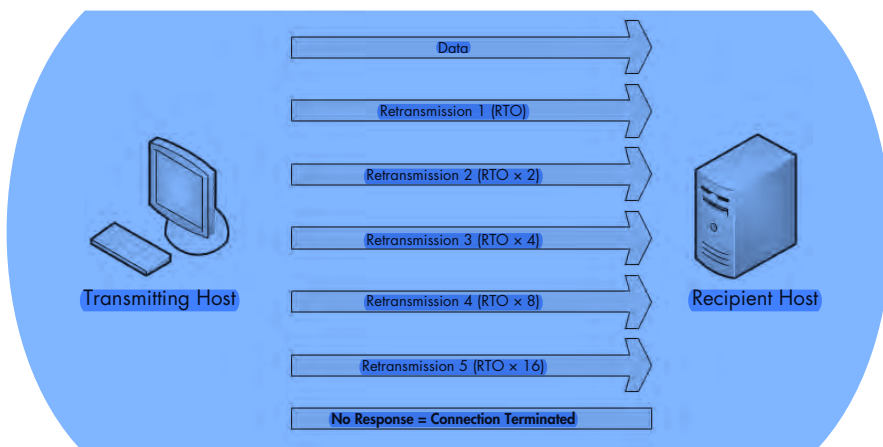


Figure 9-1: Conceptual view of the TCP retransmission process

When a packet is sent, but the recipient has not sent a TCP ACK packet, the transmitting host assumes that the original packet was lost and retransmits the original packet. When the retransmission is sent, the RTO value is doubled; if no ACK packet is received before that value is reached, another retransmission will occur. The RTO value will be doubled for the next retransmission should an ACK not be received. This process will continue, with the RTO value being doubled for each retransmission, until an ACK packet is received or until the sender reaches the maximum number of retransmission attempts it is configured to send.

The maximum number of retransmission attempts depends on the value configured in the transmitting operating system. By default, Windows hosts default to a maximum of five retransmission attempts. Most Linux hosts default to a maximum of 15 attempts. This option is configurable in either operating system category.

To see an example of TCP retransmission, open the file *tcp_retransmissions.pcap*, which contains six packets. The first packet is shown in Figure 9-2.

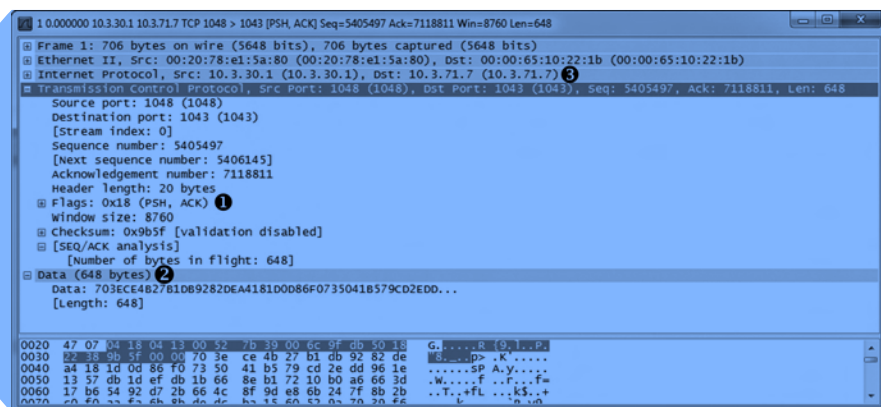


Figure 9-2: A simple TCP packet containing data

This packet is a TCP PSH/ACK packet ❶ containing 648 bytes of data ❷ that is sent from 10.3.30.1 to 10.3.71.7 ❸. This is a typical data packet.

Under normal circumstances, you would expect to see a TCP ACK packet in response fairly soon after the first packet is sent. In this case, however, the next packet is a retransmission. You can tell this by looking at the packet in the Packet List pane. The Info column clearly says [TCP Retransmission], and the packet will appear with red text on a black background. Figure 9-3 shows examples of retransmissions listed in the Packet List pane.

No.	Time	Source	Destination	Protocol	Info
1	0.000000	10.3.30.1	10.3.71.7	TCP	1048 > 1043 [PSH, ACK] Seq=5405497 Ack=7118811 win=8760 Len=648
2	0.206000	10.3.30.1	10.3.71.7	TCP	[TCP Retransmission] 1048 > 1043 [PSH, ACK] Seq=5405497 Ack=7118811 win=8760 Len=648
3	0.600000	10.3.30.1	10.3.71.7	TCP	[TCP Retransmission] 1048 > 1043 [PSH, ACK] Seq=5405497 Ack=7118811 win=8760 Len=648
4	1.200000	10.3.30.1	10.3.71.7	TCP	[TCP Retransmission] 1048 > 1043 [PSH, ACK] Seq=5405497 Ack=7118811 win=8760 Len=648
5	2.400000	10.3.30.1	10.3.71.7	TCP	[TCP Retransmission] 1048 > 1043 [PSH, ACK] Seq=5405497 Ack=7118811 win=8760 Len=648
6	4.805000	10.3.30.1	10.3.71.7	TCP	[TCP Retransmission] 1048 > 1043 [PSH, ACK] Seq=5405497 Ack=7118811 win=8760 Len=648

Figure 9-3: Retransmissions in the Packet List pane

You can also determine if a packet is a retransmission by examining it in the Packet Details and Packet Bytes panes, as shown in Figure 9-4.

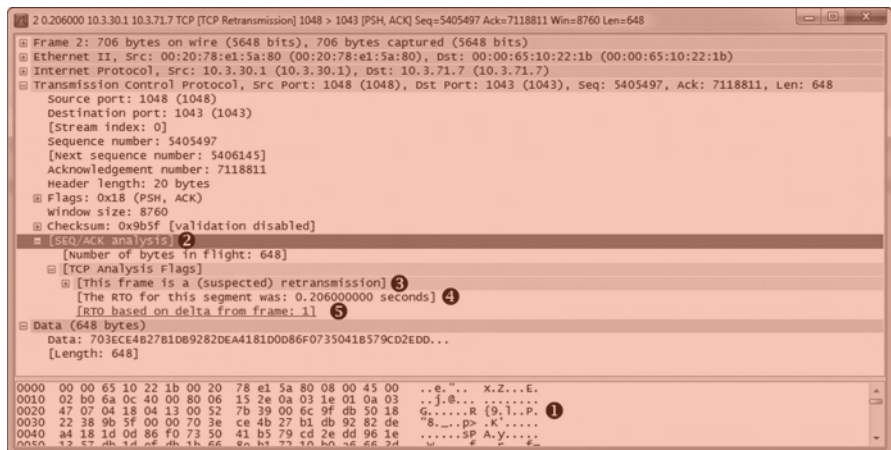


Figure 9-4: An individual retransmission packet

Note that this packet is the same as the original packet (other than the IP identification and Checksum fields). To verify this, compare the Packet Bytes pane of this retransmitted packet with the original one ❶.

In the Packet Details pane, notice that the retransmission packet has some additional information included under the SEQ/ACK Analysis heading ❷. This useful information is provided by Wireshark and is not actually contained in the packet itself. The SEQ/ACK analysis tells us that this is indeed a retransmission ❸, that the RTO value is 0.206 seconds ❹, and that the RTO is based on the delta time from packet 1 ❺.

Examination of the remaining packets should yield similar results, with the only differences between the packets found in the IP identification and Checksum fields, and the RTO value. To visualize the time lapse between each packet, look at the Time column in the Packet List pane, as shown in Figure 9-5. Here, you see exponential growth in time as the RTO value is doubled after each retransmission.

The TCP retransmission feature is used by the transmitting device to detect and recover from packet loss. Next, we'll examine *TCP duplicate acknowledgments*, a feature that the data recipient uses to detect and recover from packet loss.

No.	Time
1	0.000000
2	0.206000
3	0.600000
4	1.200000
5	2.400000
6	4.805000

Figure 9-5: The Time column shows the increase in RTO value.

TCP Duplicate Acknowledgments and Fast Retransmissions

`tcp_dupack.pcap`

A duplicate ACK is a TCP packet sent from a recipient when that recipient receives packets that are out of order. TCP uses the sequence and acknowledgment number fields within its header to reliably ensure that data is received and reassembled in the same order in which it was sent.

NOTE The proper term for a TCP packet is actually a TCP segment, but most people tend to refer to them as packets.

When a new TCP connection is established, one of the most important pieces of information exchanged during the handshake process is an initial sequence number (ISN). Once the ISN is set for each side of the connection, each subsequently transmitted packet increments the sequence number by the size of its data payload.

Consider a host that has an ISN of 5000 and sends a 500-byte packet to a recipient. Once this packet has been received, the recipient host will respond with a TCP ACK packet with an acknowledgment number of 5500, based on the following formula:

$$\text{Sequence Number In} + \text{Bytes of Data Received} = \text{Acknowledgment Number Out}$$

As a result of this calculation, the acknowledgment number returned to the transmitting host is actually the next sequence number that the recipient expects to receive. An example of this can be seen in Figure 9-6.



Figure 9-6: TCP sequence and acknowledgment numbers

The detection of packet loss by the data recipient is made possible through the sequence numbers. As the recipient tracks the sequence numbers it is receiving, it can determine when it receives sequence numbers that are out of order.

When the recipient receives an unexpected sequence number, it assumes that a packet has been lost in transit. In order to reassemble data properly, the recipient must have the missing packet, so it resends the ACK packet that

contains the lost packet's expected sequence number in order to elicit a retransmission of that packet from the transmitting host.

When the transmitting host receives three duplicate ACKs from the recipient, it assumes that the packet was indeed lost in transit and immediately sends a *fast retransmission*. Once a fast retransmission is triggered, all other packets being transmitted are queued until the fast retransmission packet is sent. This process is depicted in Figure 9-7.

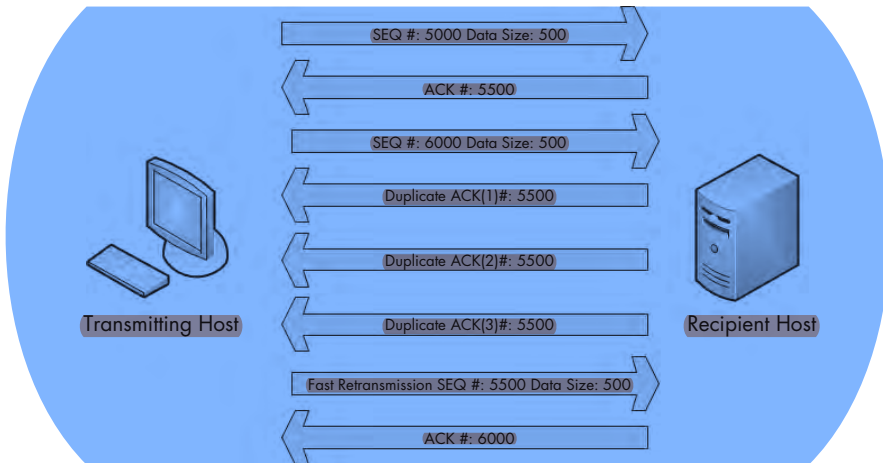


Figure 9-7: Duplicate ACKs from the recipient result in a fast retransmission.

You'll find an example of duplicate ACKs and fast retransmissions in the file *tcp_dupack.pcap*. The first packet in this capture is shown in Figure 9-8.

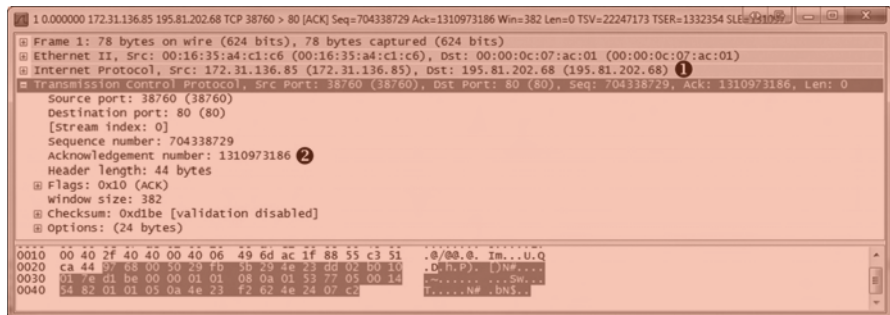


Figure 9-8: The ACK showing the next expected sequence number

This packet, a TCP ACK sent from the data recipient (172.31.136.85) to the transmitter (195.81.202.68) ❶, has an acknowledgment of the data sent in the previous packet that is not included in this capture file.

NOTE By default, Wireshark uses relative sequence numbers to make the analysis of these numbers easier, but the examples and screenshots in the next few sections do not use this feature. To turn off this feature, select **Edit ► Preferences**. In the Preferences window, select **Protocols** and then the **TCP** section. Then uncheck the box next to **Relative sequence numbers and window scaling**.

The acknowledgment number in this packet is 1310973186 ❷, which should be the sequence number of the next packet received, as shown in Figure 9-9.

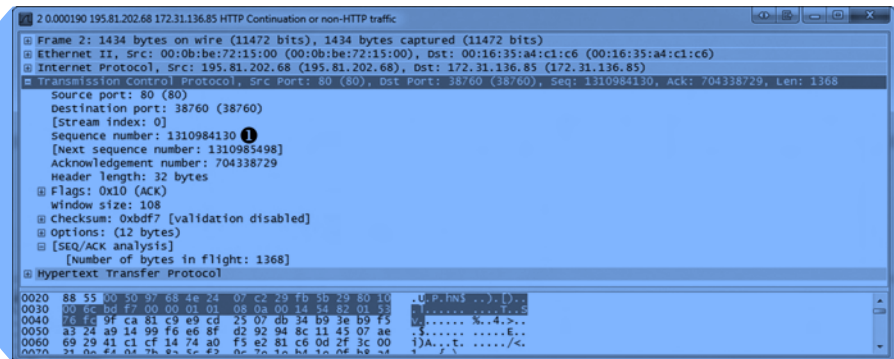


Figure 9-9: The sequence number of this packet is not what is expected.

Unfortunately for us and our recipient, the sequence number of the next packet is 1310984130 ❶, which is not what we are expecting. This indicates that the expected packet was somehow lost in transit. The recipient host notices that this packet is out of sequence and sends a duplicate ACK in the third packet of this capture, as shown in Figure 9-10.

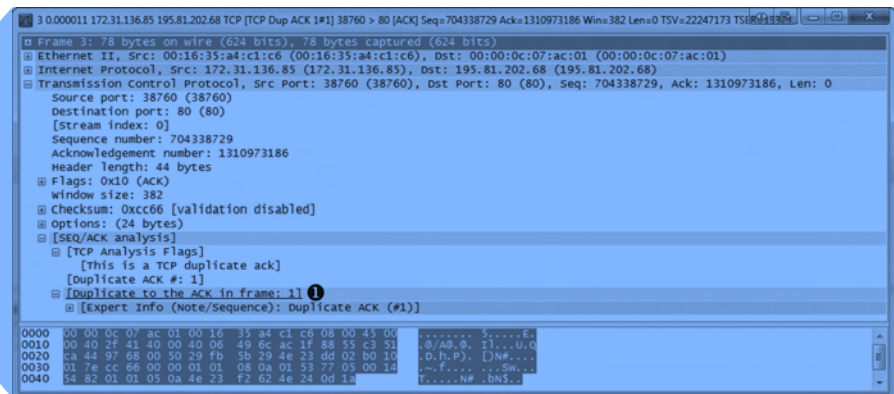


Figure 9-10: The first duplicate ACK packet

You can determine that this is a duplicate ACK packet by examining either of the following:

- The Info column in the Packet Details pane. The packet should appear as red text on a black background.
- The Packet Details pane under the SEQ/ACK Analysis heading. If you expand this heading, you will find that the packet is listed as a duplicate ACK of packet 1.

The next several packets continue this process, as shown in Figure 9-11.

No.	Time	Source	Destination	Protocol	Info
1	0.000000	172.31.136.85	195.81.202.68	TCP	38760 → 80 [ACK] Seq=704338729 Ack=1310973186 win=382 Len=0 TSV=22247173 TSER=
2	0.000190	195.81.202.68	172.31.136.85	HTTP	Continuation or non-HTTP traffic
3	0.000001	172.31.136.85	195.81.202.68	TCP	[TCP Dup ACK #1] 38760 → 80 [ACK] Seq=704338729 Ack=1310973186 win=382 Len=0
4	0.000093	195.81.202.68	172.31.136.85	HTTP	Continuation or non-HTTP traffic
5	0.000010	172.31.136.85	195.81.202.68	TCP	[TCP Dup ACK #2] 38760 → 80 [ACK] Seq=704338729 Ack=1310973186 win=382 Len=0
6	0.000121	195.81.202.68	172.31.136.85	HTTP	Continuation or non-HTTP traffic
7	0.000010	172.31.136.85	195.81.202.68	TCP	[TCP Dup ACK #3] 38760 → 80 [ACK] Seq=704338729 Ack=1310973186 win=382 Len=0

Figure 9-11: Additional duplicate ACKs are generated due to out-of-order packets.

The fourth packet in the capture file is another chunk of data sent from the transmitting host with the wrong sequence number ①. As a result, the recipient then sends its second duplicate ACK ②. One more packet with the wrong sequence number is received by the recipient ③. That forces the transmission of the third and final duplicate ACK ④.

As soon as the transmitting host receives the third duplicate ACK from the recipient, it is forced to halt all packet transmission and resend the lost packet. Figure 9-12 shows the fast retransmission of the lost packet.

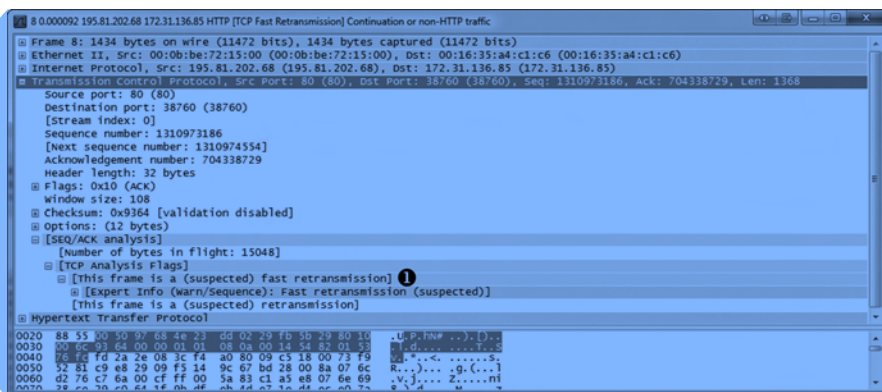


Figure 9-12: The duplicate ACKs cause this fast retransmission of the lost packet.

The retransmission packet is once again noticeable through the Info column in the Packet List pane. As with previous examples, the packet is clearly labeled with red text on a black background. The SEQ/ACK Analysis section of this packet tells us that this is suspected to be a fast retransmission ①. (Once again, the information that labels this packet as a fast retransmission is not a value set in the packet itself, but rather a feature of Wireshark.) The final packet in the capture is an ACK packet acknowledging receipt of the fast retransmission.

NOTE One feature to consider that may affect the flow of data in TCP communications where packet loss is present is the Selective Acknowledgement feature. In the packet capture above, Selective ACK was negotiated as an enabled feature during the initial three-way handshake process. As a result, whenever a packet is lost and a duplicate ACK received, only the lost packet has to be retransmitted, even though other packets were received successfully after the lost packet. Had Selective ACK not been enabled, every packet occurring after the lost packet would have had to be retransmitted as well. Selective ACK makes data loss recovery much more efficient. Because most modern TCP/IP stack implementations support Selective ACK, you should usually find that this feature is implemented.

TCP Flow Control

Retransmissions and duplicate ACKs are reactive TCP functions designed to recover from packet loss. TCP would be a poor protocol if it didn't include some form of proactive method for preventing packet loss, but luckily it does.

TCP implements a *sliding-window mechanism* to detect when packet loss may occur and adjust the rate of data transmission to prevent this. The sliding-window mechanism leverages the data recipient's *receive window* to control the flow of data.

The receive window is a value specified by the data recipient and stored in the TCP header (in bytes) that tells the transmitting device how much data it is willing to store in its *TCP buffer space*. This buffer space is where data is stored temporarily until it can be passed up the stack to the application layer protocol waiting to process it. As a result, the transmitting host can send only the amount of data specified in the Window Size field at one time. In order for the transmitter to send more data, the recipient must send an acknowledgment that the previous data was received. It also must clear TCP buffer space by processing the data that is occupying that position. Figure 9-13 illustrates how the receive window works.

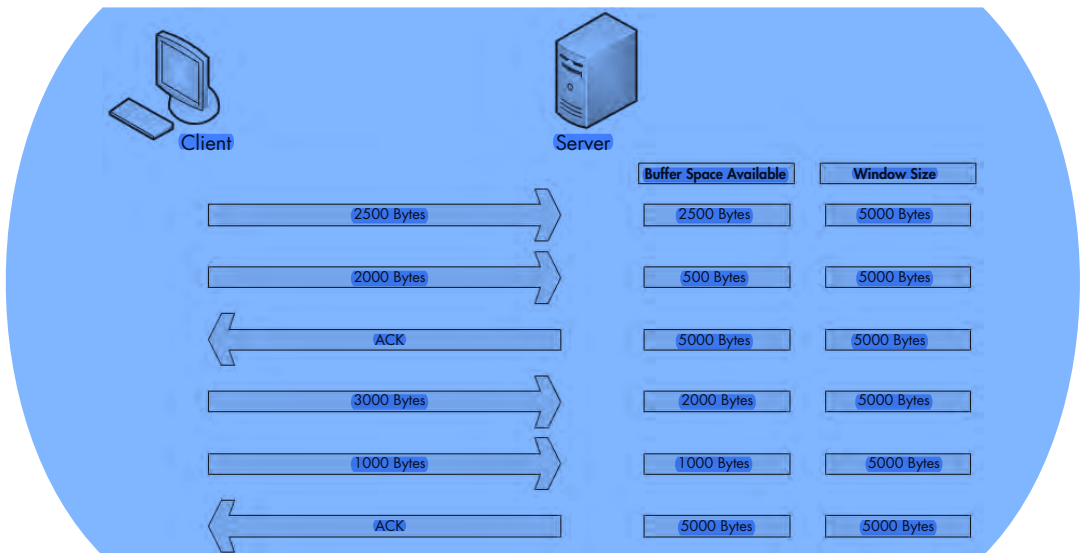


Figure 9-13: The receive window keeps the data recipient from getting overwhelmed.

In Figure 9-13, the client is sending data to a server that has communicated a receive window size of 5,000 bytes. The client sends 2,500 bytes, reducing the server's buffer space to 2,500 bytes, and then sends another 2,000 bytes, further reducing the buffer to 500 bytes. The server then sends an acknowledgment of this data. It processes the data in its buffer and then has an empty buffer available. This process repeats, with the client sending 3,000 bytes and another 1,000 bytes, reducing the server's buffer to 1,000 bytes. The client once more acknowledges this data and processes the contents of its buffer.

Adjusting the Window Size

This process of adjusting the window size is fairly clear-cut, but it isn't always perfect. Whenever data is received by the TCP stack, an acknowledgment is generated and sent in reply, but the data placed in the recipient's buffer is not always processed immediately.

When a busy server is processing packets from multiple clients, it's quite possible that the server could be slow in clearing its buffer and not be able to make room for new data to be received. With no means of flow control, this could lead to packets being lost and corruption of data. Fortunately, when a server becomes too busy to process data at the rate its receive window is advertising, it can adjust the size of the receive window. It does this by decreasing the window size value in the TCP header of the ACK packet it is sending back to the hosts that are sending it data. Figure 9-14 shows an example of this.

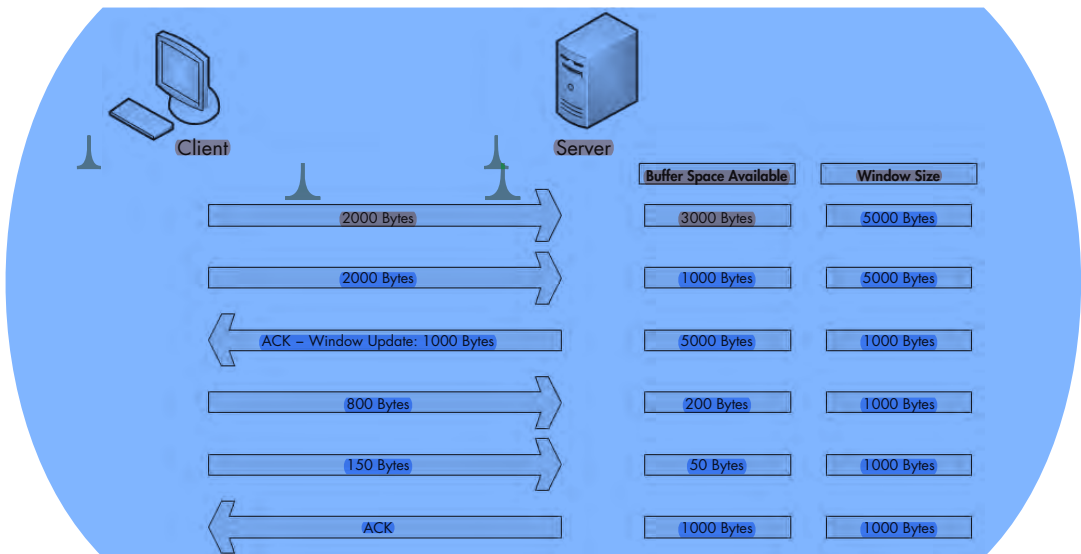


Figure 9-14: The window size can be adjusted when the server becomes busy.

In Figure 9-14, the server starts with an advertised window size of 5,000 bytes. The client sends 2,000 bytes, followed by another 2,000 bytes, leaving only 1,000 bytes of buffer space available. The server realizes that its buffer is filling up quickly. It knows that if data transfer keeps up at this rate, packets will soon be lost. To rectify this, the server sends an acknowledgment to the client with an updated window size of 1,000 bytes. As a result, less data is sent by the client, and the server can process its buffer contents at an acceptable rate that allows data to flow in a constant manner.

The resizing process works both ways. When the server can process data at a faster rate, it can send an ACK packet with a larger window size.

Halting Data Flow with a Zero Window Notification

In some cases, a server can no longer process data sent from a client. This might be due to a lack of memory, lack of processing capability, or another problem. This could result in packets being dropped and the communication process halting, but the receive window can help minimize the negative impact.

When this situation arises, a server can send a packet that contains a window size of zero. When the client receives this packet, it will halt any data transmission but will keep the connection to the server open with the transmission of keep-alive packets. Keep-alive packets are sent by the client at regular intervals to check the status of the server's receive window. Once the server can begin processing data again, it will respond with a nonzero window size, and communication will resume. Figure 9-15 illustrates an example of zero window notification.



Figure 9-15: Data transfer stops when the window size is set to 0 bytes.

In Figure 9-15, the server begins receiving data with a 5,000-byte window size. After receiving 4,000 bytes of data from the client, the server begins experiencing a very heavy processor load, and can no longer process any data from the client. The server then sends a packet with the Window Size field set to 0. The client halts transmission of data and sends a keep-alive packet. After the keep-alive packet, the server responds with a packet notifying the client that it can now receive data, and that its window size is 1,000 bytes. The client resumes sending data.

The TCP Sliding Window in Practice

`tcp_zerowindow-
recovery.pcap`
`tcp_zerowindow-
dead.pcap`

Having covered the theory behind the TCP sliding window, we will now examine it in the capture file `tcp_zerowindowrecovery.pcap`.

In this file, we begin with several TCP ACK packets traveling from 192.168.0.20 to 192.168.0.30. The main value of interest to us is the Window

Size field, which can be seen in both the Info column of the Packet List pane and in the TCP header in the Packet Details pane. You can see immediately that this field's value decreases over the course of the first three packets, as shown in Figure 9-16.

No.	Time	Source	Destination	Protocol	Info
1	0.000000	192.168.0.20	192.168.0.30	TCP	2235 > 1720 [ACK] Seq=1422793785 Ack=2710996659 win=8760 Len=0
2	0.000237	192.168.0.20	192.168.0.30	TCP	2235 > 1720 [ACK] Seq=1422793785 Ack=2710999579 win=5840 Len=0
3	0.000193	192.168.0.20	192.168.0.30	TCP	2235 > 1720 [ACK] Seq=1422793785 Ack=2711002499 win=2920 Len=0

Figure 9-16: The window size of these packets is decremting.

This value goes from 8,760 bytes in the first packet to 5,840 bytes in the second packet and then 2,920 bytes in the third packet ❶. This lowering of the window size value is a classic indicator of increased latency from the host. Notice in the Time column that this happens very quickly ❷. When the window size is lowered this fast, it's common for the window size to drop to zero, which is exactly what happens in the fourth packet, as shown in Figure 9-17.

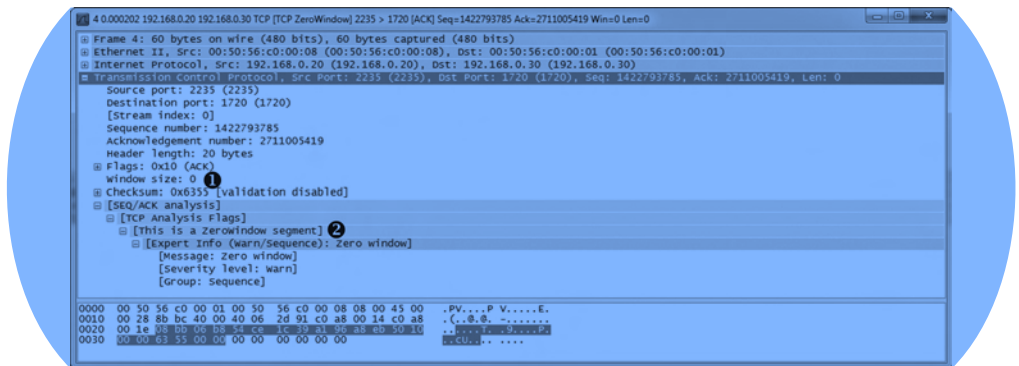


Figure 9-17: This zero window packet says that the host cannot accept any more data.

The fourth packet is also being sent from 192.168.0.20 to 192.168.0.30, but its purpose is to tell 192.168.0.30 that it can no longer receive any data. The 0 value is seen in the TCP header ❶, and Wireshark also tells us that this is a zero window packet in the Info column of the Packet List pane and under the SEQ/ACK Analysis section of the TCP header ❷.

Once this zero window packet is sent, the device at 192.168.0.30 will not send any more data until it receives a window update from 192.168.0.20 notifying it that the window size has increased. Luckily for us, the issue causing the zero window condition in this capture file was only temporary. So, a window update is sent in the next packet, as shown in Figure 9-18.

In this case, the window size is increased to a very healthy 64,240 bytes ❶. Wireshark once again lets us know that this is a window update under the SEQ/ACK Analysis heading.

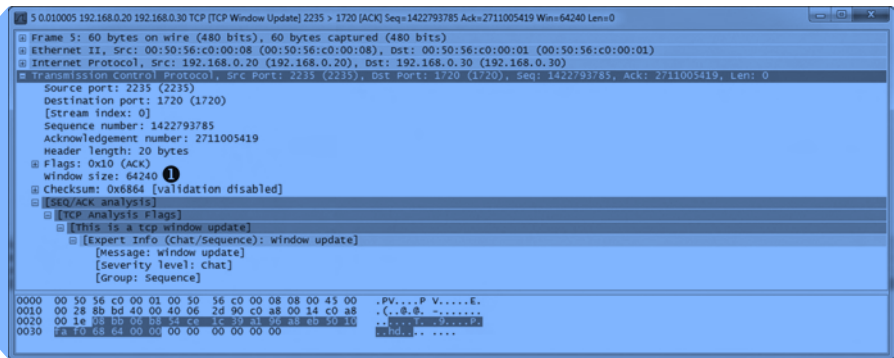


Figure 9-18: A TCP window update packet lets the other host know it can begin transmitting again.

Once the update packet is received, the host at 192.168.0.30 can begin sending data again, as it does in packets 6 and 7. This process takes place very quickly. Had it lasted only slightly longer, it could have caused a potential hiccup on the network, resulting in a slower or failed data transfer.

As one last look at the sliding window, examine the file *tcp_zerowindowdead.pcap*. The first packet in this capture is normal HTTP traffic being sent from 195.81.202.68 to 172.31.136.85. The packet is immediately followed with a zero window packet sent back from 172.31.136.85, as shown in Figure 9-19.



Figure 9-19: Zero window packet halting data transfer

This looks very similar to the zero window packet shown in Figure 9-17, but the result is much different. Rather than the 172.31.136.85 host sending a window update and communication resuming, we see a keep-alive packet, as shown in Figure 9-20.

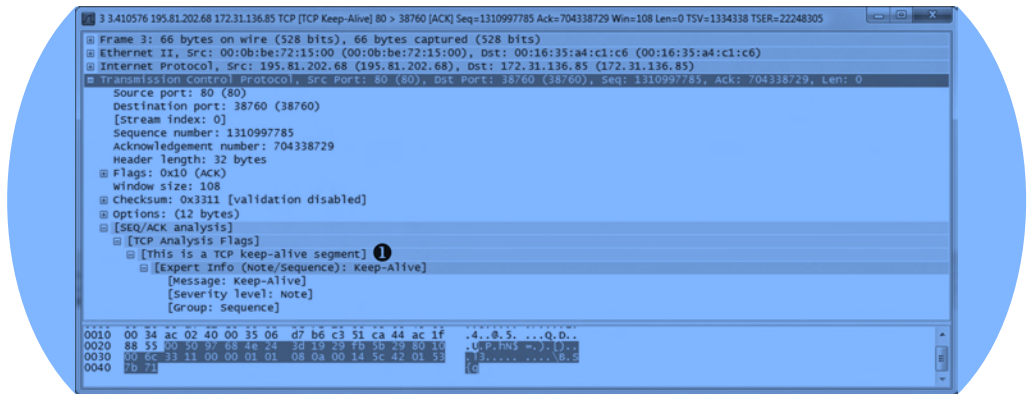


Figure 9-20: This keep-alive packet ensures the zero window host is still alive.

This packet is marked as a keep-alive by Wireshark under the SEQ/ACK Analysis section of the TCP header in the Packet Details pane ❶. The Time column tells us that this packet occurred 3.4 seconds after the last received packet. This process continues several more times, with one host sending a zero window packet and the other sending a keep-alive packet, as shown in Figure 9-21.

No.	Time ❶	Source	Destination	Protocol	Info
3	0.000000	195.81.202.68	172.31.136.85	TCP	TCP zero window 38760 → 80 ACK seq=1310997785 Ack=704338729 win=0 Len=0 TSval=1334338 TSer=22248305
4	3.410516	172.31.136.85	195.81.202.68	TCP	TCP keep-alive 80 → 38760 ACK seq=1310997785 Ack=704338729 win=108 Len=0 TSval=22248305 TSer=1334338
5	6.821032	195.81.202.68	172.31.136.85	TCP	TCP zero window 38760 → 80 ACK seq=704338729 Ack=1310997785 win=0 Len=0 TSval=22248305 TSer=1334338
6	10.231548	172.31.136.85	195.81.202.68	TCP	TCP keep-alive 80 → 38760 ACK seq=1310997785 Ack=704338729 win=108 Len=0 TSval=1334338 TSer=22248305
7	13.642064	195.81.202.68	172.31.136.85	TCP	TCP zero window 38760 → 80 ACK seq=704338729 Ack=1310997785 win=0 Len=0 TSval=22248305 TSer=1334338
8	17.052580	172.31.136.85	195.81.202.68	TCP	TCP keep-alive 80 → 38760 ACK seq=1310997785 Ack=704338729 win=108 Len=0 TSval=1334338 TSer=22248305
9	20.463096	195.81.202.68	172.31.136.85	TCP	TCP zero window 38760 → 80 ACK seq=704338729 Ack=1310997785 win=0 Len=0 TSval=22248305 TSer=1334338
10	23.873612	172.31.136.85	195.81.202.68	TCP	TCP keep-alive 80 → 38760 ACK seq=1310997785 Ack=704338729 win=108 Len=0 TSval=1334338 TSer=22248305

Figure 9-21: The zero window and keep-alive packets keep occurring over time.

These keep-alive packets occur at intervals of 3.4, 6.8, and 13.5 seconds ❶. This process can go on for quite a long time, depending on the operating systems of the communicating devices. In this case, as you can see by adding up the values in the Time column, the connection is halted for nearly 25 seconds. Imagine attempting to authenticate with a domain controller or download a file from the Internet while experiencing a 25-second delay—unacceptable!

Learning from TCP Error-Control and Flow-Control Packets

Let's put retransmission, duplicate ACKs, and the sliding-window mechanism into some context. Here are a few notes to keep in mind when troubleshooting latency issues:

Retransmission packets

Retransmissions occur because the client has detected that the server is not receiving the data it's sending. Therefore, depending on the side of the communication you are analyzing, you may never see retransmissions. If you are capturing data from the server, and it is truly not receiving the packets being sent and retransmitted from the client, you may be left in the dark because you won't see the retransmission packets. If you suspect

that you are the victim of packet loss on the server side, consider attempting to capture traffic from the client (if possible) so that you can actually see if retransmission packets are present.

Duplicate ACK packets

I tend to think of a duplicate ACK as the pseudo-opposite of a retransmission, because it is sent when the server detects that a packet from the client it is communicating with was lost in transit. In most cases, you can see duplicate ACKs when capturing traffic on both sides of the communication. Remember that duplicate ACKs are triggered when packets are received out of sequence. For example, if the server received just the first and third of three packets sent, that would cause a duplicate ACK to be sent to elicit a fast retransmission of the second packet from the client. Since you have received the first and third packets, it's likely that whatever condition caused the second packet to be dropped was only temporary, so the duplicate ACK would be sent and received successfully in most cases. Of course, this scenario isn't true all the time, so when you suspect packet loss on the server side and don't see any duplicate ACKs, consider capturing packets from the client side of the communication.

Zero window and keep-alive packets

The sliding window directly relates to the server's inability to receive and process data. Any decrease in the window size or zero window states are a direct result of some issue with the server, so if you see either occurring on the wire, you should focus your investigation there. You should typically always see window update packets on both sides of network communications.

Locating the Source of High Latency

In some cases, packet loss may not be the cause of latency. You may find that even though communications between two hosts are slow, that slowness doesn't show the common symptoms of TCP retransmissions or duplicate ACKs. In cases such as these, you need another technique to locate the source of the high latency.

One of the most effective ways to find the source of high latency is to examine the initial connection handshake and the first couple of packets that follow it. For example, consider a simple connection between a client and a web server as the client attempts to browse a site hosted on the web server. The portion of this communication sequence we are concerned with is the first six packets, consisting of the TCP handshake, the initial HTTP GET request, the acknowledgment to that GET request, and the first data packet sent from the server to the client.

NOTE *In order to follow along with this section, ensure that you have the proper time display format set in Wireshark by selecting **View ▶ Time Display Format ▶ Seconds Since Previous Displayed Packet**.*

Normal Communications

latency1.pcap

We'll discuss network baselines in detail a little later in the chapter. For now, just know that you need a baseline of normal communications to compare with the conditions of high latency. For these examples, we will use the file *latency1.pcap*. We have already covered the details of the TCP handshake and HTTP communication, so we won't review those topics again. In fact, we won't look at the Packet Details pane at all. All we are really concerned about is the Time column, as shown in Figure 9-22.

No.	Time	Source	Destination	Protocol	Info
1	0.000000	172.16.16.128	74.125.95.104	TCP	1606 > 80 [SYN] Seq=2082691767 win=8192 Len=0 MSS=1460 WS=2
2	0.030107	74.125.95.104	172.16.16.128	TCP	80 > 1606 [SYN, ACK] Seq=2775577373 Ack=2082691768 win=5720 Len=0 MSS=1406 WS=6
3	0.000075	172.16.16.128	74.125.95.104	TCP	1606 > 80 [ACK] Seq=2082691768 Ack=2775577374 win=4218 Len=0
4	0.000066	172.16.16.128	74.125.95.104	HTTP	GET / HTTP/1.1
5	0.048778	74.125.95.104	172.16.16.128	TCP	80 > 1606 [ACK] Seq=2775577374 Ack=2082692395 win=109 Len=0
6	0.022176	74.125.95.104	172.16.16.128	TCP	[TCP segment of a reassembled PDU]

Figure 9-22: This traffic happens very quickly and can be considered normal.

This communication sequence is quite quick. The entire process takes less than 0.1 seconds.

The next few capture files we'll examine will consist of this same traffic pattern with a few differences in the timing of the packets.

Slow Communications—Wire Latency

latency2.pcap

Now let's turn to the capture file *latency2.pcap*. Notice that all of the packets are the same except for the time values in two of them, as shown in Figure 9-23.

No.	Time	Source	Destination	Protocol	Info
1	0.000000	172.16.16.128	74.125.95.104	TCP	1606 > 80 [SYN] Seq=2082691767 win=8192 Len=0 MSS=1460 WS=2
2	0.878530	74.125.95.104	172.16.16.128	TCP	80 > 1606 [SYN, ACK] Seq=2775577373 Ack=2082691768 win=5720 Len=0 MSS=1406 WS=6
3	0.016604	172.16.16.128	74.125.95.104	TCP	1606 > 80 [ACK] Seq=2082691768 Ack=2775577374 win=4218 Len=0
4	0.000335	172.16.16.128	74.125.95.104	HTTP	GET / HTTP/1.1
5	1.159228	74.125.95.104	172.16.16.128	TCP	80 > 1606 [ACK] Seq=2775577374 Ack=2082692395 win=109 Len=0
6	0.015866	74.125.95.104	172.16.16.128	TCP	[TCP segment of a reassembled PDU]

Figure 9-23: Packets 2 and 5 depict high latency

As we begin stepping through these six packets, we encounter our first sign of latency immediately. The initial SYN packet is sent by the client (172.16.16.128) to begin the TCP handshake, and a delay of 0.87 seconds is seen before the return SYN/ACK is received from the server (74.125.95.104). This is our first indicator that we are experiencing wire latency, which is caused by a device between the client and server.

We can make the determination that this is wire latency because of the nature of the types of packets being transmitted. When the server receives a SYN packet, a very minimal amount of processing is required to send a reply, because the workload doesn't involve any processing above the transport layer. Even when a server is experiencing a very heavy traffic load, it will typically respond to a SYN packet with a SYN/ACK rather quickly. This eliminates the server as the potential cause of the high latency.

The client is also eliminated because, at this point, it is not doing any processing beyond the actual receipt of the SYN/ACK packet.

Elimination of both the client and server points us to potential sources of slow communication within the first two packets of this capture.

Continuing on, we see that the transmission of the ACK packet that completes the three-way handshake occurs quickly, as does the HTTP GET request sent by the client. All of the processing that generates these two packets occurs locally on the client following receipt of the SYN/ACK, so these two packets are expected to be transmitted quickly, as long as the client is not under a heavy processing load.

At packet 5, we see another packet with an incredibly high time value. It appears that after our initial HTTP GET request was sent, the ACK packet returned from the server took 1.15 seconds to be received. Upon receipt of the HTTP GET request, the server first sent a TCP ACK before it began sending data, which once again requires very little processing by the server. This is another sign of wire latency.

Whenever you experience true wire latency, you will almost always see it exhibited in both the SYN/ACK during the initial handshake and in other ACK packets throughout the communication. Although this information doesn't tell you the exact source of the high latency on this network, it does tell you that neither client nor server is the source, so you know that the latency is due to some device in between. At this point, you could begin examining the various firewalls, routers, and proxies between the affected host to locate the culprit.

Slow Communications—Client Latency

latency3.pcap

The next latency scenario we'll examine is contained in the file *latency3.pcap*, as shown in Figure 9-24.

No.	Time	Source	Destination	Protocol	Info
1	0.000000	172.16.16.128	74.125.95.104	TCP	1606 > 80 [SYN] Seq=2082691767 win=8192 Len=0 MSS=1460 WS=2
2	0.023790	74.125.95.104	172.16.16.128	TCP	80 > 1606 [SYN, ACK] Seq=2775577373 Ack=2082691768 win=5720 Len=0 MSS=1406 WS=6
3	0.014894	172.16.16.128	74.125.95.104	TCP	1606 > 80 [ACK] Seq=2082691768 Ack=2775577374 win=4218 Len=0
4	1.345023	172.16.16.128	74.125.95.104	HTTP	GET / HTTP/1.1
5	0.046121	74.125.95.104	172.16.16.128	TCP	80 > 1606 [ACK] Seq=2775577374 Ack=2082692395 win=109 Len=0
6	0.016182	74.125.95.104	172.16.16.128	TCP	[TCP segment of a reassembled PDU]

Figure 9-24: The slow packet in this capture is the initial HTTP GET

This capture begins normally, with the TCP handshake occurring very quickly and without any signs of latency. Everything appears to be fine until packet 4, an HTTP GET request after the handshake has completed. This packet shows a 1.34-second delay from the previously received packet.

We need to examine what is occurring between packets 3 and 4 in order to determine the source of this delay. Packet 3 is the final ACK in the TCP handshake sent from the client to the server, and packet 4 is the GET request sent from the client to the server. The common thread here is that these are both packets sent by the client and are independent of the server. The GET request should occur quickly after the ACK is sent, since all of these actions are centered on the client.

Unfortunately for the end user, the transition from ACK to GET doesn't happen quickly. The creation and transmission of the GET packet does require processing up to the application layer, and the delay in this processing indicates that the client was unable to perform the action in a timely manner. This means that the client is ultimately responsible for the high latency in the communication.

Slow Communications—Server Latency

latency4.pcap

The last latency scenario we'll examine uses the file *latency4.pcap*, as shown in Figure 9-25. This is an example of server latency.

No.	Time	Source	Destination	Protocol	Info
1	0.000000	172.16.16.128	74.125.95.104	TCP	1606 > 80 [SYN] Seq=2082691767 Win=8192 Len=0 MSS=1460 WS=2
2	0.018583	74.125.95.104	172.16.16.128	TCP	80 > 1606 [SYN, ACK] Seq=2775577373 Ack=2082691768 Win=5720 Len=0 MSS=1406 WS=6
3	0.016197	172.16.16.128	74.125.95.104	TCP	1606 > 80 [ACK] Seq=2082691768 Ack=2775577374 Win=4218 Len=0
4	0.000172	172.16.16.128	74.125.95.104	HTTP	GET / HTTP/1.1
5	0.047936	74.125.95.104	172.16.16.128	TCP	80 > 1606 [ACK] Seq=2775577374 Ack=2082692395 Win=109 Len=0
6	0.982983	74.125.95.104	172.16.16.128	TCP	[TCP segment of a reassembled PDU]

Figure 9-25: High latency isn't exhibited until the last packet of this capture.

In this capture, the TCP handshake process between these two hosts completes flawlessly and quickly, so things begin well. The next couple of packets bring more good news, as the initial GET request and response ACK packets are delivered quickly as well. It is not until the last packet in this file that we see a packet exhibiting signs of high latency.

This sixth packet is the first HTTP data packet sent from the server in response to the GET request sent by the client, but with a rather slow arrival time of 0.98 seconds after the server sends its TCP ACK for the GET request. The transition between packets 5 and 6 is very similar to the transition we saw in the previous scenario between the handshake ACK and GET request. However, in this case, the server is the focus of our concern.

Packet 5 is the ACK that the server sends in response to the GET request it received from the client. As soon as that packet has been sent, the server should begin sending data almost immediately. The accessing, packaging, and transmitting of the data in this packet is done by the HTTP protocol, and because this is an application layer protocol, a bit of processing is required by the server. The delay in receipt of this packet indicates that the server was unable to process this data in a reasonable amount of time, ultimately pointing to it as the source of latency in this capture file.

Latency Locating Framework

Using six packets, we've managed to locate the source of high network latency from the client and the server. These scenarios may seem a bit complex, but the diagram shown in Figure 9-26 should make the process a bit quicker when troubleshooting your own latency issues. These principles can be applied to almost any TCP-based communication.

NOTE Notice that we have not talked a lot about UDP latency. Because UDP is designed to be quick but unreliable, it doesn't have any built-in features to detect and recover from latency. Instead, it relies on the application layer protocols (and ICMP) that it's paired with to handle data delivery reliability.

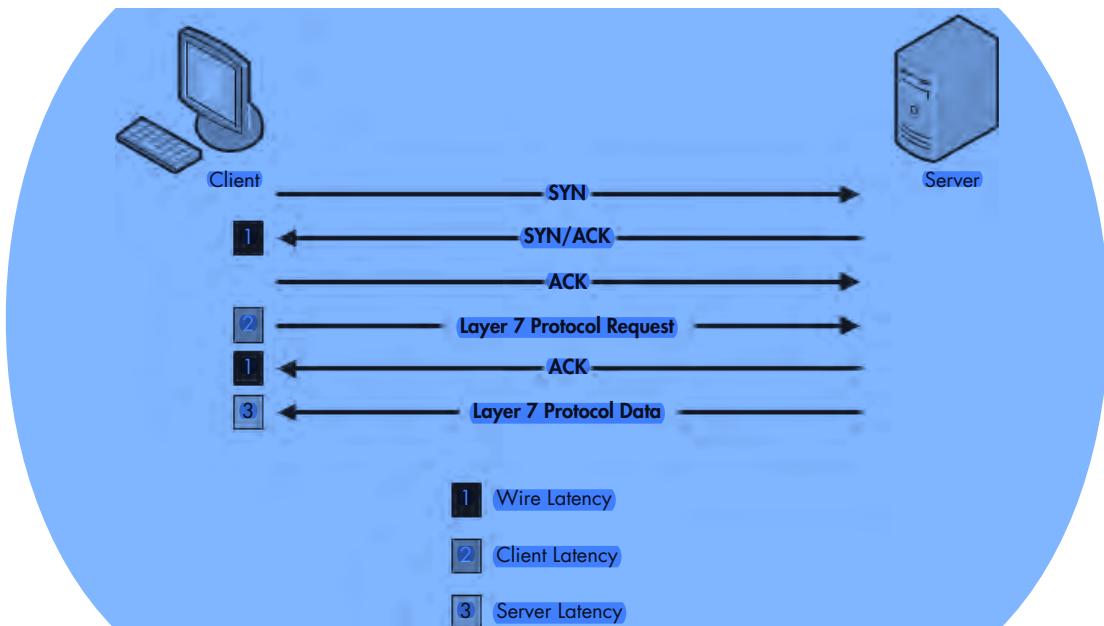


Figure 9-26: This diagram can be used to troubleshoot your own latency issues.

Network Baselineing

When all else fails, your *network baseline* can be one of the most crucial pieces of data you have when troubleshooting slowness on the network. For our purposes, a network baseline consists of a sample of traffic from various points on the network that includes a large chunk of what we would consider “normal” network traffic. The goal of the network baseline is to serve as a basis of comparison when the network or devices on it are not acting correctly.

For example, consider a scenario in which several clients on the network complain of slowness when logging in to a local web application server. If you were to capture this traffic and compare it to a network baseline, you might find that the web server is responding normally but that the external DNS requests resulting from external content embedded into the web application are running twice as slowly as normal.

You might have noticed the slow external DNS server without the aid of a network baseline, but when you are dealing with subtle changes, that may not be the case. Ten DNS queries taking 0.1 seconds longer than normal to process is just as bad as one DNS query taking 1 full second longer than normal, but the former is much harder to detect without a network baseline.

Because no two networks are alike, the components of a network baseline can vary drastically. The following sections provide examples of the components of a network baseline. You may find that all of these items apply to your network infrastructure or that very few of them do. Regardless, you should be able to place each component of your baseline inside one of three basic baseline categories: site, host, and application.

Site Baseline

The purpose of the site baseline is to gain an overall snapshot of the traffic at each physical site on your network. Ideally, this would be every segment of the WAN.

Components of this baseline might include the following:

Protocols in use

Use the Protocol Hierarchy Statistics window (**Statistics ▶ Protocol Hierarchy**) while capturing traffic from all of the devices on the network segment at the network edge (router/firewall), so that you can see traffic from all devices. Later, you can compare against this to find out if normally present protocols are missing or if new protocols have introduced themselves on the network. You can also use this to find above ordinary amounts of certain types of traffic based on protocol.

Broadcast traffic

This includes all broadcast traffic on the network segment. Sniffing at any point within the site should let you capture all of the broadcast traffic, allowing you to know who or what normally sends a lot of broadcast traffic out on the network, so you can quickly determine whether you have too much (or not enough) broadcasting going on.

Authentication sequences

These include traffic from authentication processes on random clients to all services, such as Active Directory, web applications, and organization-specific software. Authentication is one area where services are commonly slow. The baseline allows you to determine if authentication is to blame for slow communications.

Data-transfer rate

This usually consists of a measure of a large data transfer from the site to various other sites in the network. You can use the capture summary and graphing features of Wireshark to determine the transfer rate and consistency of the connection. This is probably the most important site baseline you can have. Whenever any connection entering or leaving the network segment seems slow, you can perform the same data transfer as in your baseline and compare the results. This will tell you if the connection is actually slow and possibly even help you find the area in which the slowness begins.

Host Baseline

Having a host baseline doesn't mean that you must baseline every single host within your network. The host baseline should be performed on only high-traffic or mission-critical servers. Basically, if a slow server will result in angry phone calls from management, you should have a baseline of that host.

Components of the host baseline include the following:

Protocols in use

This baseline provides a good opportunity to use the Protocol Hierarchy Statistics window while capturing traffic from the host. Later, you can compare against this to find out if normally present protocols are missing or if new protocols have introduced themselves on the host. You can also use this to find above ordinary amounts of certain types of traffic based on protocol.

Idle/busy traffic

This baseline simply consists of general captures of normal operating traffic during peak and off-peak times. Knowing the number of connections and amount of bandwidth used by those connections at different points of the day will allow you to determine if slowness is a result of user load or another issue.

Startup/shutdown

In order to obtain this baseline, you will need to create a capture of the traffic generated during the startup and shutdown sequences of the host. If the computer refuses to boot, refuses to shut down, or is abnormally slow during either sequence, you can use this to determine if the cause is network-related.

Authentication sequences

This baseline requires capturing traffic from authentication processes to all services on the host. Authentication is one area where services are commonly slow. The baseline allows you to determine if authentication is to blame for slow communications.

Associations/dependencies

This baseline consists of a longer duration capture to determine what other hosts this host is dependent upon (and are dependent upon this host). You can use the Conversations window (**Statistics ► Conversations**) to see these associations and dependencies. An example of this is a SQL Server host on which a web server depends. We are not always aware of the underlying dependencies between hosts, so the host baseline can be used to determine these. From there, you can determine if a host is not functioning properly due to a malfunctioning or high-latency dependency.

Application Baseline

The final network baseline category is the application baseline. This baseline should be performed on all business-critical network-based applications.

The following are the components on the application baseline:

Protocols in use

Again, for this baseline, use the Protocol Hierarchy Statistics window in Wireshark, this time while capturing traffic from the host running the application. Later, you can compare against this list to find out if protocols that the application depends on are functioning incorrectly or not at all.

Startup/shutdown

This baseline includes a capture of the traffic generated during the startup and shutdown sequences of the application. If the application refuses to start or is abnormally slow during either sequence, you can use this to determine the cause.

Associations/dependencies

This baseline requires a longer duration capture in which the Conversations window can be used to determine the other hosts and applications on which this application depends. We are not always aware of the underlying dependencies between applications, so this baseline can be used to determine those. From there, you can determine if an application is not functioning properly due to a malfunctioning or high-latency dependency.

Data-transfer rate

You can use the capture summary and graphing features of Wireshark to determine the transfer rate and consistency of the connections to the application server during its normal operation to create this baseline. Whenever the application is reported as being slow, you can use this baseline to determine if the issues being experienced are a result of high utilization or a high user load.

Additional Notes on Baselines

Here are a few more points to keep in mind when creating your network baseline:

- When creating your baselines, do each one at least three times: once during a low-traffic time (early morning), once during a high-traffic time (mid-afternoon), and once during a no traffic time (late night).
- When possible, avoid capturing directly from the hosts you are baselining. During periods of high traffic, this may put an increased load on the device, hurt its performance, and cause your baseline to be invalid due to dropped packets.

- Your baseline will contain some very intimate information about your network, so be sure to secure it. Store it in a safe place where only the appropriate individuals have access. But at the same time, keep it close so that it remains functional for you. Consider keeping it on a USB flash drive or on an encrypted partition.
- Keep all *.pcap* files associated with your baseline and create a “cheat sheet” of the more commonly referenced values, such as associations or average data-transfer rates.

Final Thoughts

This chapter has focused on troubleshooting slow networks. We’ve covered some of the more useful reliability detection and recovery features of TCP, demonstrated how to locate the source of high latency in network communications, and discussed the importance of a network baseline and some of its components. Using the techniques discussed here, along with some of Wireshark’s graphing and analysis features (as discussed in Chapter 5), you should be well equipped to troubleshoot when you get that call complaining that the network is slow.

10

PACKET ANALYSIS FOR SECURITY



Although most of this book focuses on using packet analysis for network troubleshooting, a considerable amount of real-world packet analysis is done for security purposes. This could be the job of an intrusion analyst reviewing network traffic from potential intruders, or of a forensic investigator attempting to ascertain the extent of a malware infection on a compromised host. Packet analysis for security is a big topic, suitable for an entire book. This chapter provides a taste of analyzing packets with a security focus.

In this chapter, we'll take the viewpoint of a security practitioner, as we examine different aspects of a system compromise at the network level. We'll cover network reconnaissance, malicious traffic redirection, and system exploitation. Next, we'll take on the role of an intrusion analyst, as we dissect traffic based on alerts from an intrusion-detection system (IDS). Reading this chapter will provide you with critical insight into network security, even if you are not in a security-focused role.

Reconnaissance

The first step that an attacker takes is to perform in-depth research on the target system. This step, commonly referred to as *footprinting*, is often accomplished using various publicly available resources, such as the target company's website or Google. Once this research is completed, the attacker will typically begin scanning the IP address (or DNS name) of its target for open ports or running services.

This scanning allows the attacker to determine whether the target is alive and reachable. For example, consider a scenario in which a bank robber is planning to steal from the largest bank in the city, located at 123 Main Street. He spends weeks planning an elaborate heist, only to find out upon arrival at the address that the bank has moved to 555 Vine Street. Worse yet, imagine a scenario in which the robber plans on walking into the bank during normal business hours, intending to steal from the vault, only to get to the bank and discover it is closed that day. Ensuring the target is alive and accessible is the first hurdle that must be crossed.

Another important result of scanning is that it tells the attacker on which ports the target is listening. Returning to our bank robber analogy, consider what would happen if the robber showed up at the bank with absolutely no knowledge of the building's physical layout. He would have no idea of how to gain access to the building, because he wouldn't know the weak points in its physical security.

In this section, we'll discuss a few of the more common scanning techniques used to identify hosts, their open ports, and vulnerabilities on a network.

NOTE *So far, this book has referred to the sides of a connection as the transmitter and receiver or as the client and server. This chapter refers to each side of the communication as either the attacker or the victim.*

SYN Scan

`synscan.pcap`

The type of scanning often done first against a system is a *TCP SYN scan*, also known as a *stealth scan* or a *half-open scan*. A SYN scan is the most common type for several reasons:

- It is very fast and reliable.
- It is accurate on all platforms, regardless of TCP stack implementation.
- It is less noisy than other scanning techniques.

The TCP SYN scan relies on the three-way handshake process to determine which ports are open on a target host. The attacker sends a TCP SYN packet to a range of ports on the victim, as if trying to establish a channel for normal communication on the ports. Once this packet is received by the victim, one of a few things may happen, as shown in Figure 10-1.

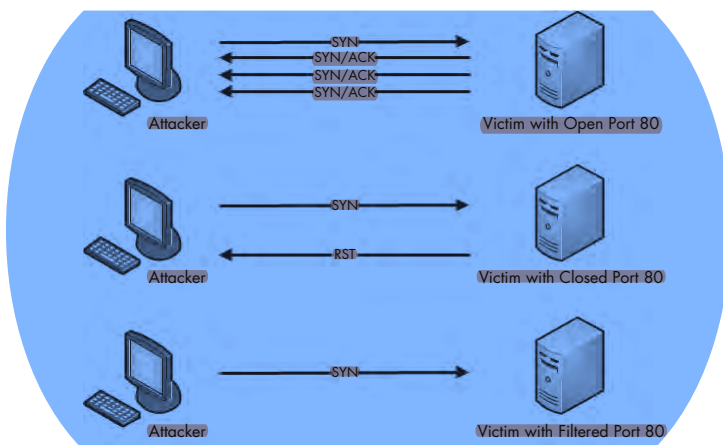


Figure 10-1: Possible results of a TCP SYN scan

If a service on the victim's machine is listening on a port that receives the SYN packet, it will reply to the attacker with a TCP SYN/ACK packet, the second part of the TCP handshake. Then the attacker knows that port is open and a service is listening on it. Under normal circumstances, a final TCP ACK would be sent in order to complete the connection handshake, but in this case, the attacker doesn't want that to happen, since he will not be communicating with the host further at this point. So, the attacker doesn't attempt to complete the TCP handshake.

If no service is listening on a scanned port, the attacker will not receive a SYN/ACK. Depending on the configuration of the victim's operating system the attacker could receive an RST packet in return, indicating that the port is closed. Alternatively, the attacker may receive no response at all. That could mean that the port is filtered by an intermediate device, such as a firewall or the host itself. On the other hand, it could just be that the response was lost in transit. This result typically indicates that the port is closed, but it's ultimately inconclusive.

The file `synscan.pcap` provides a great example of a SYN scan performed with the NMAP tool. NMAP is a robust network-scanning application developed by Fyodor. It can perform just about any kind of scan you can imagine. You can download NMAP for free from <http://www.nmap.com/download.html>.

Our sample capture contains roughly 2,000 packets, which tells us that this scan is reasonably sized. One of the best ways to ascertain the scope of a scan of this nature is to view the Conversations window, as shown in Figure 10-2. There, you should see only one IPv4 conversation ❶ between the attacker (172.16.0.8) and the victim (63.13.134.52). You will also see that there are 1,994 TCP conversations between these two hosts ❷—basically a new conversation for every port pairing involved in the communications.

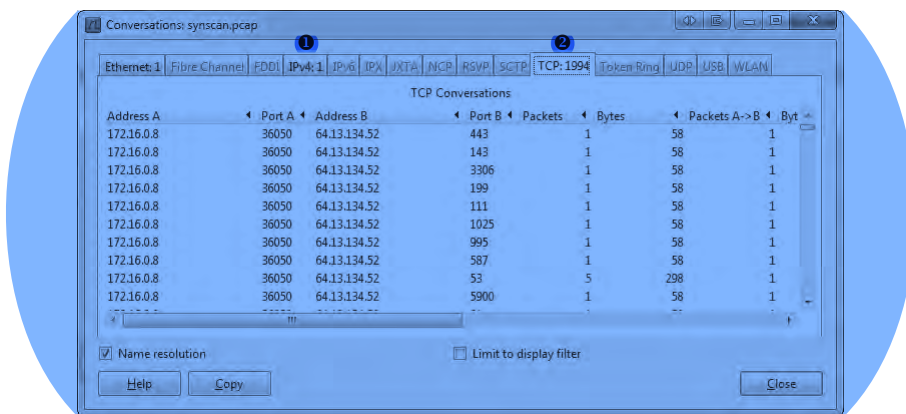


Figure 10-2: The Conversations window shows the variety of TCP communications taking place.

The scanning is occurring very quickly, so scrolling through the capture file is not the best way to find the response associated with each initial SYN packet. Several more packets might be sent before a response to the original packet is received. Fortunately, we can create filters to help us find the right traffic.

Using Filters with SYN Scans

As an example of filtering, let's consider the first packet, which is a SYN packet sent to the victim on port 443 (HTTPS). To see if there was a response to this packet, we can create a filter to show all traffic to and from port 443. Here's how to do this quickly:

1. Select the first packet in the capture file.
2. Expand the TCP header in the Packet Details pane.
3. Right-click the **Destination Port** field, select **Prepare as Filter**, and click **Selected**.
4. This will place a filter in the filter dialog for all packets with the destination port of 443. Now, because we also want all packets from the source port 443, click in the filter dialog at the top of the screen and erase the *dst* portion of the filter.

The resulting filter will yield two packets, which are both TCP SYN packets sent from attacker to victim, as shown in Figure 10-3.

No.	Time	Source	Destination	Protocol	Info
1	0.000000	172.16.0.8	64.13.134.52	TCP	36050 > 443 [SYN] Seq=3713172248 win=3072 Len=0 MSS=1460
32	0.000065	172.16.0.8	64.13.134.52	TCP	36051 > 443 [SYN] Seq=3713237785 Win=2048 Len=0 MSS=1460

Figure 10-3: Two attempts to establish a connection with SYN packets

Since there is no response to either of these packets, it's possible that the response is being filtered by the victim host or an intermediary device, or that the port is closed. Ultimately, the result of the scan against port 443 is inconclusive.

We can attempt this same technique on another packet to see if we get different results. To do so, first clear the previously created filter by clicking the **Clear** button next to the filter area. Then select the ninth packet in the list. This is a SYN packet to port 53, commonly associated with DNS. Using the method outlined in the previous steps, create a filter based on the destination port and erase the *dst* portion of the filter so that it applies to all TCP port 53 traffic. When you apply this filter, you should see five packets, as shown in Figure 10-4.

No.	Time	Source	Destination	Protocol	Info
9	0.000052	172.16.0.8	64.13.134.52	TCP	36050 > 53 [SYN] Seq=3713172248 win=3072 Len=0 MSS=1460
11	0.061832	64.13.134.52	172.16.0.8	TCP	53 > 36050 [SYN, ACK] Seq=1117405124 Ack=3713172249 win=5840 Len=0 MSS=1380
529	0.057126	64.13.134.52	172.16.0.8	TCP	53 > 36050 [SYN, ACK] Seq=1117405124 Ack=3713172249 win=5840 Len=0 MSS=1380
2006	4.930109	64.13.134.52	172.16.0.8	TCP	53 > 36050 [SYN, ACK] Seq=1117405124 Ack=3713172249 win=5840 Len=0 MSS=1380
2009	10.029025	64.13.134.52	172.16.0.8	TCP	53 > 36050 [SYN, ACK] Seq=1117405124 Ack=3713172249 win=5840 Len=0 MSS=1380

Figure 10-4: Five packets indicating a port is open

The first of these packets is the SYN we selected at the beginning of the capture. The second is an actual response from the victim. It's a TCP SYN/ACK—the response expected when setting up the three-way handshake. Under normal circumstances, the next packet would be an ACK from the host that sent the initial SYN. However, in this case, our attacker doesn't want to complete the connection and doesn't send a response. As a result, the victim retransmits the SYN/ACK three more times before giving up. Since a SYN/ACK response is received when attempting to communicate with the host on port 53, it's safe to assume that a service is listening on that port.

Let's rinse and repeat this process one more time for packet 13. This is a SYN packet sent to port 113, which is commonly associated with the Ident protocol, often used for IRC identification and authentication services. If you apply the same type of filter to the port listed in this packet, you will see four packets, as shown in Figure 10-5.

No.	Time	Source	Destination	Protocol	Info
13	0.000070	172.16.0.8	64.13.134.52	TCP	36050 > 113 [SYN] Seq=3713172248 win=4096 Len=0 MSS=1460
14	0.061491	64.13.134.52	172.16.0.8	TCP	113 > 36050 [RST, ACK] Seq=2462244745 Ack=3713172249 Win=0 Len=0
530	0.006942	172.16.0.8	64.13.134.52	TCP	36061 > 113 [SYN] Seq=3696394776 win=2048 Len=0 MSS=1460
571	0.000827	64.13.134.52	172.16.0.8	TCP	113 > 36061 [RST, ACK] Seq=1027049353 Ack=3696394777 Win=0 Len=0

Figure 10-5: A SYN followed by a RST, indicating the port is closed

The first packet is the initial SYN, which is followed immediately by a RST from the victim. This is an indication that the victim is not accepting connections on the targeted port, and that a service is most likely not running on it.

Identifying Open and Closed Ports

After understanding the different types of response a SYN scan can elicit, the next logical thought is to find a fast method of identifying which ports are open or closed. The answer lies within the Conversations window once again. In this window, you can sort the TCP conversations by packet number, with the highest values at the top by clicking the Packets column twice, as shown in Figure 10-6.

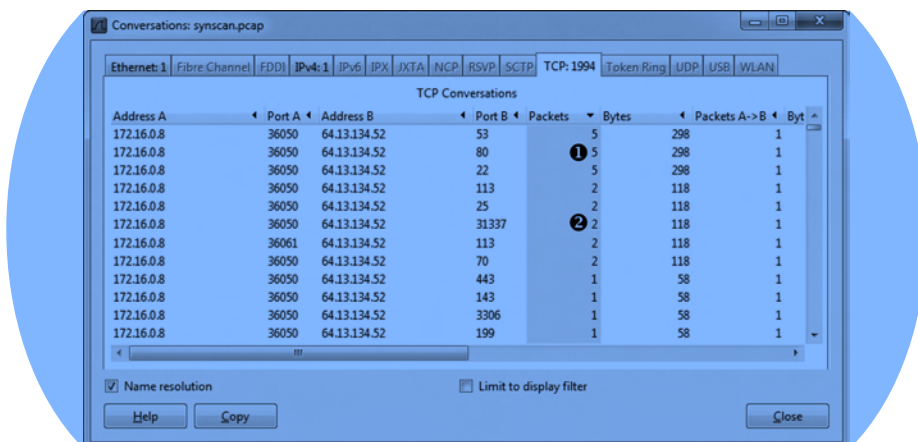


Figure 10-6: Finding open ports with the Conversations window

Three scanned ports include five packets in each of their conversations ❶. We know that ports 53, 80, and 22 are open, because these five packets represent the initial SYN, the corresponding SYN/ACK, and the retransmitted SYN/ACKs from the victim.

For five ports, only two packets were involved in the communication ❷. The first is the initial SYN, and the second is the RST from the victim. This indicates that ports 113, 25, 31337, 113, and 70 are closed.

The remaining entries in the Conversations window include only one packet, meaning that the victim host never responded to the initial SYN. These remaining ports are most likely closed, but we're not sure.

Operating System Fingerprinting

An attacker puts a great deal of value on knowing his target's operating system. Knowledge of the operating system in use ensures that all of the attack methods employed by the attacker are configured correctly for that system. This also allows the attacker to know the location of certain critical files and directories within the target file system, should he actually succeed in accessing the system.

Operating system fingerprinting is the name given to a group of techniques used to determine the operating system running on a system without actually having physical access to that system. There are two types of operating system fingerprinting: passive and active.

Passive Fingerprinting

passiveosfinger-
printing.pcap

Using *passive fingerprinting*, you examine certain fields within packets sent from the target in order to determine the operating system in use. The technique is considered passive because you only listen to the packets the target host is sending and don't actively send any packets to the host yourself. This is the most ideal type of operating system fingerprinting for attackers, because it allows them to be stealthy.

That being said, how can we determine which operating system a host is running based on nothing but the packets it sends? Well, this is actually pretty easy and is made possible by the lack of specificity in the specifications defined by protocol RFCs. Although the various fields contained in TCP, UDP, and IP headers are very specific, typically, no default values are defined for these fields. This means that the TCP/IP stack implementation in each operating system must define its own default values for these fields. Table 10-1 lists some of the more common fields and the default values that can be used to link them to various operating systems.

Table 10-1: Common Passive Fingerprinting Values

Protocol Header	Field	Default Value	Operating System
IP	Initial Time to Live	64	NMap, BSD, Mac OS 10, Linux
		128	Novell, Windows
		255	Cisco IOS, Palm OS, Solaris
IP	Don't Fragment Flag	Set	BSD, Mac OS 10, Linux, Novell, Windows, Palm OS, Solaris
		Not set	Nmap, Cisco IOS
TCP	Max Segment Size	0	Nmap
		1440	Windows, Novell
		1460	BSD, Mac OS 10, Linux, Solaris
TCP	Window Size	1024–4096	Nmap
		65535	BSD, Mac OS 10
		2920–5840	Linux
		16384	Novell
		4128	Cisco IOS
		24820	Solaris
		Variable	Windows
TCP	SackOK	Set	Linux, Windows, OpenBSD
		Not set	Nmap, FreeBSD, Mac OS 10, Novell, Cisco IOS, Solaris

The packets contained in the file *passiveosfingerprint.pcap* are great examples of this technique. There are two packets in this file. Both are TCP SYN packets sent to port 80, but they come from different hosts. Using only the values contained in these packets and referring to Table 10-1, we should be able to determine the operating system architecture in use on each host. The details of each packet are shown in Figure 10-7.

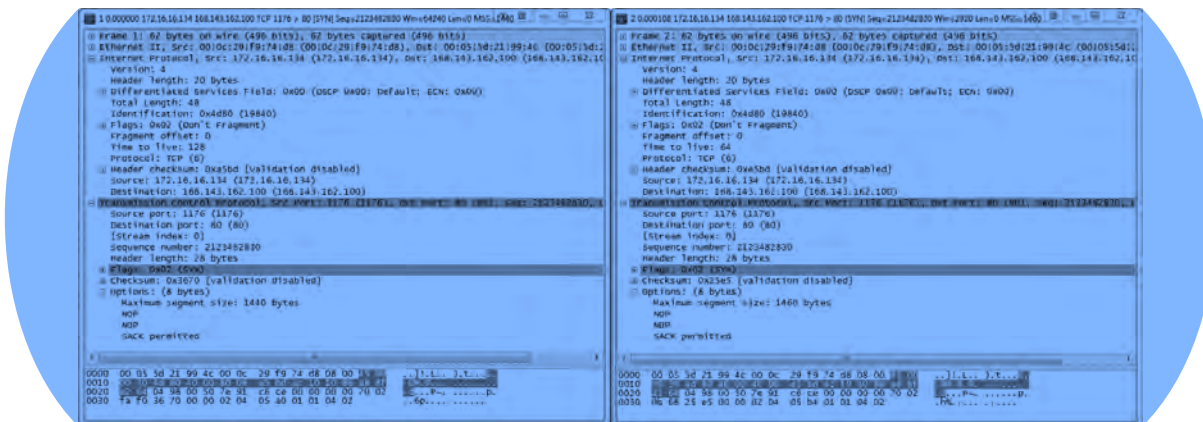


Figure 10-7: These packets can tell us which operating system they were sent from.

Using Table 10-1 as a reference, we can create Table 10-2, which is a breakdown of the relevant fields in these packets.

Table 10-2: Breakdown of the Operating System Fingerprinting Packets

Protocol Header	Field	Packet 1 Value	Packet 2 Value
IP	Initial Time to Live	128	64
IP	Don't Fragment Flag	Set	Set
TCP	Max Segment Size	1440 Bytes	1460 Bytes
TCP	Window Size	64240 Bytes	2920 Bytes
TCP	SackOK	Set	Set

Based on these values, we can conclude that packet 1 was most likely sent by a device running Windows, and packet 2 was most likely sent by a device running Linux.

Keep in mind that the list of common passive fingerprinting identifying fields in Table 10-1 is by no means exhaustive. There are many quirks that may result in deviations from these expected values. Therefore, you cannot fully rely on the results gained from passive operating system fingerprinting.

NOTE One tool that uses operating system fingerprinting techniques is *p0f*. This tool analyzes relevant fields in a packet capture and outputs the suspected operating system. Using tools like *p0f*, you can not only get the operating system architecture, but sometimes even the appropriate version or patch level. You can download *p0f* from <http://lcamtuf.coredump.cx/p0f.shtml>.

Active Fingerprinting

activeosfinger-
printing.pcap

When passively monitoring traffic doesn't yield the desired results, a more direct approach may be required. This approach is called *active fingerprinting*. It involves the attacker actively sending specially crafted packets to the victim in

order to elicit replies that will reveal the operating system in use on the victim's machine. Of course, since this approach involves communicating directly with the victim, it is not the least bit stealthy, but it can be highly effective.

The file *activeosfingerprinting.pcap* contains an example of an active operating system fingerprinting scan initiated with the Nmap scanning utility. Several packets in this file are the result of Nmap sending different probes designed to elicit responses that will allow for operating system identification. Nmap records the responses to these probes and builds a fingerprint, which it compares to a database of values in order to make a determination.

NOTE *The techniques used by Nmap to actively fingerprint an operating system are quite complex. To learn more about how Nmap performs active operating system fingerprinting, read the definitive guide to Nmap, Nmap Network Scanning, by the tool's author Gordon "Fyodor" Lyon.*

Exploitation

Every attacker lives for the exploitation phase. The attacker has done his research, performed reconnaissance on the target, and found a vulnerability that he is prepared to exploit in order to gain access to the target system. In the remainder of this chapter, we'll look at packet captures of various exploitation techniques, including an exploit for a semi-recent Microsoft vulnerability, traffic redirection via ARP cache poisoning, and a remote-access Trojan performing data exfiltration.

Operation Aurora

aurora.pcap

In January 2010, Operation Aurora exploited an as yet unknown vulnerability in Internet Explorer. This vulnerability allowed attackers to gain remote root-level control of targeted machines at Google, among other companies.

In order to execute this malicious code, a user simply needed to visit a website using a vulnerable version of Internet Explorer. The attackers then had immediate access to the user's machine with administrative privileges. *Spear phishing*, in which the attackers send an email message to victims designed to get them to click a link leading to a malicious site, was used to lure the victims. Since spear phishing messages appear to come from trusted sources, they are often successful.

In the case of Aurora, we pick up this story as soon as the targeted user clicks the link in the spear phishing email message. The resulting packets are contained in the file *aurora.pcap*.

This capture begins with a three-way handshake between the victim (192.168.100.206) and the attacker (192.168.100.202). The initial connection is to port 80, which would lead us to believe this is HTTP traffic. That assumption is confirmed in the fourth packet, an HTTP GET request for */info* ❶, as shown in Figure 10-8.

Figure 10-8: The victim makes a GET request for /info.

The attacker's machine acknowledges receipt of the GET request and reports a response code of 302 (Moved Temporarily) in packet 6, the status code commonly used to redirect a browser to another page, which is the case here. Along with the 302 response code ❶, a Location field specifies the location `/info?rFfWELUjLJHpP` ❷, as shown in Figure 10-9.

Figure 10-9: The client browser is redirected with this packet.

After receiving the HTTP 302 packet, the client initiates another GET request to the `/info?rFfWELUjLJHpP` URL in packet 7, for which an ACK is received in packet 8. Following the ACK, the next several packets represent data being transferred from the attacker to the victim. To take a closer look at that data, right-click one of the packets in the stream, such as packet 9, and select **Follow TCP Stream**. In this stream output, we see the initial GET request, the 302 redirection, and the second GET request, as shown in Figure 10-10.

After this, things start getting really strange. The attacker responds to the GET request with some very odd-looking content, the first section of which is shown in Figure 10-11.

This content appears to be a series of random numbers and letters inside a `<script>` tag ❶. The `<script>` tag is used within HTML to denote the use of a higher-level scripting language. Within this tag, you normally see various scripting statements. But this gibberish indicates that the content may be encoded to hide it from detection. Since we know this is exploit traffic, we might assume that this obfuscated section of text contains the hexadecimal padding and shellcode used to actually exploit the vulnerable service.

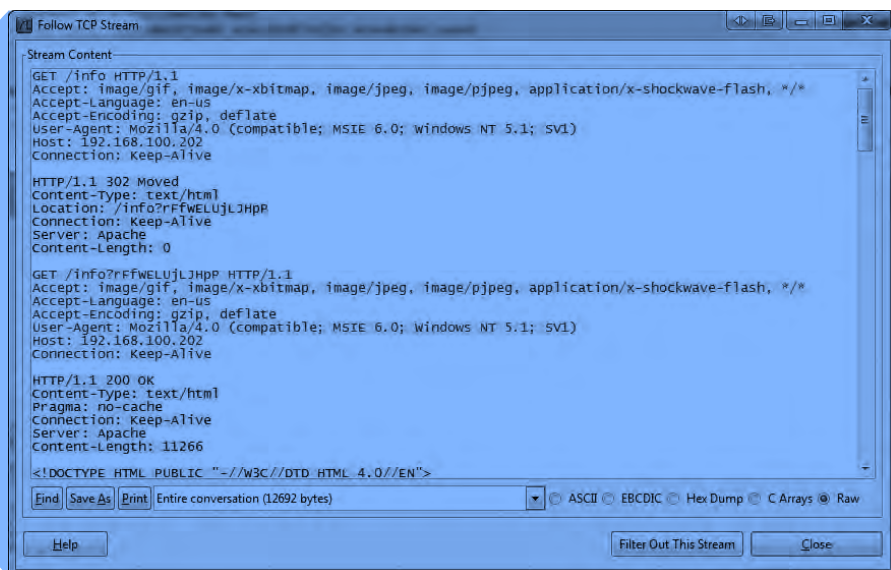


Figure 10-10: The data stream being transmitted to the client

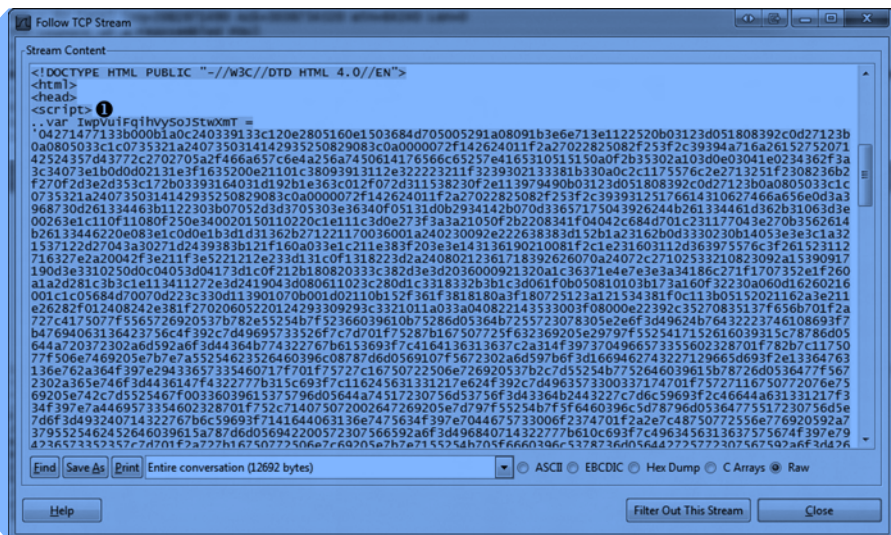


Figure 10-11: This scrambled content within a <script> tag appears to be encoded.

The second portion of the content sent from the attacker is shown in Figure 10-12. After the encoded text, we finally see some text that is readable. Even without extensive programming knowledge, we can see that this text appears to do some string parsing based on a few variables. This is the last bit of text before the closing </script> tag.

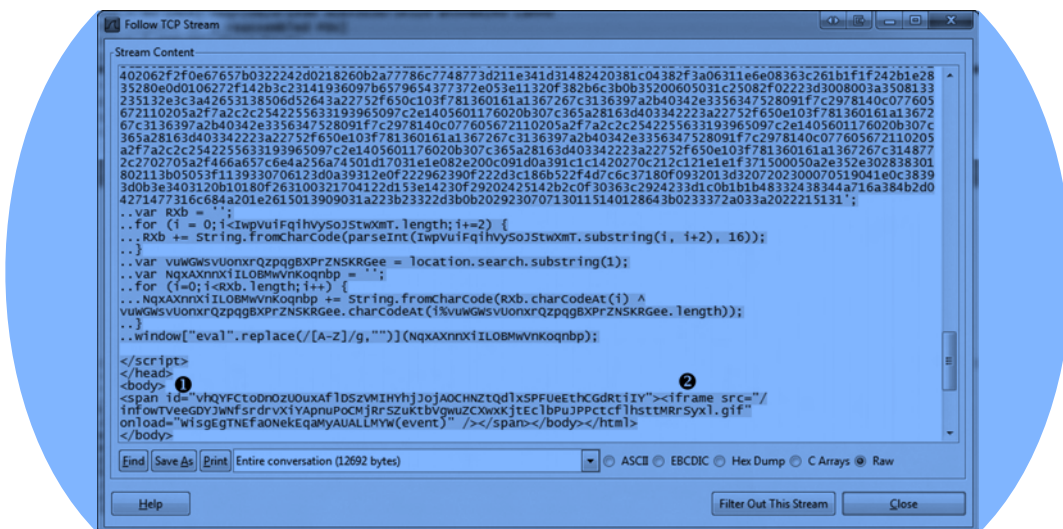


Figure 10-12: This portion of the content sent from the server contains readable text and a suspicious iframe.

The last section of data sent from attacker to client has two parts. The first is a `` section ①. The second is contained within the `/>` tags and is `<iframe src="/infowTveeGDYJWNfSrdrvXiYApnuPoCmJRrSZuKtbVgwuZCXwxKJtEc1bPuJPPctcf1hst-tMRrSyx1.gif" onload="WisEgTNEfaONekEqMyAUALLMYW(event)" />` ②. Once again, this content may be a sign of malicious activity, due to the suspicious long, random strings of unreadable and potentially obfuscated text.

The portion of the code contained within the `` tag is an *iframe*, which is a common method used by attackers to embed additional unexpected content into an HTML page. The `<iframe>` tag creates an inline frame that can go undetected by the user. In this case, the `<iframe>` tag references an oddly named GIF file. As shown in Figure 10-13, when the victim's browser sees the reference to this file, it makes a GET request for it in packet 21 ①, and the GIF is sent immediately following that ②. This GIF is probably used to somehow trigger the exploit code that has already been downloaded to the victim's machine.

No.	Time	Source	Destination	Protocol	Info
21	0.455107	192.168.100.206	192.168.100.202	HTTP	① GET /infowTveeGDYJWNfSrdrvXiYApnuPoCmJRrSZuKtbVgwuZCXwxKJtEc1bPuJPPctcf1hst-tMRrSyx1.gif HTTP/1.1
22	0.199959	192.168.100.202	192.168.100.206	TCP	80 → 1031 [ACK] Seq=3036736951 Ack=3982971911 Win=64518 Len=0
23	0.001166	192.168.100.206	192.168.100.202	HTTP	② HTTP/1.1 200 OK (GIF89a)
24	0.161592	192.168.100.206	192.168.100.202	TCP	1031 → 80 [ACK] Seq=3982971911 Ack=3036737098 Win=64093 Len=0

Figure 10-13: The GIF specified in the iframe is requested and downloaded by the victim.

The most peculiar part of this capture occurs at packet 25, when the victim initiates a connection back to the attacker on port 4321. Viewing this second stream of communication from the Packet Details pane doesn't yield much information, so we will once again view the TCP stream of the communication to get a clearer picture of the data being communicated. Figure 10-14 shows the Follow TCP Stream window output.

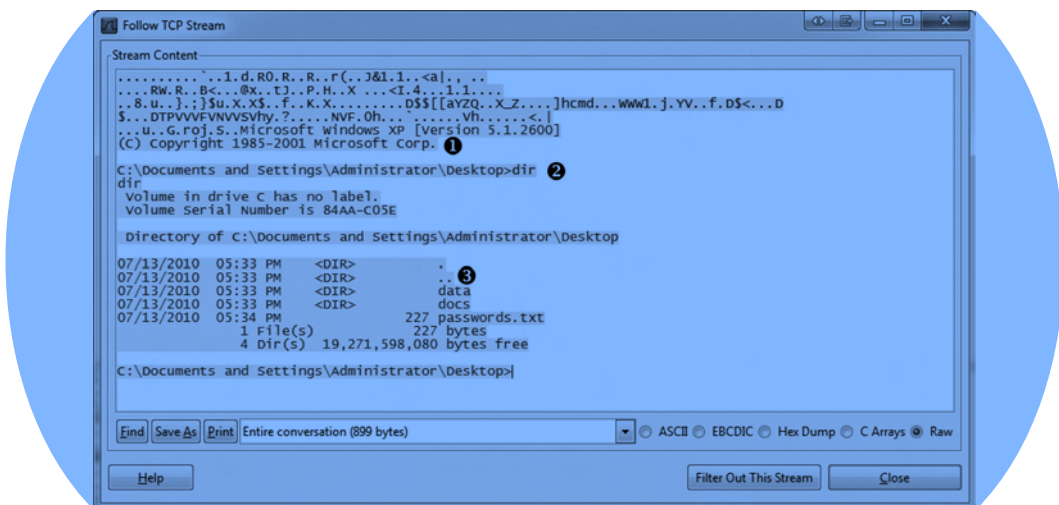


Figure 10-14: The attacker is interacting with a command shell through this connection.

In this display, we see something that should set off immediate alarms: a Windows command shell ❶. This shell is sent from the victim to the server, indicating that the attacker's exploit attempt succeeded and the payload was dropped: The client transmitted a command shell back to the attacker once the exploit was launched. In this capture, we can even see the attacker interacting with the victim by entering the `dir` command ❷ to view a directory listing on the victim's machine ❸.

An attacker with access to this command shell has unrestricted administrative access to the victim's machine and can do virtually anything he wishes to it. With just a single click, in a matter of a few seconds, the victim has just given complete control of his computer to an attacker.

Exploits like this are typically encoded to be unrecognizable when going across the wire in order to prevent them from being picked up by the network IDS. As such, without prior knowledge of this exploit or even a sample of the exploit code, it might be difficult to tell exactly what it is happening on the victim's system without further analysis. Luckily, we were able to pick out some telltale signs of malicious code in this packet capture. This includes the obfuscated text in the `<script>` tags, the peculiar `iframe`, and the command shell seen in plaintext.

Here is a summary of how the Aurora exploit works:

- The victim receives a targeted email from the attacker that appears to be legitimate, clicks a link within it, and sends a GET request to the attacker's malicious site.
- The attacker's web server issues a 302 redirection to the victim, and the victim's browser automatically issues a GET request to the redirected URL.
- The attacker's web server transmits a web page containing obfuscated JavaScript code to the client that includes a vulnerability exploit and an `iframe` containing a link to a malicious GIF image.

- The victim issues a GET request for the malicious image and downloads it from the server.
- The JavaScript code transmitted earlier is deobfuscated using the malicious GIF, and the code executes on the victim's machine, exploiting a vulnerability in Internet Explorer.
- Once the vulnerability is exploited, the payload hidden within the obfuscated code is executed, opening a new session from the victim to the attacker on port 4321.
- A command shell is spawned from the payload and shoveled back to the attacker, so that he may interact with it.

From a defender's point of view, we can use this capture file to create a signature for our IDS that might help capture further occurrences of this attack. For example, we might filter on a nonobfuscated part of the capture, such as the plaintext code at the end of the obfuscated text in the <script> tag. Another train of thought might be to write a signature for all HTTP traffic with a 302 redirection to a site with *info* in the URL. This signature would need some additional tuning in order to be viable in a production environment, but it's a good start.

NOTE *The ability to create traffic signatures based on malicious traffic samples is a crucial step for someone attempting to defend a network against unknown threats. Captures such as the one described here are a great way to develop skills in writing those signatures. To learn more about intrusion detection and attack signatures, visit the Snort project at <http://www.snort.org/>.*

ARP Cache Poisoning

arppoison.pcap

In Chapter 2, we discussed ARP cache poisoning as a way to tap into the wire and intercept traffic from hosts whose packets you need to analyze. ARP cache poisoning can be an effective and useful tool for a network engineer. However, when used with malicious intent, it's also a very lethal form of man-in-the-middle (MITM) attack.

In an MITM attack, an attacker redirects traffic between two hosts in order to intercept or modify it in transit. There are many forms of MITM attacks, including session hijacking, DNS spoofing, and SSL hijacking.

ARP cache poisoning works because specially crafted ARP packets trick two hosts into thinking they are communicating with each other, when, in fact, they are communicating with a third party who is relaying packets as an intermediary.

The file *arppoison.pcap* contains an example of ARP cache poisoning. When you open it, you'll see at first glance that this traffic appears normal. However, if you follow the packets, you will see our victim, 172.16.0.107, browsing to Google and performing a search. As a result of this search, there is quite a bit of HTTP traffic with some DNS queries mixed in.

We know that ARP cache poisoning is a technique that occurs at layer 2, so if we just casually peruse the packets in the Packet List pane, it may be hard to see any foul play. In order to give us a leg up, we will add a couple of columns to the Packet List pane, as follows:

1. Select **Edit ► Preferences**.
2. Click **Columns** on the left side of the Preferences window.
3. Click **Add**.
4. Type **Source MAC** and press **Enter**.
5. In the **Field type** drop-down list, select **Hw src addr (resolved)**.
6. Click the newly added entry, and drag it so that it is directly after the **Source** column.
7. Click **Add**.
8. Type **Dest MAC** and press **Enter**.
9. In the **Field type** drop-down list, select **Hw dest addr (resolved)**.
10. Click the newly added entry and drag it so that it is directly after the **Destination** column.
11. Click **OK** to apply the changes.

When you have completed these steps, your screen should look like Figure 10-15. You should now have two additional columns showing the source and destination MAC addresses of the packets.

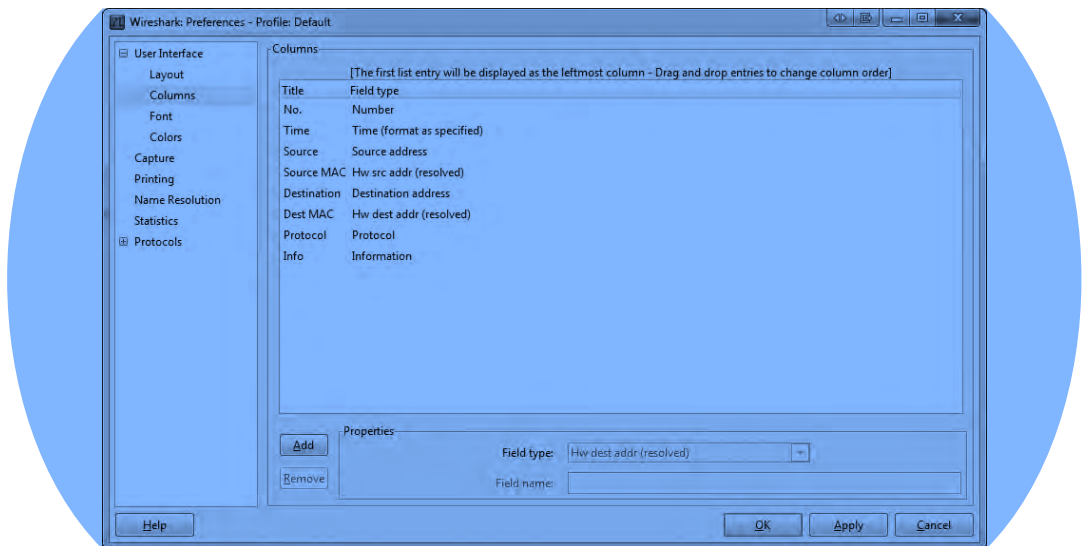


Figure 10-15: The column configuration screen with newly added columns for source and destination hardware addresses

If you still have MAC name resolution turned on, you should see that the communicating devices have MAC addresses that indicate Dell and Cisco hardware. This is very important to remember, because as we scroll through the capture, this changes at packet 54, when we see some peculiar ARP traffic occurring between the Dell host (our victim) and a newly introduced HP host (the attacker), as shown in Figure 10-16.

No.	Time	Source	Source MAC	Destination	Dest MAC	Protocol	Info
54	4.171500	HewlettP_bf:91:ee	HewlettP_bf:91:ee	Dell_c0:56:f0	① Dell_c0:56:f0	ARP	who has 172.16.0.107? Tell 172.16.0.1
② 55	0.000053	Dell_c0:56:f0	Dell_c0:56:f0	HewlettP_bf:91:ee	HewlettP_bf:91:ee	ARP	172.16.0.107 1s at 00:21:70:c0:56:f0
56	0.000013	HewlettP_bf:91:ee	HewlettP_bf:91:ee	Dell_c0:56:f0	Dell_c0:56:f0	ARP	③ 172.16.0.1 1s at 00:25:b3:bf:91:ee

Figure 10-16: Strange ARP traffic between the Dell device and an HP device

Before proceeding further, note the endpoints involved in this communication, which are listed in Table 10-3.

Table 10-3: Endpoints Being Monitored

Role	Device Type	IP Address	MAC Address
Victim	Dell	172.16.0.107	00:21:70:c0:56:f0
Router	Cisco	172.16.0.1	00:26:0b:21:07:33
Attacker	HP	Unknown	00:25:b3:bf:91:ee

But what makes this traffic strange? Recall from our discussion of ARP in Chapter 6 that there are two primary types of ARP packets: a request and a response. The request packet is sent as a broadcast to all hosts on the network in order to find the machine that has the MAC address associated with a particular IP address. The response is then sent as a unicast packet from the machine that replies to the device that transmitted the request. Given this background, we can identify a few peculiar things in this communication sequence, referring to Figure 10-16.

First, packet 54 is an ARP request sent from the attacker, with MAC address 00:25:b3:bf:91:ee, as a unicast packet directly to the victim with MAC 00:21:70:c0:56:f0 ①. This type of request should be broadcast to all hosts on the network, but it targets the victim directly. Also, notice that although this packet is sent from the attacker and includes the attacker's MAC address in the ARP header, it lists the router's IP address rather than its own.

This packet is followed by a response from the victim to the attacker containing its MAC address information ②. The real voodoo here occurs in packet 56, when the attacker sends a packet to the victim with an unsolicited ARP reply telling it that 172.16.0.1 is located at its MAC address, 00:25:b3:bf:91:ee ③. The problem is that the MAC address 172.16.0.1 isn't 00:25:b3:bf:91:ee but 00:26:0b:31:07:33. We know this because we saw the router at 172.16.0.1 communicating with the victim earlier in the packet capture. Since the ARP protocol is inherently insecure (it accepts unsolicited updates to its ARP table), the victim will now be sending traffic that should be going to the router to the attacker instead.

NOTE Because this packet capture was taken from the victim's machine, you don't actually see the entire picture. For this attack to work, the attacker must send the same sequence of packets to the router in order to trick it into thinking the attacker is actually the victim, but we would need to take another packet capture from the router (or the attacker) in order to see those packets.

Once both parties have been duped, the communication between the victim and the router flows through the attacker, as illustrated in Figure 10-17.

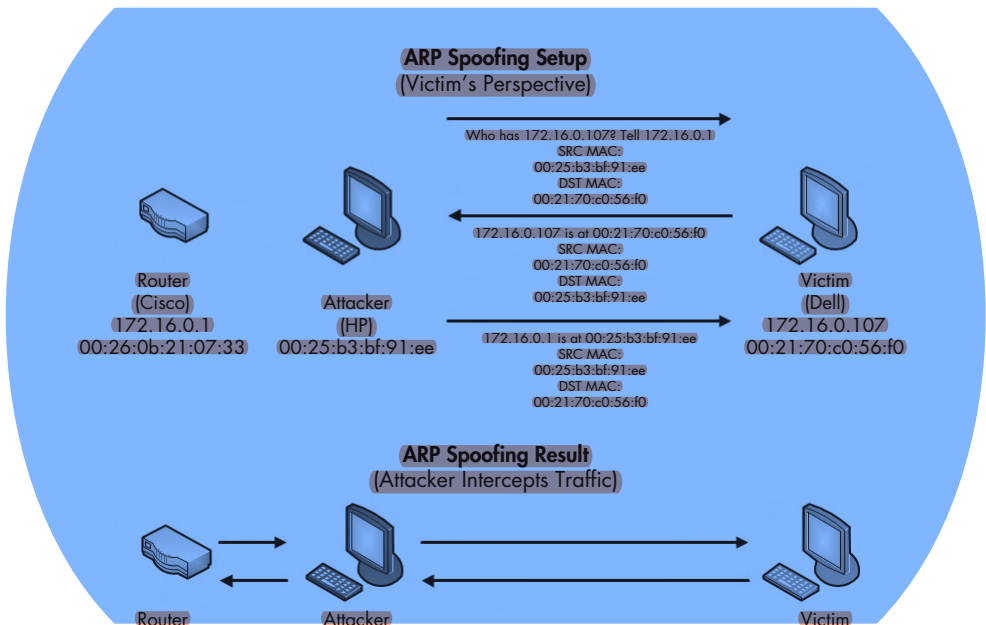


Figure 10-17: ARP cache poisoning as an MITM attack

Packet 57 confirms the success of this attack. When you compare this packet with one sent before the mysterious ARP traffic, such as packet 40 (see Figure 10-18), you will see that the IP address of the remote server (Google) remains the same ①, but the target MAC address has changed ②. This change in MAC address tells us that the traffic is now being routed through the attacker before it gets to the router.

Because this attack is so subtle, it's very difficult to detect. To find it, you typically need the aid of an IDS configured specifically to address it or software running on devices designed to detect sudden changes in ARP table entries. Since you will most likely use ARP cache poisoning to capture packets on networks you are analyzing, it's important to know how this technique can be used against you as well.

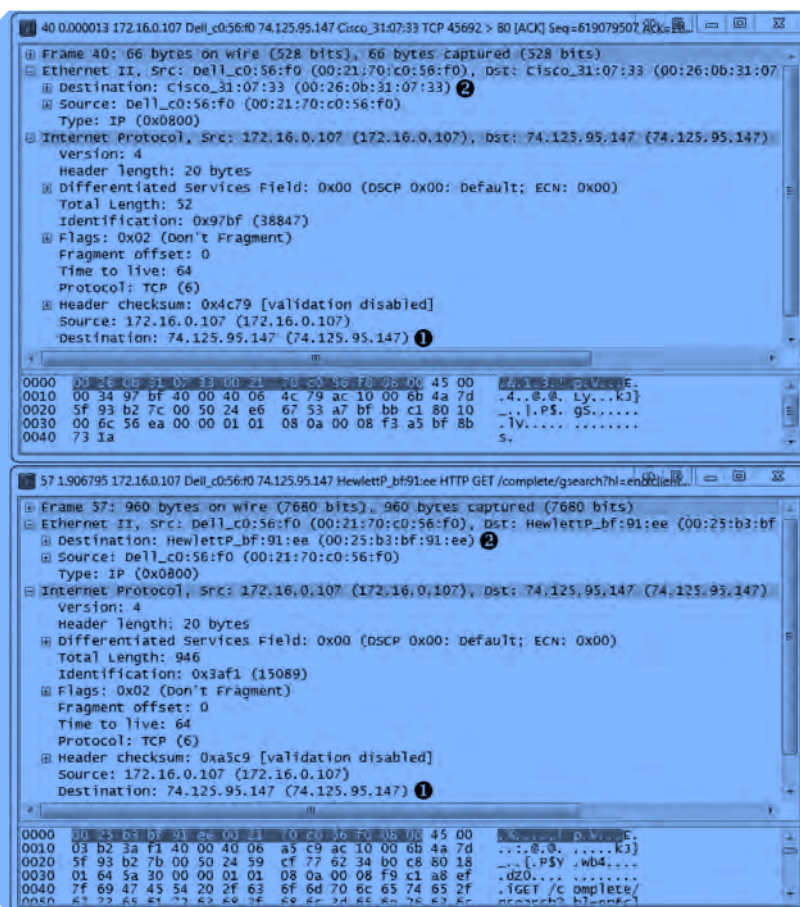


Figure 10-18: The change in target MAC address shows this attack was a success.

Remote-Access Trojan

ratinfected.pcap

So far, we've examined security events with some knowledge of what we have before we begin examining the capture. This is a great way to learn what attacks look like, but it's not very real world. In most real-world scenarios, people tasked with defending a network won't examine every packet that goes across the network. Instead, they will use some form of IDS to alert them to anomalies in network traffic that warrant further examination based on a predefined attack signature.

In the next example, we'll begin with a simple alert, as if we're the real-world analyst. In this case, our IDS generates this alert:

```
[**] [1:132456789:2] CyberEYE RAT Session Establishment [**]
[Classification: A Network Trojan was detected] [Priority: 1]
07/18-12:45:04.656854 172.16.0.111:4433 -> 172.16.0.114:6641
TCP TTL:128 TOS:0x0 ID:6526 IpLen:20 DgmLen:54 DF
***AP*** Seq: 0x53BAEB5E Ack: 0x18874922 Win: 0xFAF0 TcpLen: 20
```

Our next step would be to view the signature rule that triggered this alert:

```
alert tcp any any -> $HOME_NET any (msg:"CyberEYE RAT Session Establishment";  
content:"|41 4E 41 42 49 4C 47 49 7C|"; classtype:trojan-activity;  
sid:132456789; rev:2;)
```

This rule is set to alert whenever it sees a packet from any network entering the internal network with the hexadecimal content 41 4E 41 42 49 4C 47 49 7C. This content converts to *ANA BILGI* in human-readable ASCII. When detected, an alert fires, signifying the possible presence of the CyberEYE *remote-access Trojan (RAT)*. RATs are malicious programs that run silently on a victim's computer and connect back to an attacker, so that the attacker can remotely administer the victim's machine.

NOTE *CyberEYE is a popular Turkish-born tool used to create RAT executables and administer compromised hosts. Ironically, the Snort rule seen here fires on the string ANA BILGI, which is Turkish for BASIC INFORMATION.*

Now we'll look at some traffic associated with the alert in the file *ratinfected.pcap*. This Snort alert would typically capture only the single packet that triggered the alert, but fortunately, we have the entire communication sequence between the hosts involved. To skip to the punch line, search for the hexadecimal string mentioned in the Snort rule, as follows:

1. Select **Edit ► Find Packet**.
2. Select the **Hex Value** radio button.
3. Enter the value **41 4E 41 42 49 4C 47 49 7C** into the text area.
4. Click **Find**.

As shown in Figure 10-19, you should now see the first occurrence of the above string in the data portion of packet 4 ❶.

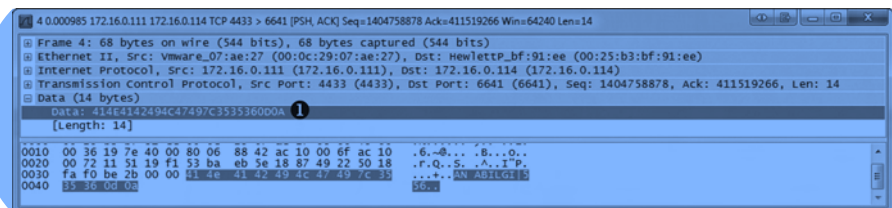


Figure 10-19: The content string in the Snort alert is first seen here in packet 4.

If you select **Edit ► Find Next** a few more times, you will see that this string also occurs in packets 5, 10, 32, 156, 280, 405, 531, and 652. Although all of the communication in this capture file is between the attacker (172.16.0.111) and victim (172.16.0.114), it appears as though some instances of the string occur in different conversations. While packets 4 and 5 are communicating using ports 4433 and 6641, most of the other instances occur between port 4433

and other randomly selected ephemeral ports. We can confirm that multiple conversations exist by looking at the **TCP** tab of the Conversations window, as shown in Figure 10-20.

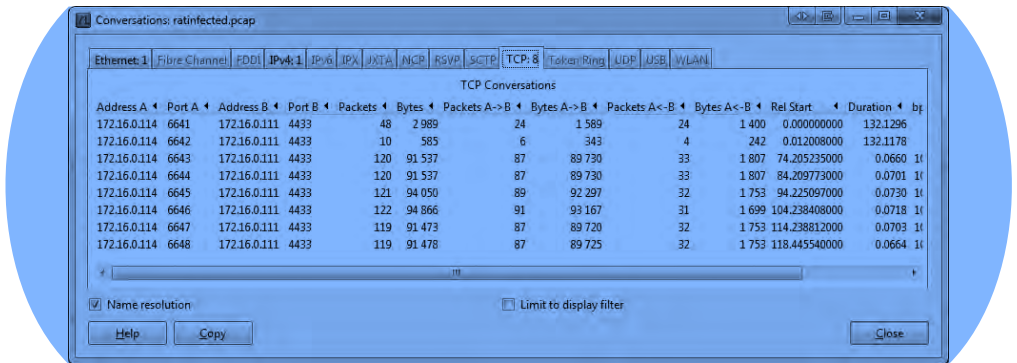


Figure 10-20: Three individual conversations exist between the attacker and victim.

We can visually separate the different conversations in this capture file by coloring them, as follows:

1. In the filter dialog above the Packet List pane, type the filter **(tcp.flags.syn == 1) && (tcp.flags.ack == 0)**. Then click **Apply**. This will select the initial SYN packet for each conversation in the traffic.
2. Right-click the first packet and select **Colorize Conversation**.
3. Select **TCP**, and then select a color.
4. Repeat this process for the remaining SYN packets, choosing a different color for each.
5. When finished, click **Clear** to remove the filter.

Having colored the conversations, we can see how they relate to each other, which will help us to better track the communication process between the two hosts. The first conversation (ports 6641/4433) is where the communication between the two hosts begins, so it's a good place to start. Right-click any packet within the conversation and select **Follow TCP Stream** to see the data that was transferred, as shown in Figure 10-21.

Immediately, we see that the text string **ANABILGI |556** is sent from the attacker to the victim ❶. As a result, the victim responds with some basic system information, including the computer name (**CSANDERS-6F7F77**) and the operating system in use (**Windows XP Service Pack 3**) ❷, and begins transmitting the same string of **BAGLIMI?** back to the attacker ❸. The only communication back from the attacker is the string **CAPSCREEN60** ❹, which appears six times.

This **CAPSCREEN60** string returned by the attacker is interesting, so let's see where it leads. To do so, we search for the text string within the packets using the search dialog again, specifying the **String** option.

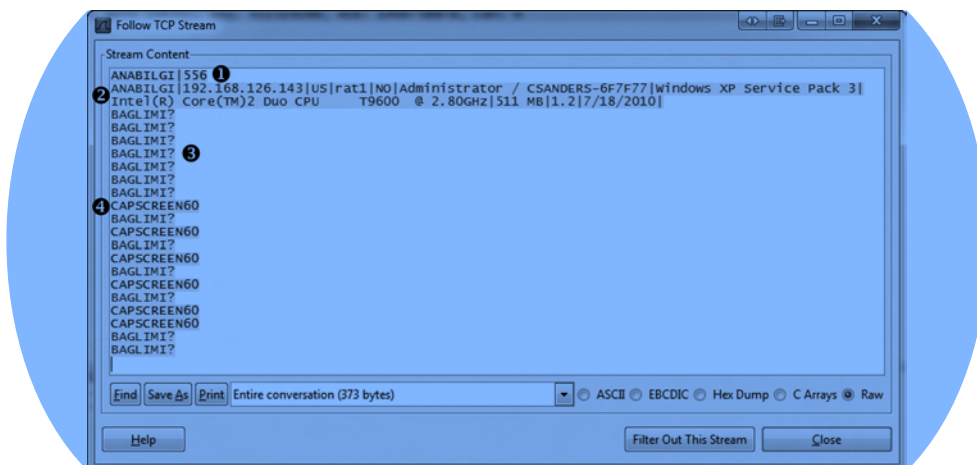


Figure 10-21: The first conversation yields interesting results.

Upon performing this search, we find the first instance of the string in packet 27. The intriguing thing about this bit of information is that as soon as the string is sent from the attacker to the client, the client acknowledges receipt of the packet, and a new conversation is started in packet 29.

Now, if we follow the TCP stream output of this new conversation (shown in Figure 10-22), we see the familiar string ANABILGI|12, followed by the string SH|556, and finally, the string CAPSCREEN|C:\WINDOWS\jpegvhook.dat|84972. Notice the file path specified after the CAPSCREEN string, which is followed by unreadable text. The most intriguing thing here is that the unreadable text is preceded by the string JFIF, which a quick Google search will tell you is commonly found at the beginning of JPG files.

At this point, it's safe to conclude that the attacker initiated this conversation to transfer this JPG image. But even more importantly, we are beginning to see a command structure evolve from the traffic. It appears that CAPSCREEN is a command sent by the attacker to initiate the transfer of this JPG image. In fact, whenever the CAPSCREEN command is sent, the result is the same. To verify this, view the stream of each conversation, or try Wireshark's IO graphing feature as follows:

1. Select **Statistics ▸ IO Graphs**.
2. Insert the filters **tcp.stream eq 2**, **tcp.stream eq 3**, **tcp.stream eq 4**, **tcp.stream eq 5**, and **tcp.stream eq 6**, respectively, into the five filter dialogs.
3. Click the **Graph 1**, **Graph 2**, **Graph 3**, **Graph 4**, and **Graph 5** buttons to enable the data points for the filters specified.
4. Change the y-axis scale to **Bytes/Tick**.

Figure 10-23 shows the resulting graph.

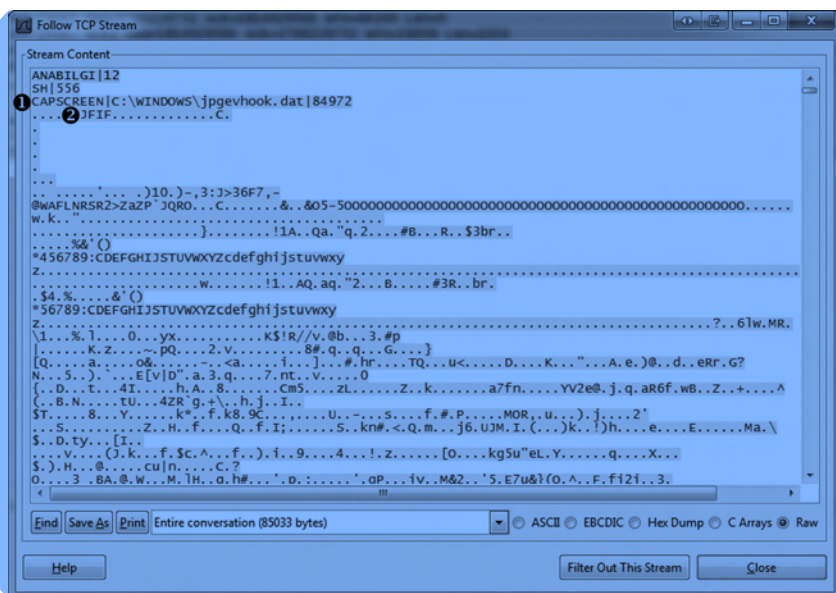


Figure 10-22: The attacker appears to be initiating a request for a JPG file.

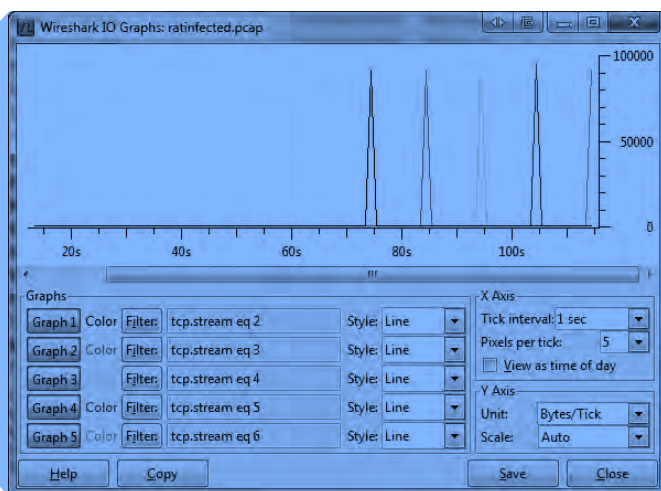


Figure 10-23: This graph shows that the same activity appears to repeat.

Based on this graph, it appears as though each conversation contains the same amount of data and occurs for the same amount of time. We can now conclude that this activity repeats several times.

1. First, follow the TCP stream of the appropriate packets as described in the text preceding Figure 10-22.
2. The communication must then be isolated so that we only see the stream data sent from the victim to the attacker. Do this by selecting the arrow next to the drop-down that says **Entire Conversation (85033 bytes)**. Be sure to select the appropriate directional traffic, which is **172.16.0.114:6643 --> 172.16.0.111:4433 (85020 bytes)**.
3. Save the data by selecting the **Save As** button, ensuring that you save the file with a **.jbg** file extension.

If you try to open the image now you may be surprised to find that it won't open. That's because we have one more step to perform. Unlike the scenario in Chapter 8 where we extracted a file cleanly from FTP traffic, the traffic here added some additional content to the actual data. In this case, the first two lines seen in the TCP stream are actually part of the trojan's command sequence, not part of the data that makes up the JPG (see Figure 10-24). When we saved the stream, this extraneous data was also saved. As a result, the file viewer that is looking for a JPG file header is seeing content that doesn't match what it is expecting, and therefore it can't open the image.

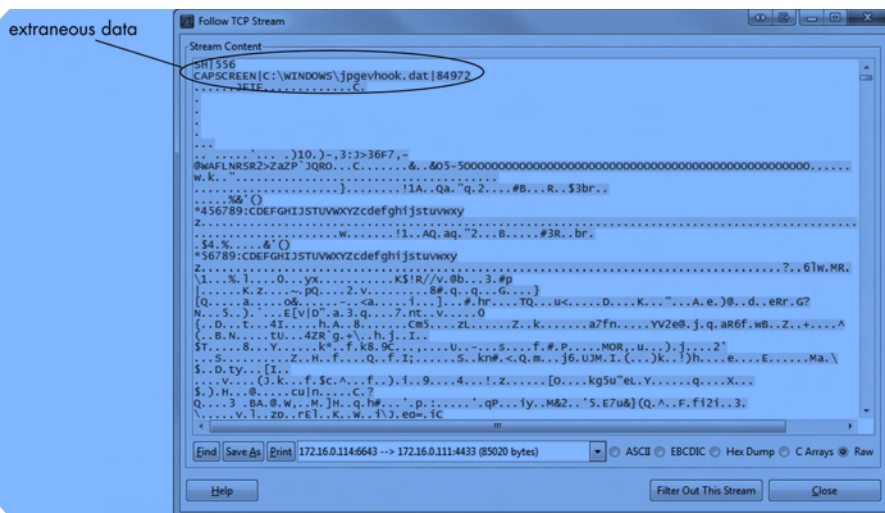


Figure 10-24: The extraneous data added by the trojan prevents the file from being opened correctly.

Fixing this issue is a painless process, requiring a bit of manipulation with a hex editor. This process is called *file carving*. In Figure 10-25, I've used WinHex to highlight the first several bytes of the JPG file. You will need to delete these bytes and save the image file using any hex editor.

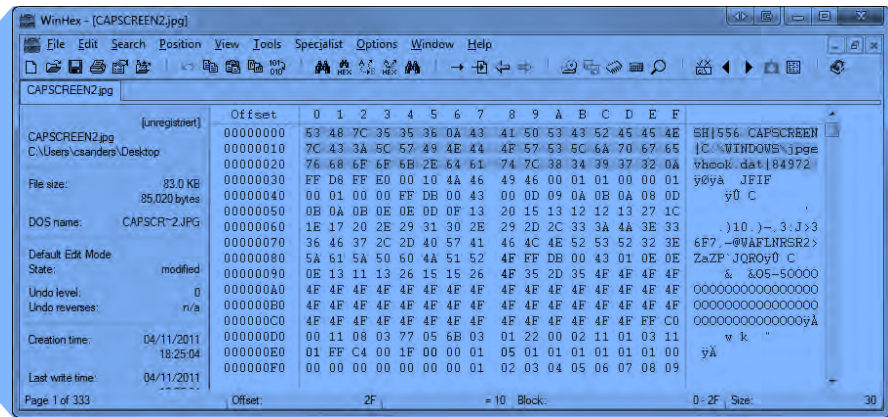



Figure 10-25: Removing the extraneous bytes from the JPG file

With the unneeded bytes of data removed, the file should open. It should be clear now that the trojan is taking screen captures of the victim's desktop and transmitting them back to the attacker (Figure 10-26).



Figure 10-26: The JPG being transferred is a screen capture of the victim's computer.



After these communication sequences have completed, the communication ends with a normal TCP teardown sequence.

This scenario is a prime example of the thought process an intrusion analyst would follow when analyzing traffic based on an IDS alert:

- Examine the alert and the signature that fired it.
- Confirm that the signature was actually in the traffic in the proper context.
- Examine traffic to find out what the attacker did with the compromised machine.
- Begin containment of the issues before any more sensitive information leaks from the compromised victim.

Final Thoughts

Entire books could be written on breaking down packet captures in security-related scenarios, analyzing common attacks, and responding to IDS alerts. In this chapter we've examined some common scanning and enumeration types, a common MITM attack, and two examples of how a system might be exploited and what might happen once it has been owned.



11

WIRELESS PACKET ANALYSIS



The world of wireless networking is a bit different than traditional wired networking. Although we are still dealing with common communication protocols such as TCP and IP, the game changes a bit when moving to the lowest levels of the OSI model. Here, the data link layer is of special importance due to the nature of wireless networking and the physical layer. This puts new restrictions on the data we access and how we capture it.

Given these extra considerations, it should come as no surprise that an entire chapter of this book is dedicated to packet capture and analysis on wireless networks. In this chapter, we will discuss exactly why wireless networks are unique when it comes to packet analysis and how to overcome the challenges. Of course, we will be doing this by looking at actual practical examples of wireless network captures.

Physical Considerations

The first thing to consider about capturing and analyzing data transmitted across a wireless network is the physical transmission medium. Until now, we have not considered the physical layer, because we've been communicating over physical cabling. Now we are communicating through invisible airwaves, with packets flying right by us.

Sniffing One Channel at a Time

The most unique consideration when capturing traffic from a wireless local area network (WLAN) is that the wireless spectrum is a shared medium. Unlike wired networks, where each client has its own network cable connected to a switch, the wireless communication medium is the airspace client's share, which is limited in size. A single WLAN will occupy only a portion of the 802.11 spectrum. This allows multiple systems to operate in the same physical area on different portions of the spectrum.

NOTE *Wireless networking is based on the 802.11 standard, developed by the Institute of Electrical and Electronics Engineers (IEEE). Throughout this chapter, the terms wireless network and WLAN refer to networks that adhere to the 802.11 standard.*

This separation of space is made possible by dividing the spectrum into operation channels. A *channel* is simply a portion of the 802.11 wireless spectrum. In the United States, 11 channels are available (more are allowed in some other countries). This is relevant because, just as a WLAN can operate on only one channel at a time, we can sniff packets on only one channel at a time, as illustrated in Figure 11-1. Therefore, if you are troubleshooting a WLAN operating on channel 6, you must configure your system to capture traffic seen on channel 6.

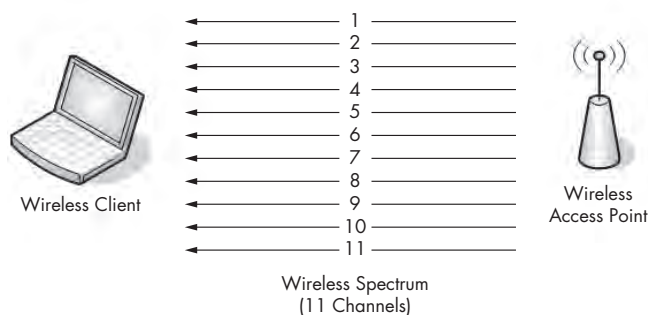


Figure 11-1: Sniffing wirelessly can be tedious, since it can be done on only one channel at a time.

NOTE *Traditional wireless sniffing can only be done one channel at a time, with one exception: Certain wireless scanning applications utilize a technique called channel hopping to change channels rapidly in order to collect data. One of the most popular tools of this type, Kismet (<http://www.kismetwireless.net/>), can hop up to 10 channels per second, which makes it very effective at sniffing multiple channels at once.*

Wireless Signal Interference

With wireless communications, we sometimes can't rely on the integrity of the data being transmitted over the air. It's possible that something will interfere with the signal. Wireless networks include some features to handle interference, but those features don't always work. Therefore, when capturing packets over a wireless network, you must pay close attention to your environment to ensure that there are no large sources of interference, such as big reflective surfaces, large rigid objects, microwaves, 2.4 GHz phones, thick walls, and high-density surfaces. These can cause packet loss, duplicated packets, and malformed packets.

Interference between channels is also a concern. Although you can sniff only one channel at a time, this comes with a small caveat: Several different transmission channels are available in the wireless networking spectrum, but because space is limited, there is a slight overlap between channels, as illustrated in Figure 11-2. This means that if there is traffic present on channel 4 and channel 5, and you are sniffing on one of these channels, you will likely capture packets from the other channel. Typically, networks that coexist in the same area are designed to use nonoverlapping channels of 1, 6, and 11, so you will probably not encounter this problem, but just in case, you should understand why it happens.

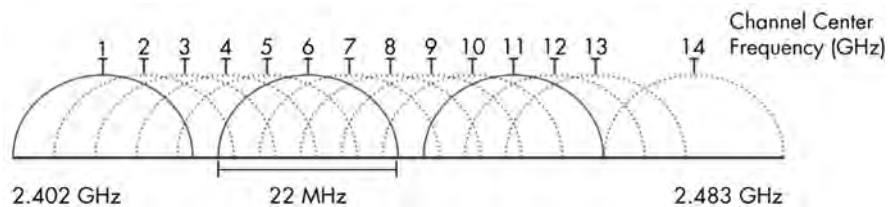


Figure 11-2: There is overlap between channels due to limited spectrum space.

Detecting and Analyzing Signal Interference

Troubleshooting wireless signal interference isn't something that can be done by looking at packets in Wireshark. If you are going to make a habit or a career out of troubleshooting WLANs, you will surely need to check for signal interference regularly. This task is done with a *spectrum analyzer*, which is a tool that displays data or interference across the spectrum.

Commercial spectrum analyzers can cost upward of thousands of dollars, but there is a great solution for common everyday use. MetaGeek makes a product called the Wi-Spy, which is a USB hardware device that monitors the entire 802.11 spectrum for interference. When paired with MetaGeek's Chanalyzer software, this hardware outputs the spectrum graphically to aid in the troubleshooting process. Sample output from Chanalyzer is shown in Figure 11-3.

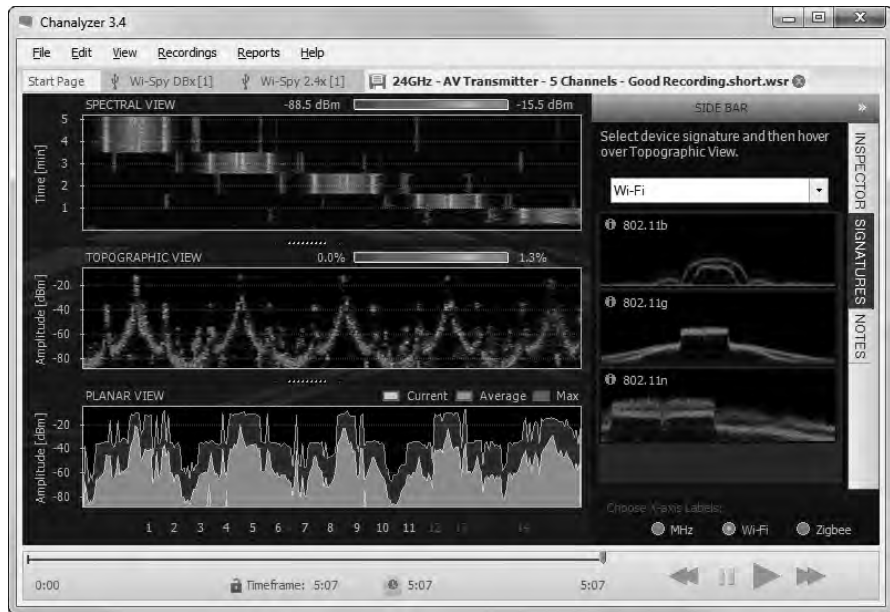


Figure 11-3: This Chanalyzer output shows several WLANs operating in the same area.

Wireless Card Modes

Before we start sniffing wireless packets, we need to look at the different modes in which a wireless card can operate as it pertains to packet capture.

Four wireless NIC modes are available:

Managed mode This mode is used when your wireless client connects directly to a wireless access point (WAP). In these cases, the driver associated with the wireless NIC relies on the WAP to manage the entire communication process.

Ad hoc mode This mode is used when you have a wireless network setup in which devices connect directly to each other. In this mode, two wireless clients that want to communicate with each other share the responsibilities that a WAP would normally handle.

Master mode Some higher-end wireless NICs also support master mode. This mode allows the wireless NIC to work in conjunction with specialized driver software in order to allow the computer to act as a WAP for other devices.

Monitor mode This is the most important mode for our purposes. Monitor mode is used when you want your wireless client to stop transmitting and receiving data, and listen only to the packets flying through the air. In order for Wireshark to capture wireless packets, your wireless NIC and accompanying driver must support monitor mode (also known as RFMON mode).

Most users use wireless cards in only managed mode or ad hoc mode. A graphical representation of the way each mode operates is shown in Figure 11-4.

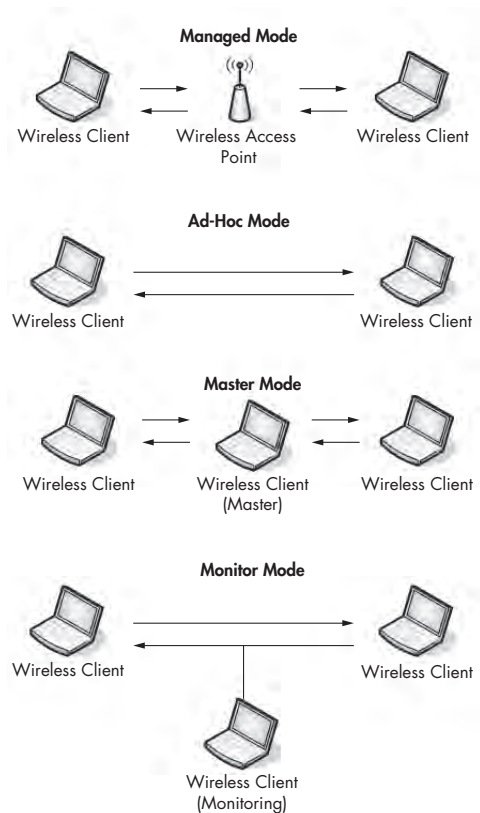


Figure 11-4: The different wireless card modes

NOTE I'm often asked which wireless card I recommend for wireless packet analysis. I use and highly recommend the ALFA 1000mW USB wireless adapter. It's highly regarded as one of the best on the market for ensuring you are capturing every possible packet. It is available through most online computer hardware retailers.

Sniffing Wirelessly in Windows

Even if you have a wireless NIC that supports monitor mode, most Windows-based wireless NIC drivers won't allow you to change into this mode (WinPcap doesn't support this either). You'll need a little extra hardware to get the job done.

Configuring AirPcap

AirPcap (from CACE Technologies, now a part of Riverbed, <http://www.cacetechnology.com/>) is designed to overcome the limitations that Windows places on wireless packet analysis. AirPcap is a small USB device that resembles a flash drive, as shown in Figure 11-5. It is designed to capture wireless traffic.

AirPcap uses the WinPcap driver discussed in Chapter 3 and a special client configuration utility.



Figure 11-5: The AirPcap device is very compact, making it easy to tote along with a laptop.

The AirPcap configuration program is simple to use, with only a few configurable options. The AirPcap Control Panel, shown in Figure 11-6, offers the following options:

Interface You can select the device you are using for your capture here. Some advanced analysis scenarios may require you to use more than one AirPcap device to sniff simultaneously on multiple channels.

Blink Led Clicking this button will make the LED lights on the AirPcap device blink. This is primarily used to identify the specific adapter you are using if you have multiple AirPcap devices.

Channel In this field, you select the channel you want AirPcap to listen on.

Include 802.11 FCS in Frames By default, some systems strip the last four checksum bits from wireless packets. This checksum, known as a frame check sequence (FCS), is used to ensure that packets have not been corrupted during transmission. Unless you have a specific reason to do otherwise, check this box to include the FCS checksums.

Capture Type The two options here are 802.11 Only and 802.11 + Radio. This 802.11 Only option includes the standard 802.11 packet header on all captured packets. The 802.11+ Radio option includes this header and also prepends it with a radiotap header, which contains additional information about the packet, such as data rate, frequency, signal level, and noise level. Choose 802.11 + Radio in order to see all available packet information.

FCS Filter Even if you uncheck the Include 802.11 FCS in Frames box, this option lets you filter out packets that FCS determines are corrupted. Use the Valid Frames option to show only those packets that FCS thinks can be received successfully.

WEP Configuration This area (accessible on the Keys tab of the AirPcap Control Panel) allows you to enter WEP decryption keys for the networks you will be sniffing. In order to be able to interpret data encrypted by WEP, you will need to enter the correct WEP keys into this field. WEP keys are discussed in “Wireless Security” on page 228.

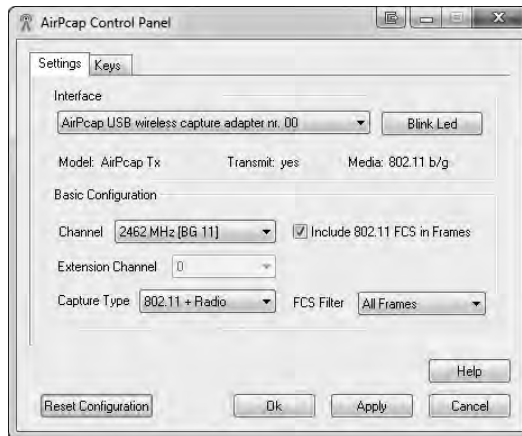


Figure 11-6: The AirPcap configuration program

Capturing Traffic with AirPcap

Once you have AirPcap installed and configured, the capture process should be familiar to you. Just start up Wireshark and select **Capture ► Options**. Next, select your AirPcap device in the **Interface** selection box ❶, as shown in Figure 11-7.

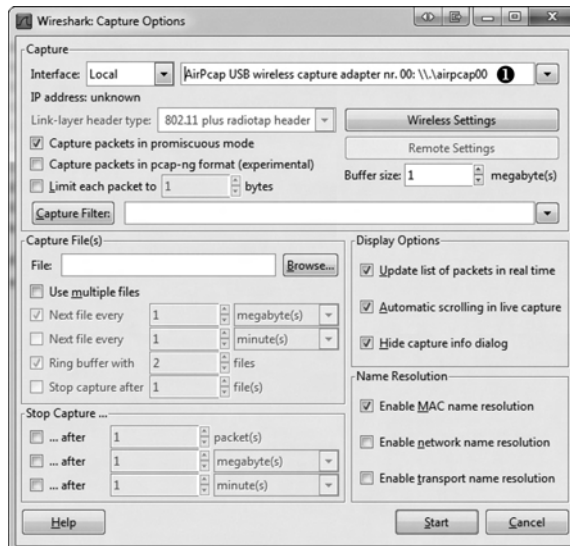


Figure 11-7: Choosing the AirPcap device as your capture interface

Everything on this screen should look familiar except for the Wireless Settings button. Clicking this button will give you the same options that the AirPcap utility gave you, as shown in Figure 11-8. Because Wireshark is completely integrated with AirPcap, anything configured in the client utility can also be configured from within Wireshark.



Figure 11-8: The Advanced Wireless Settings dialog allows you to configure AirPcap from within Wireshark.

Once you have everything configured to your liking, begin capturing packets by clicking the **Start** button.

Sniffing Wirelessly in Linux

Sniffing in Linux is simply a matter of enabling monitor mode on the wireless NIC and firing up Wireshark. Unfortunately, the procedure for enabling monitor mode differs with each model of wireless NIC, so I can't offer a definitive guide for that here. In fact, some wireless NICs don't require you to enable monitor mode. Your best bet is to do a quick Google search for your NIC model to determine how to enable it and if you need to do so.

One of the more common ways to enable monitor mode in Linux is through its built-in wireless extensions. You can access these wireless extensions with the `iwconfig` command. If you type `iwconfig` from the console, you should see results like this:

```
$ iwconfig
Eth0  no wireless extensions
Lo0   no wireless extensions
Eth1  IEEE 802.11gESSID: "Tesla Wireless Network"
      Mode: Managed Frequency: 2.462 GHz Access Point: 00:02:2D:8B:70:2E
      Bit Rate: 54 Mb/s Tx-Power=20 dBm Sensitivity=8/0
      Retry Limit: 7 RTS thr: off Fragment thr: off
      Power Management: off
      Link Quality=75/100 Signal level=-71 dBm Noise level=-86 dBm
      Rx invalid nwid: 0 Rx invalid crypt: 0 Rx invalid frag: 0
      Tx excessive retries: 0 Invalid misc: 0 Missed beacon: 2
```

The output from the `iwconfig` command shows that the `Eth1` interface can be configured wirelessly. This is apparent because it shows data for the 802.11g protocol, whereas the interfaces `Eth0` and `Lo0` return the phrase no wireless extensions.

Along with all of the wireless information this command provides, such as the wireless extended service set ID (ESSID) and frequency, notice that the second line under Eth1 shows that the mode is currently set to managed. This is what we want to change.

In order to change the Eth1 interface to monitor mode, you must be logged in as the root user, either directly or via the switch user (su) command, as shown here.

```
$ su
Password: <enter root password here>
```

Once you're root, you can type commands to configure the wireless interface options. To configure Eth1 to operate in monitor mode, type this:

```
# iwconfig eth1 mode monitor
```

Once the NIC is in monitor mode, running the iwconfig command again should reflect your changes. Now ensure that the Eth1 interface is operational by typing the following:

```
# iwconfig eth1 up
```

We'll also use the iwconfig command to change the channel we are listening on. Change the channel of the Eth1 interface to channel 3 by typing this:

```
# iwconfig eth1 channel 3
```

NOTE *You can change channels on the fly as you are capturing packets, so don't hesitate to do this at will. This iwconfig command can also be scripted to make the process easier.*

When you have completed these configurations, start Wireshark and begin your packet capture.

802.11 Packet Structure

80211beacon.pcap

The primary difference between wireless and wired packets is the addition of the 802.11 header. This is a layer 2 header that contains extra information about the packet and the medium on which it is transmitted. There are three types of 802.11 packets:

Management These packets are used to establish connectivity between hosts at layer 2. Some important subtypes of management packets include authentication, association, and beacon packets.

Control Control packets allow for delivery of management and data packets and are concerned with congestion management. Common subtypes include request-to-send and clear-to-send packets.

Data These packets contain actual data and are the only packet type that can be forwarded from the wireless network to the wired network.

The type and subtype of a wireless packet determines its structure, so there are a large number of possible structures. We will examine one such structure by looking at a single packet in the file *80211beacon.pcap*. This file contains an example of a management packet called a *beacon*, as shown in Figure 11-9.

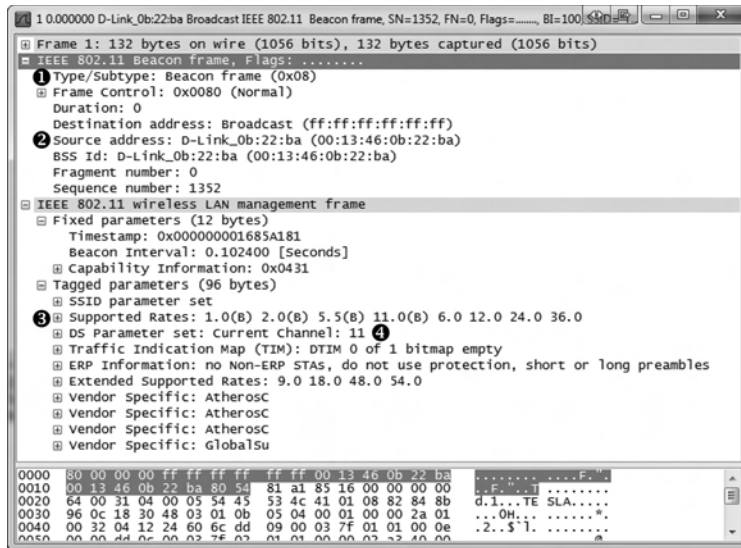


Figure 11-9: This is an 802.11 beacon packet.

A beacon is one of the most informative wireless packets you can find. It is sent as a broadcast packet from a WAP across a wireless channel to notify any listening wireless clients that the WAP is available and to define the parameters that must be set in order to connect to it. In our example file, you can see that this packet is defined as a beacon in the Type/Subtype field in the 802.11 header ❶.

A great deal of additional information is found in the 802.11 management frame header, including the following:

Timestamp The time the packet was transmitted.

Beacon Interval The interval at which the beacon packet is retransmitted.

Capability Information Information about the hardware capabilities of the WAP.

SSID Parameter Set The SSID (network name) broadcast by the WAP.

Supported Rates The data transfer rates supported by the WAP.

DS Parameter The channel on which the WAP is broadcasting.

The header also includes the source and destination addresses and vendor-specific information.

Based on this knowledge, we can determine quite a few things about the WAP transmitting the beacon in the example file. It is apparent that it is a D-Link device ❷ using the 802.11b standard (B) ❸ on channel 11 ❹.

Although the exact contents and purpose of 802.11 management packets will change, the general structure remains similar to this example.

Adding Wireless-Specific Columns to the Packet List Pane

As you've seen, Wireshark typically shows six individual columns in the Packet List pane. Before we proceed with any additional wireless analysis, it will be helpful to add three new columns to the Packet List pane:

- The RSSI (for Received Signal Strength Indication) column, to show the radio frequency (RF) signal strength of a captured packet
- TX Rate (for Transmission Rate) column, to show the data rate of a captured packet
- The Frequency/Channel column, to show the frequency and channel on which the packet was collected

These indicators can be of great help when troubleshooting wireless connections. For instance, even if your wireless client software says you have excellent signal strength, doing a capture and checking these columns may show you a number that does not support this claim.

To add these columns to the Packet List pane, follow these steps:

1. Choose **Edit ▶ Preferences**.
2. Navigate to the **Columns** section and click **New**.
3. Type **RSSI** in the **Title** field and select **IEEE 802.11 RSSI** in the **Field type** drop-down list.
4. Repeat this process for the TX Rate and Frequency/Channel columns, titling them appropriately and selecting **IEEE 802.11 TX Rate** and **Channel/Frequency** in the **Field type** drop-down list. Figure 11-10 shows what the Preferences window should look like after you have added all three columns.

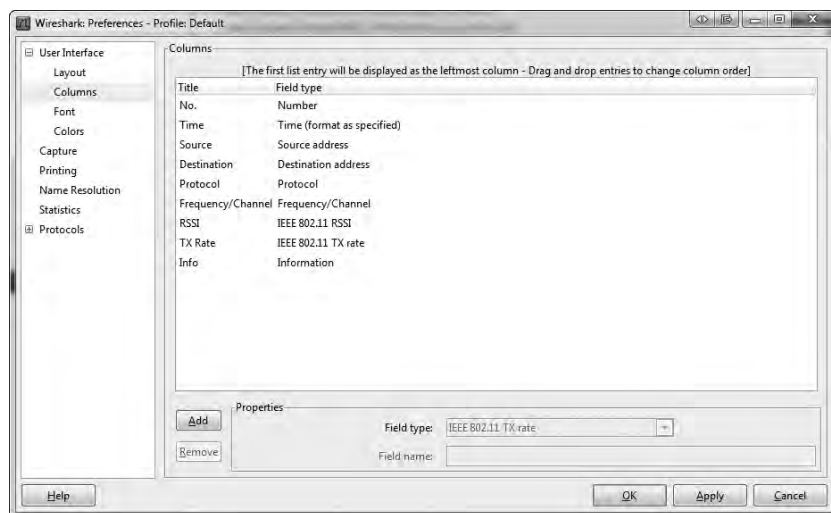


Figure 11-10: Adding the IEEE wireless-specific columns in the Packet List pane

5. Click **OK** to save your changes.
6. Restart Wireshark to display the new columns.

Wireless-Specific Filters

We discussed the benefits of capture and display filters in Chapter 4. Filtering traffic in a wired infrastructure is a lot easier, since each device has its own dedicated cable. In a wireless network, however, all traffic generated by wireless clients coexists on shared channels, which means that a capture of any one channel may contain traffic from dozens of clients. This section is devoted to some packet filters that can be used to help you find specific traffic.

Filtering Traffic for a Specific BSS ID

Each WAP in a network has a unique identifying name called its *basic service set identifier* (BSS ID). This name is sent in every wireless management packet and data packet the access point transmits.

Once you know the name of the BSS ID you want to examine, all you really need to do is find a packet that has been sent from that particular WAP. Wireshark shows the transmitting WAP in the Info column of the Packet List pane, so finding this information is typically pretty easy.

Once you have a packet from the WAP of interest, find its BSS ID field in the 802.11 header. This is the address on which you will base your filter. After you have found the BSS ID MAC address, you can use this filter:

```
wlan.bssid.eq 00:11:22:33:44:55:66
```

And you will see only the traffic flowing through the specified WAP.

Filtering Specific Wireless Packet Types

Earlier in this chapter, we discussed the different types of wireless packets you might see on a network. You will often need to filter based on these types and subtypes. This can be done with the filters *wlan.fc.type* for specific types, and *wc.fc.type_subtype* for specific type or subtype combinations. For instance, to filter for a NULL data packet (a Type 2 Subtype 4 packet in hex), you could use the filter *wlan.fc.type_subtype eq 0x24*. Table 11-1 provides a quick reference to some common filters you might need when filtering on 802.11 packet types and subtypes.

Table 11-1: Wireless Types/Subtypes and Associated Filter Syntax

Frame Type/Subtype	Filter Syntax
Management frame	wlan.fc.type eq 0
Control frame	wlan.fc.type eq 1
Data frame	wlan.fc.type eq 2
Association request	wlan.fc.type_subtype eq 0x00
Association response	wlan.fc.type_subtype eq 0x01
Reassociation request	wlan.fc.type_subtype eq 0x02
Reassociation response	wlan.fc.type_subtype eq 0x03
Probe request	wlan.fc.type_subtype eq 0x04
Probe response	wlan.fc.type_subtype eq 0x05
Beacon	wlan.fc.type_subtype eq 0x08
Disassociate	wlan.fc.type_subtype eq 0x0A
Authentication	wlan.fc.type_subtype eq 0x0B
Deauthentication	wlan.fc.type_subtype eq 0x0C
Action frame	wlan.fc.type_subtype eq 0x0D
Block ACK requests	wlan.fc.type_subtype eq 0x18
Block ACK	wlan.fc.type_subtype eq 0x19
Power save poll	wlan.fc.type_subtype eq 0x1A
Request to send	wlan.fc.type_subtype eq 0x1B
Clear to send	wlan.fc.type_subtype eq 0x1C
ACK	wlan.fc.type_subtype eq 0x1D
Contention free period end	wlan.fc.type_subtype eq 0x1E
NULL data	wlan.fc.type_subtype eq 0x24
QoS data	wlan.fc.type_subtype eq 0x28
Null QoS data	wlan.fc.type_subtype eq 0x2C

Filtering a Specific Frequency

If you are examining a compilation of traffic that includes packets from multiple channels, it can be very useful to filter based on each individual channel. For instance, if you are expecting to have traffic present on only channels 1 and 6, you can input a filter to show all channel 11 traffic. If you

find any traffic, then you will know that something is wrong—perhaps a misconfiguration or a rogue device. In order to filter on a specific frequency, use this filter syntax:

```
radiotap.channel.freq == 2412
```

This will show all traffic on channel 1. You can replace the 2412 value with the appropriate frequency for the channel you wish to filter. Table 11-2 lists the frequencies associated with each channel.

Table 11-2: 802.11 Wireless Channels and Frequencies

Channel	Frequency
1	2412
2	2417
3	2422
4	2427
5	2432
6	2437
7	2442
8	2447
9	2452
10	2457
11	2462

There are hundreds of additional useful filters that you can use for wireless network traffic. You can view additional wireless capture filters on the Wireshark wiki at <http://wiki.wireshark.org/>.

Wireless Security

The biggest concern when deploying and administering a wireless network is the security of the data transmitted across it. With data flying through the air, free for the taking by anyone who knows how, it's crucial that data be encrypted. Otherwise, anyone with Wireshark and an AirPcap card can see it.

NOTE *When another layer of encryption, such as SSL or SSH, is used, traffic will still be encrypted at that layer, and the user's communication will still be unreadable by a person with a packet sniffer.*

The original preferred method for securing data transmitted over wireless networks was in accordance with the Wired Equivalent Privacy (WEP) standard. WEP was mildly successful for years until several weaknesses were uncovered in its encryption key management. To improve security, new standards were created. These include the Wi-Fi Protected Access (WPA) and WPA2 standards. Although WPA and its more secure revision WPA2 are still fallible, they are considered more secure than WEP.

In this section, we will look at some WEP and WPA traffic, along with examples of failed authentication attempts.

Successful WEP Authentication

80211-WEPauth.
.pcap

The file *80211-WEPauth.pcap* contains an example of a successful connection to a WEP-enabled wireless network. The security on this network is set up using a WEP key. This is a key you must provide to the WAP in order to authenticate to it and decrypt data sent from it. You can think of this WEP key as a wireless network password.

As shown in Figure 11-11, the capture file begins with a challenge from the WAP (00:11:88:6b:68:30) to the wireless client (00:14:a5:30:b0:af) in packet 4 ❶. The purpose of this challenge is to determine if the wireless client has the correct WEP key. You can see this challenge by expanding the 802.11 header and its tagged parameters.

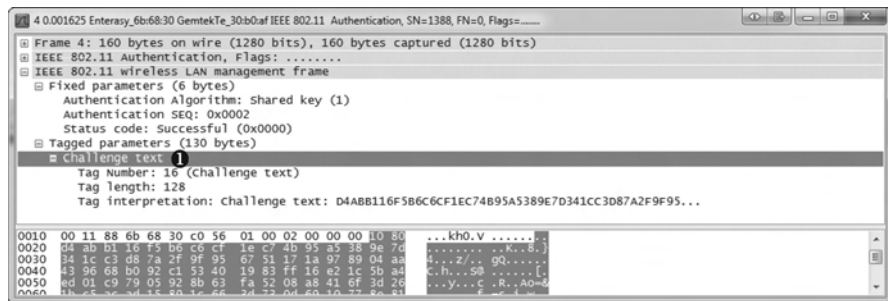


Figure 11-11: The WAP issues challenge text to the wireless client.

The challenge is acknowledged with packet 5. The wireless client then responds by decrypting the challenge text with the WEP key and returning it to the WAP ❷, as shown in Figure 11-12.

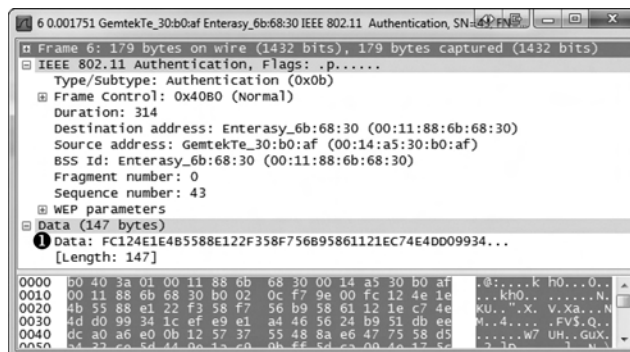


Figure 11-12: The wireless client sends the unencrypted challenge text back to the WAP.

The packet is once again acknowledged in packet 7, and the WAP responds to the wireless client in packet 8, as shown in Figure 11-13. The response contains notification that the authentication process was successful ❸.

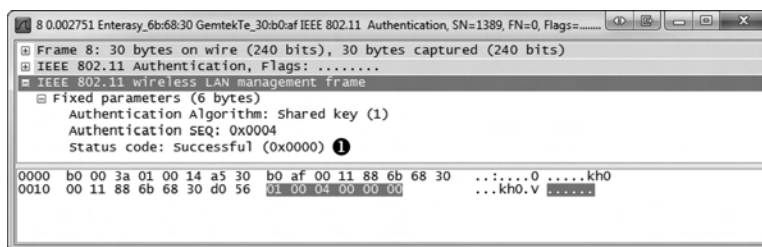


Figure 11-13: The WAP alerts the client that authentication was successful.

Finally, after successful authentication, the client can transmit an association request, receive an acknowledgment, and complete the connection process, as shown in Figure 11-14.

No.	Time	Source	Destination	Protocol	Channel	Info
10	0.000876	GemtekTe_30:b0:af	Enterasy_6b:68:30	IEEE 802.11		Association Request, SN=44, FN=0, Flags=....., SSID="DENVERDEICE"
11	0.000374			IEEE 802.11		Acknowledgement, Flags=.....
12	0.002627	Enterasy_6b:67:28	broadcast	IEEE 802.11		Data, SN=1390, FN=0, Flags=p....F
13	0.000624	Enterasy_6b:68:30	GemtekTe_30:b0:af	IEEE 802.11		Association response, SN=1391, FN=0, Flags=.....
14	0.000374			IEEE 802.11		Acknowledgement, Flags=.....
15	0.683813	GemtekTe_30:b0:af	Enterasy_6b:68:30	IEEE 802.11		Null function (No data), SN=45, FN=0, Flags=....., T
16	0.000098			IEEE 802.11		Acknowledgement, Flags=.....
17	0.000053	GemtekTe_30:b0:af	broadcast	IEEE 802.11		Data, SN=46, FN=0, Flags=p...., T

Figure 11-14: The authentication process is followed by a simple two-packet association request and response.

Failed WEP Authentication

80211-
WEPauthfail.pcap

In our next example, a user enters his WEP key to connect to a WAP, and after several seconds, the wireless client utility reports that it was unable to connect to the wireless network but fails to tell why. The resulting file is *80211-WEPauthfail.pcap*.

As with the successful attempt, this communication begins with the WAP sending challenge text to the wireless client in packet 3. This is acknowledged, and in packet 5, the wireless client sends its response using the WEP key provided by the user.

At this point, we would expect to see notification that the authentication was successful, but we see something different in packet 7, as shown in Figure 11-15 ❶.

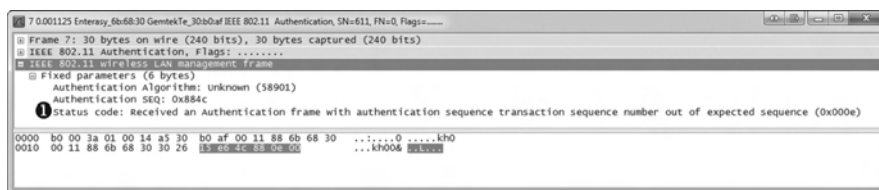


Figure 11-15: This message tells us the authentication was unsuccessful.

This message tells us that the wireless client's response to the challenge text was incorrect. This suggests that the WEP key the client used to decrypt the challenge text must have been incorrect. As a result, the connection process has failed. It must be reattempted with the proper WEP key.

Successful WPA Authentication

WPA uses a very different authentication mechanism than WEP, but it still relies on the user to enter a key into the wireless client in order to connect to the network. An example of a successful WPA authentication is found in the file *80211-WPAauth.pcap*.

The first packet in this file is a beacon broadcast from the WAP. Let's expand the 802.11 header of this packet, look under tagged parameters, and expand the Vendor Specific heading, as shown in Figure 11-16. You should see a section devoted to the WPA attributes of the WAP ❶. This lets us know that the WAP supports WPA and the version and implementation it supports.

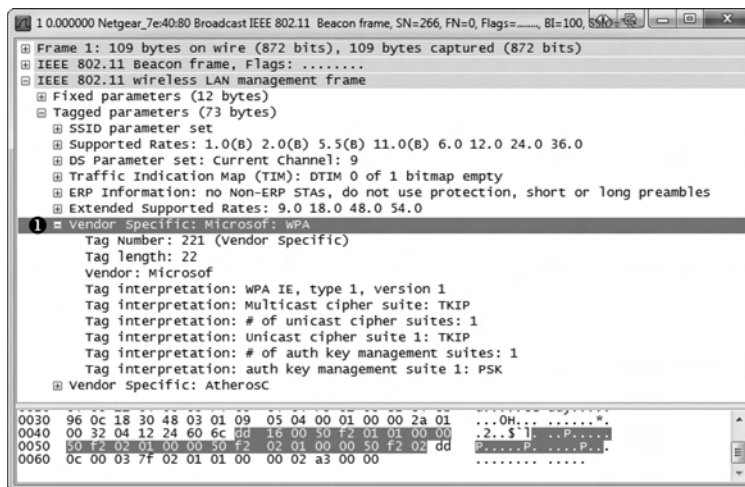


Figure 11-16: This beacon lets us know that the WAP supports WPA authentication.

Once the beacon is received, the wireless client (00:14:6c:7e:40:80) sends a probe request for the WAP (00:0f:b5:88:ac:82), and the WAP responds. Authentication and association requests and responses are generated between the wireless client and WAP in packets 4 through 7.

Things really start to pick up in packet 8. This is where the WPA handshake begins, continuing through packet 11. This handshake process is where the WPA challenge response takes place, as shown in Figure 11-17.

No.	Time	Source	Destination	Protocol	Channel	Info
8	0.004096	Netgear_7e:40:80	Netgear_88:ac:82	EAPOL		Key
9	0.004101	Netgear_88:ac:82	Netgear_7e:40:80	EAPOL		Key
10	0.003580	Netgear_7e:40:80	Netgear_88:ac:82	EAPOL		Key
11	0.000004	Netgear_88:ac:82	Netgear_7e:40:80	EAPOL		Key

Figure 11-17: These packets are a part of the WPA handshake.

There are two challenges and responses. Each can be matched with the other based on the Replay Counter field under the 802.1x Authentication header, as shown in Figure 11-18. Notice that the Replay Counter value for the first two handshake packets is 1 ❶, and for the second two handshake packets, it's 2 ❷.

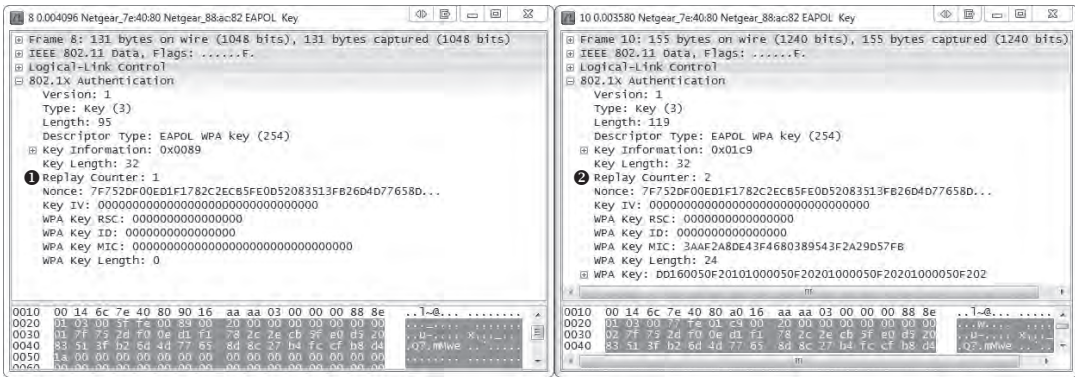


Figure 11-18: The Replay Counter field helps us pair challenges and responses.

After the WPA handshake is completed and authentication is successful, data begins transferring between the wireless client and the WAP.

Failed WPA Authentication

80211-
WPAauthfail
.pcap

As with WEP, we'll look at what happens when a user enters his WPA key and the wireless client utility reports that it was unable to connect to the wireless network, without indicating the problem. The resulting file is *80211-WPAauthfail.pcap*.

Once again, the capture file begins in a manner identical to the successful WPA authentication we just examined. This includes probe, authentication, and association requests. The WPA handshake begins in packet 8, but in this case, we can see that there are eight handshake packets instead of the four we saw in the successful authentication attempt.

Packets 8 and 9 represent the first two packets seen in the WPA handshake. In this case, however, the challenge text that is sent back to the WAP from the client is incorrect. As a result, the sequence is repeated in packets 10 and 11, 12 and 13, and 14 and 15, as shown in Figure 11-19. Each request and response can be paired using the Replay Counter value.

No.	Time	Source	Destination	Protocol	Channel	Info
9	0.003547	Netgear_88:ac:82	Netgear_7e:40:80	EAPOL		Key
10	1.000549	Netgear_7e:40:80	Netgear_88:ac:82	EAPOL		Key
11	0.000476	Netgear_88:ac:82	Netgear_7e:40:80	EAPOL		Key
12	0.999489	Netgear_7e:40:80	Netgear_88:ac:82	EAPOL		Key
13	0.000511	Netgear_88:ac:82	Netgear_7e:40:80	EAPOL		Key
14	0.999013	Netgear_7e:40:80	Netgear_88:ac:82	EAPOL		Key
15	-0.000037	Netgear_88:ac:82	Netgear_7e:40:80	EAPOL		Key

Figure 11-19: The additional EAPOL packets here indicate the failed WPA authentication.

Once the handshake process has been attempted four times, the communication is aborted. As shown in Figure 11-20, the wireless client deauthenticates from the WAP in packet 16 ❶.

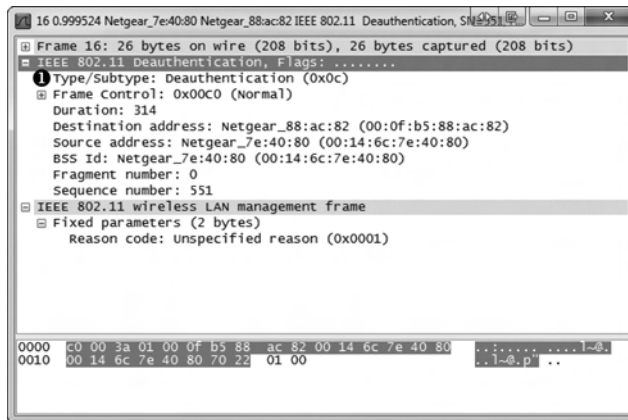


Figure 11-20: After failing the WPA handshake, the client deauthenticates.

Final Thoughts

Although wireless networks are still considered widely insecure, that hasn't slowed their deployment in various organizational environments. As the focus shifts to communication without wires, it is critical to be able to capture and analyze data on wireless networks, as well as wired ones. The skills and concepts taught in this chapter are by no means exhaustive, but they should provide a jump start in helping you understand the intricacies of troubleshooting wireless networks with packet analysis.



FURTHER READING



Although the tool used primarily in this book is Wireshark, a great deal of additional tools will come in handy when you're performing packet analysis—whether it be for general troubleshooting, slow networks, security issues, or wireless networks. This chapter lists some useful packet analysis tools and other packet analysis learning resources.

Packet Analysis Tools

There are several tools that are useful for packet analysis in addition to Wireshark. Here, we'll look at a few of the ones I have found most useful.

tcpdump and Windump

Although Wireshark is very popular, it is probably less widely used than tcpdump. Considered the de facto packet capture and analysis utility by several crowds, tcpdump is entirely text based.

Although tcpdump lacks graphical features, it is great for sifting through large amounts of data, as you can pipe its output to other commands, such as sed and awk in Linux. As you delve further into packet analysis, you will find use for both Wireshark and tcpdump. You can download tcpdump from <http://www.tcpdump.org/>.

Windump is simply a distribution of tcpdump that has been remade for Windows. You can download it from <http://www.winpcap.org/windump/>.

Cain & Abel

Discussed in Chapter 2, Cain & Abel is one of the better Windows tools for ARP cache poisoning. Cain & Abel is actually a very robust suite of tools, and you will surely be able to find other uses for it as well. It is available from <http://www.oxid.it/cain.html>.

Scapy

Scapy is a very powerful Python library that allows for the creation and manipulation of packets based on command-line scripts within its environment. Simply put, Scapy is the most powerful and flexible packet-crafting application available. You can read more about Scapy, download it, and view sample Scapy scripts at <http://www.secdev.org/projects/scapy/>.

Netdude

If you don't need something as advanced as Scapy, then Netdude is a great Linux alternative. Although Netdude is limited in its ability, it provides a GUI that is very easy to use for creating and modifying packets for research purposes. Figure A-1 shows an example of using Netdude. You can download Netdude from <http://netdude.sourceforge.net/>.

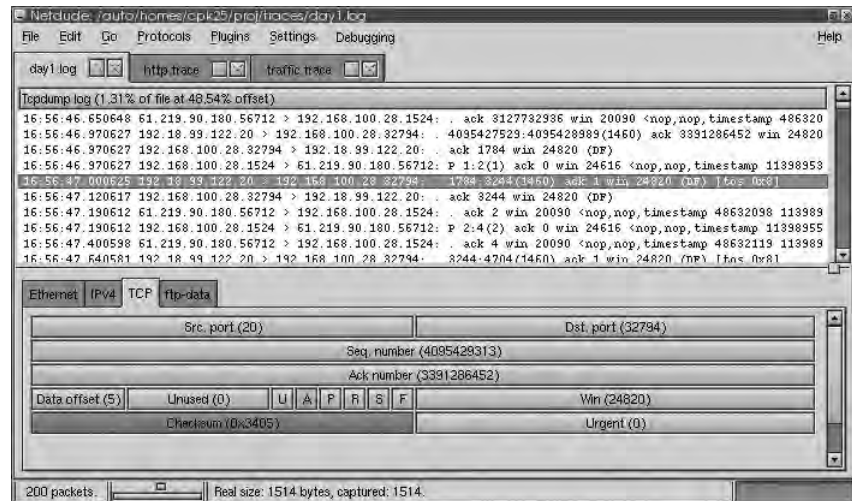


Figure A-1: Modifying packets within Netdude

Colasoft Packet Builder

If you are a Windows user and want a GUI similar to Netdude, then consider using Colasoft Packet Builder, an excellent free tool. Colasoft also provides an easy-to-use GUI for packet creation and modification. You can download it from http://www.colasoft.com/packet_builder/.

CloudShark

CloudShark (developed by QA Café) is one of my favorite online resources for sharing packet captures with others. CloudShark is a website that displays network capture files inside your browser in a Wireshark-esque manner, as shown in Figure A-2. You can upload capture files and send the links to colleagues for shared analysis.

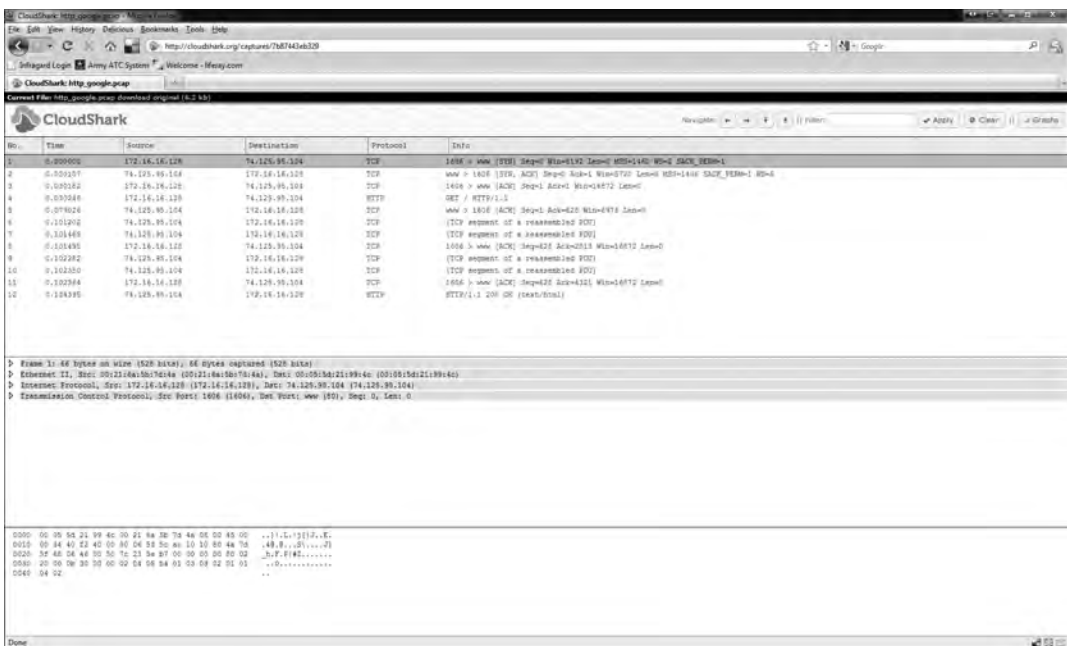


Figure A-2: A sample capture file viewed with CloudShark

My favorite thing about CloudShark is that it doesn't require registration and accepts direct linking via URL. This means that when I post a link to a PCAP file on my blog, someone can just click it and see the packets, without needing to download the file and open it in Wireshark.

CloudShark is accessible at <http://www.cloudshark.org/>.

pcapr

pcapr is a very robust Web 2.0 platform for sharing PCAP files created by the folks at Mu Dynamics. As of this writing, pcapr contains nearly 3,000 PCAP files, with examples of more than 400 different protocols. Figure A-3 shows an example of a DHCP traffic capture on pcapr.

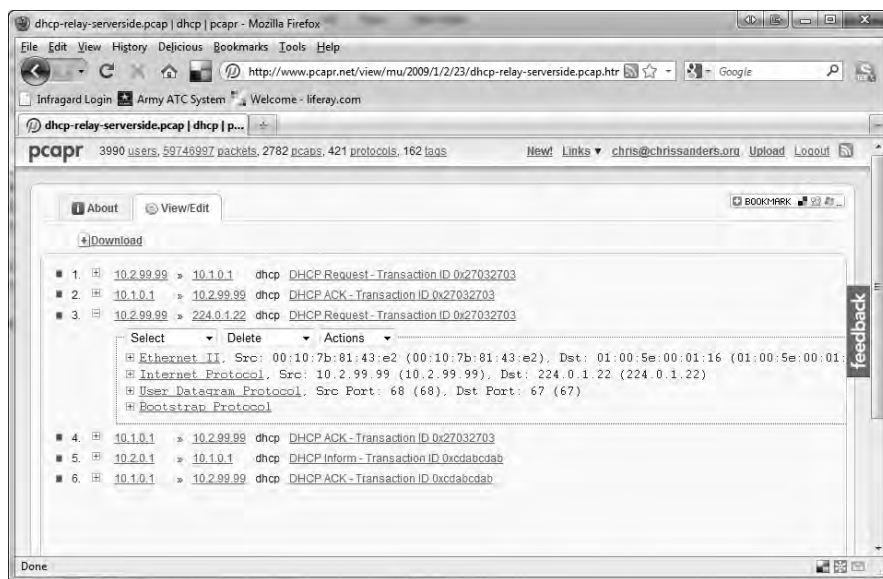


Figure A-3: Viewing a DHCP traffic capture on pcapr

When I'm looking for an example of a certain type of communication, I start by searching on pcapr. If you find yourself creating a lot of different capture files in your own experimentation, don't hesitate to share them with the community by uploading them to pcapr, at <http://www.pcap.net/>.

NetworkMiner

NetworkMiner is a tool primarily used for network forensics, but I've found it useful in a variety of other situations as well. Although it can be used to capture packets, its real strength is how it parses PCAP files. NetworkMiner will take a PCAP file and break it down into the operating systems detected and the sessions between hosts. It even allows you to extract transferred files directly from the capture. NetworkMiner is free to download from <http://networkminer.sourceforge.net/>.

Tcpreplay

Whenever I have a set of packets that I need to retransmit over the wire to see how a device reacts to them, I use Tcpreplay to perform that. Tcpreplay is designed specifically to take a PCAP file and retransmit the packets contained within it. Download it from <http://tcpreplay.synfin.net/>.

ngrep

If you are familiar with Linux, you've no doubt used grep to search through data. ngrep is very similar and allows you to perform very specific searches through PCAP data. I mostly use ngrep when capture and display filters won't do the job or get too wildly complex. You can read more about ngrep at <http://ngrep.sourceforge.net/>.

libpcap

If you plan to do any really advanced packet parsing or create applications that deal with packets, you become very familiar with libpcap. Simply put, libpcap is a portable C/C++ library for network traffic capture. Wireshark, tcpdump, and most other packet analysis applications rely on the libpcap library at some level. You can read more about libpcap at <http://www.tcpdump.org/>.

hping

hping is one of the more versatile tools to have in your arsenal. hping is a command-line packet crafting and transmission tool. It supports a variety of protocols and is very quick and intuitive to use. You can download hping from <http://www.hping.org/>.

Domain Dossier

If you need to look up the registration information for a domain or IP address, then Domain Dossier is the place to do that. It's fast, it's simple, and it works. You can access Domain Dossier at <http://www.centralops.net/co/DomainDossier.aspx>.

Perl and Python

Perl and Python aren't tools but rather scripting languages that are well worth mentioning. As you become proficient in packet analysis, you will encounter cases where no automated tool exists to meet your needs. In those cases, Perl and Python are the languages of choice for making tools that can do interesting things with packets. I typically use Python for most applications, but it's often just a matter of personal preference.

Packet Analysis Resources

From Wireshark's home page to courses and blogs, many resources for packet analysis are available. I'll list a few of my favorites here.

Wireshark Home Page

The foremost resource for everything related to Wireshark is its home page, at <http://www.wireshark.org/>. The home page includes the software documentation, a very helpful wiki that contains sample capture files, and sign-up information for the Wireshark mailing list.

SANS Security Intrusion Detection In-Depth Course

As a SANS mentor, I'm slightly biased, but I don't think there is a better packet analysis course on the planet than SANS SEC 503, Intrusion Detection In-Depth. This class focuses on the security aspects of packet analysis. Even if you aren't focused on security, the first two days of the course provide the best introduction to packet analysis and tcpdump that I've ever experienced.

The course is taught by two of my packet analysis heroes, Mike Poor and Judy Novak. It is offered at live events several times throughout the year. If your travel budget is limited, the course is also taught through an online and web-based on-demand format.

You can read more about SEC 503 and other SANS courses at <http://www.sans.org/>.

Chris Sanders Blog

I don't get around to posting nearly enough, but I do occasionally write articles related to packet analysis and post them on my blog, at <http://www.chrissanders.org/>. If nothing else, my blog serves as a portal that links to other articles and books I have written, and it provides information about how to get in touch with me.

Packetstan Blog

The blog of Mike Poor and Judy Novak is my favorite packet-related blog out there at the moment. Their site <http://www.packetstan.com/> contains some great breakdowns of interesting traffic, and every single piece of content on it is A+ material. Mike and Judy are two of the best at what they do, and they are a large inspiration to me.

Wireshark University

Laura Chappell is one of the most prolific Wireshark evangelists you will find. Her site contains loads of Wireshark tips, as well as information about her book, *Wireshark Network Analysis*, and the courses she teaches. Find out more at <http://www.wiresharktraining.com/>.

IANA

The Internet Assigned Numbers Authority (IANA), available at <http://www.iana.org/>, oversees the allocation of IP addresses and protocol number assignments for North America. Its website offers some valuable reference tools, such as the ability to look up port numbers, view information related to top-level domain names, and browse companion sites to find and view RFCs.

TCP/IP Illustrated (Addison-Wesley)

Considered the TCP/IP bible by most, this series of books by Dr. Richard Stevens is a staple on the bookshelves of most who live at the packet level. It is my favorite TCP/IP book and something I referenced quite a bit when writing this book.

The TCP/IP Guide (No Starch Press)

One more favorite of mine in the TCP/IP realm is this book by Charles Kozierok. Weighing in at over 1,000 pages, it's very detailed and contains a lot of great diagrams for the visual learner.

INDEX

Symbols & Numbers

- && (AND) operator, in BPF syntax, 58
- <iframe> tag (HTML), 200
- <script> tag (HTML), 198–199
- tag (HTML), 200
- == (equal-to) comparison operator, 64
- ! (NOT) operator, in BPF syntax, 58
- .pcap file format, 48. *See also* capture file examples
- || (OR) operator, in BPF syntax, 58, 61
- 802.11 standard, 216
 - packet structure, 223–225

A

- ACKed Lost Packet message, 84
- ACK packet, 102, 132, 167, 168
 - duplicate, 171–172, 179
- acknowledgment number, in ACK packet, 169–170
- acknowledgment packet in DHCP, 119
- active fingerprinting, 196–197
- address registries, 70
- Address Resolution Protocol (ARP), 18, 86–90
 - gratuitous ARP, 89–90
 - header, 87–88
 - packet structure, 88
 - packets, 204
 - reply, 86, 87, 89, 148

- request, 86, 87, 88–89, 145, 148
- spoofing, 27, 205
- unsolicited updates to table, 204
- addressing filters, 59
- ad hoc mode, for wireless NIC, 218, 219
- Advanced Wireless Settings dialog, 221–222
- AfriNIC (Africa), 70
- aggregated network tap, 24–25
- AirPcap
 - capturing traffic, 221–222
 - configuring, 219–220
 - Control Panel, 220–221
- AJAX (Asynchronous JavaScript and XML), 139
- alerts from IDS, 206
- ALFA 1000mW USB wireless adapter, 219
- American Registry for Internet Numbers (ARIN), 70
- analysis step, in sniffer process, 4
- Analyze menu
 - Display Filters, 65
 - Expert Info Composite, 83
- AND (&&) operator, in BPF syntax, 58
- and filter expression logical operator, 65
- APNIC (Asia/Pacific), 70
- application baseline, 185
- Application layer (OSI), 5
- archive file, extracting, 39
- ARIN (American Registry for Internet Numbers), 70

- ARP. *See* Address Resolution Protocol (ARP)
- ARP cache poisoning, 26–30, 32
 - attacker use, 202–205
 - caution on, 30
- associations/dependencies
 - in application baseline, 186
 - in host baseline, 185
- asymmetric routing, 30
- Asynchronous JavaScript and XML (AJAX), 139
- attackers
 - exploitation, 197–213
 - ping use to determine host
 - accessibility, 108
 - and random text in ICMP echo request, 110
 - reconnaissance by potential, 190–197
- Windows command shell
 - use, 201
- Aurora exploit, 197–202
- authentication
 - in host baseline, 185
 - in site baseline, 184
 - Twitter vs. Facebook, 140
- WEP
 - failed, 230
 - successful, 229–230
- WPA
 - failed, 232–233
 - successful, 231–232
- Automatic Scrolling in Live Capture
 - option, 56
- AXFR (full zone transfer), 127

B

- baseline for network, 41, 183–187
 - application baseline, 186
 - host baseline, 185
 - site baseline, 184
- basic service set identifier (BSS ID), 226
- beacon packet, 224–225
 - broadcast from WAP, 231
- benchmarking, Protocol Hierarchy
 - Statistics for, 71–72

- Berkeley Packet Filter (BPF) syntax, 58–61
- Bootstrap Protocol (BOOTP), 113, 116
- bottleneck, analyzer as, 30
- BPF (Berkeley Packet Filter) syntax, 58–61
- branch office, troubleshooting
 - connections, 155–159
- broadcast address, 116
- broadcast domain, 14–15, 18
- broadcast packet, 14–15
- broadcast traffic, in site
 - baseline, 184
- BSS ID (basic service set identifier), 226
- buffer space in TCP, 173
- byte offset, for protocol field
 - filters, 60

C

- CACE Technologies, 219
- Cain & Abel, 27–29, 236
- CAM (Content Addressable Memory) table, 12, 86
- CAPSCREEN command, 208–209
- capture file examples
 - 80211beacon.pcap*, 224
 - 80211-WEPauthfail.pcap*, 230
 - 80211-WEPauth.pcap*, 229
 - 80211-WPAauthfail.pcap*, 232
 - 80211-WPAauth.pcap*, 231
 - activeosfingerprinting.pcap*, 197
 - arp_gratuitous.pcap*, 90
 - arppoison.pcap*, 202
 - arp_resolution.pcap*, 88
 - aurora.pcap*, 197
 - dhcp_inlease_renewal.pcap*, 120
 - dhcp_nolease_renewal.pcap*, 115
 - dns_axfr.pcap*, 127
 - dns_query_response.pcap*, 122
 - dns_recursivequery_client.pcap*, 124
 - dns_recursivequery_server.pcap*, 125
 - download-fast.pcap*, 79, 81
 - download-slow.pcap*, 78, 80, 83
 - facebook_login.pcap*, 138
 - facebook_message.pcap*, 139

- http_espn.pcap*, 140
- http_google.pcap*, 77, 82, 129
- http_post.pcap*, 131
- icmp_echo.pcap*, 108
- inconsistent_printer.pcap*, 153
- ip_frag_source.pcap*, 95, 96
- ip_ttl_dest.pcap*, 95
- ip_ttl_source.pcap*, 94
- latency1.pcap*, 180
- latency2.pcap*, 180
- latency3.pcap*, 181
- latency4.pcap*, 182
- lotsofweb.pcap*, 70, 71
- nowebaccess1.pcap*, 145
- nowebaccess2.pcap*, 147
- nowebaccess3.pcap*, 150
- passiveosfingerprinting.pcap*, 195
- ratinfected.pcap*, 207
- stranded_branchdns.pcap*, 157
- stranded_clientside.pcap*, 156
- synscan.pcap*, 191
- tcp_dupack.pcap*, 170
- tcp_handshake.pcap*, 102
- tcp_ports.pcap*, 99
- tcp_refuseconnection.pcap*, 105
- tcp_retransmissions.pcap*, 167
- tcp_tearardown.pcap*, 104
- tcp_zerowindowdead.pcap*, 177
- tcp_zerowindowrecovery.pcap*, 175–177
- tickedoffdeveloper.pcap*, 159
- twitter_login.pcap*, 134
- twitter_tweet.pcap*, 136
- udp_dnsrequest.pcap*, 106
- wrongdissector.pcap*, 74
- capture files, 47–49
 - automatically storing packets in, 54–55
 - conversations in, colorizing, 208
 - merging, 49
 - saving and exporting, 48
- capture filters, 56
 - BPF syntax, 58–61
 - sample expressions, 61–62
- Capture menu, Interfaces, 53
- Capture Options dialog, 53–54
 - Display options, 56
 - enabling name resolution, 73
 - for filtering, 57
 - Name Resolution section, 56
- Capture section, for Wireshark
 - preferences, 44
- capture type, for AirPcap, 220
- Chanalyzer software, 217–218
- channel hopping, 216
- channels, 216
 - changing when monitoring, 223
 - overlapping, 217
- Chappell, Laura, 240
- Chat category of expert
 - information, 82, 83
- CIDR (Classless Inter-Domain Routing), 92
- Cisco, set span command, 22
- Cisco router, 13
- Classless Inter-Domain Routing (CIDR), 92
- clearing filters, 193
- Client Identifier DHCP option
 - field, 117
- clients
 - in branch office, access to WAN, 155–159
 - latency, 181
 - misconfigured, 147
- closed ports, identifying, 193–194
- CloudShark, 237
- Colasoft Packet Builder, 237
- collection step, in sniffer process, 3
- collisions, on hub network, 20
- color
 - coding for packets, 45–46
 - in Follow TCP Stream window, 77
- Coloring Rules window
 - (Wireshark), 45–46
- colorization rule, exporting end-points to, 68
- colorizing conversations, 208
- Combs, Gerald, 35
- comma-separated values (CSV) files
 - saving capture file as, 48
 - transmission to central database, 159–163
- comparison operators, 64
- compiling Wireshark from source, 39–40

- computers
 - communication process, 4–14
 - data encapsulation, 8–10
 - OSI model, 5–8
 - protocols, 4
 - screen capture by attacker, 212
- connectionless protocol, 105–106
- Content Addressable Memory (CAM) table, 12, 86
- control packets (802.11), 223
- conversations, 68
 - in capture file, colorizing, 208
 - viewing, 69
- Conversations window
 - ESPN.com traffic in, 140–141
 - with TCP communications, 191–192
 - troubleshooting with, 70–71
- conversion step, in sniffer process, 3
- costs, of packet sniffers, 3
- CSV (comma-separated values) files
 - saving capture file as, 48
 - transmission to central database, 159–163
- CyberEYE remote-access Trojan (RAT), 207

D

- data encapsulation, 8–10
- data flow, halting with zero window notification, 175
- Data link layer (OSI), 6, 9
- data packets (802.11), 224
- data set, graph for overview, 79
- data-transfer rate
 - in application baseline, 186
 - in site baseline, 184
- data transmission, testing for corruption, 159–163
- DEB-based Linux distributions,
 - installing Wireshark on, 39
- Decode As dialog, 74
- default gateway, 147
 - attempt to find MAC address for, 145–146
- denial-of-service (DoS) attacks, 27

- Department of Defense (DoD)
 - model, 5
- destination port, for TCP, 98–100
- DHCP. *See* Dynamic Host Configuration Protocol (DHCP)
- direct install method, for sniffer placement, 31, 32
- direct messaging, in Twitter, 137
- discover packet for DHCP, 116–117
- Display Filter dialog, 65–66
- display filters, 56–65
 - sample expressions, 65
 - saving, 65–66
- dissection
 - expert information from, 82–84
 - viewing source code, 76
- DNS. *See* Domain Name System (DNS)
- DoD (Department of Defense)
 - model, 5
- domain controller, and branch office, 155–159
- Domain Dossier, 239
- Domain Name System (DNS), 120–129
 - communication problems, 157
 - filter for traffic, 142–143
 - name-to-IP address mapping, 149
 - packet structure, 121–122
 - queries, 122–123, 142
 - conditions preventing, 149
 - question types, 124
 - recursion, 124–127
 - resource record types, 124
 - zone transfers, 127–129
- DORA process, 115
- DoS (denial-of-service) attacks, 27
- dotted-quad notation, 91
- double-headed packet, 111
- downloading
 - NMAP tool, 191
 - pages from web server, 129–131
 - pOf tool, 196
 - WinPcap capture driver, 37
- dropping packets, 10
- dst qualifier, filter based on, 59

- duplicate ACK packet, 83,
171–172, 179
- Dynamic Host Configuration Protocol (DHCP), 113–120
 - acknowledgment packet, 119
 - discover packet, 116–117
 - in-lease renewal, 119–120
 - offer packet, 117–118
 - options and message types, 120
 - packet structure, 114–115
 - renewal process, 115–118
 - request packet, 118–119

E

- echo, vs. ping, 109
- Edit menu
 - Preferences, 44, 170
 - Name Resolution, 100
 - Set Time Reference, 53
- email message, with link to malicious site, 197
- encryption, 228
- endpoints, 67–68
 - exporting, to colorization rule, 68
 - monitoring, 204
 - viewing, 68–69
- Endpoints window, 68–69
 - troubleshooting with, 70–71
- Enterasys, set port mirroring create command, 22
- ephemeral port group, 99
- equal-to comparison operator (==), 64
- Error category of expert information, 82, 84
- ESPN.com traffic, 140–144
- Ethereal, 35
- Ethernet, 9
 - broadcast address, 88
 - hub, 10
 - networks
 - ARP process for computers on, 26–27
 - default MTU, 95
 - maximum frame size, 78
 - switch, rack-mountable, 11

- expert information, from dissection, 82–84
- exporting
 - capture files, 48
 - endpoint to colorization rule, 68
- expression, in BPF syntax, 58
- extracting
 - archive, 39
 - JPG data from Wireshark, 211–212

F

- Facebook
 - capturing traffic, 137–139
 - login process, 138
 - private messaging with, 139
 - vs. Twitter, 140
- fast retransmission, 84, 170, 172
- FCS filter, for AirPcap, 220
- file carving, 212
- Filter Expression dialog, 63
- Filter Expression Syntax Structure, 64–65
- filters, 56–66
 - addressing, 59
 - BPF syntax, 58–61
 - clearing, 193
 - display, 62–65
 - Filter Expression dialog, 63
 - Filter Expression Syntax Structure, 64–65
 - sample expressions, 65
 - for DNS traffic, 142–143
 - hostname and addressing, 59
 - port and protocol, 60
 - protocol field, 60–61
 - for STOR command, 160
 - with SYN scans, 192–193
 - wireless-specific, 226–228
- FIN flag, 103
- finding packets, 50
- Find Packet dialog, 50
- fingerprinting operating systems, 194–197
- flow graphing, 82
 - for data transmission testing, 159–160

- Follow TCP Stream feature, 76–77, 161–162
- footer, in packet, 8
- footprinting, 190
- forced decode, 74–76
- Fragment Offset field, for packets, 96, 97
- frames, maximum size on Ethernet network, 78
- frequency, filter for specific, 227–228
- Frequency/Channel data, for wireless, 225
- full-duplex devices, 11
 - switches as, 20
- full zone transfer (AXFR), 127
- Fyodor, 191

G

- gateway. *See* default gateway
- GET request packet (HTTP), 130, 135, 181
 - for Facebook, 138
- GIF file, to trigger exploit code, 200
- GNU Public License (GPL), 35
- graphing, 79–82
 - flow, 82
 - IO graphs, 79–80
 - round-trip time, 81
- gratuitous ARP, 89–90

H

- half-duplex mode, 10
- half-open scan, 190
- handshake for TCP, 101–103
 - initial sequence number, 169
 - and latency, 179
 - in Twitter authentication process, 134
- hardware, Wireshark requirements, 37. *See also* network hardware
- header in packet, 8
 - for ARP, 87–88
 - for ICMP, 107
 - for IPv4 header, 92–93

- for TCP, 98
- for UDP, 106–107
- help. *See* program support
- hexadecimal, searching for packets
 - with specified value, 50
- hex editor, 212
- Hide Capture Info Dialog option, 56
- high latency, 166, 179–183
- high-traffic servers, host baseline
 - for, 185
- host address, in IP address, 91
- host baseline, 185
- hostname, filters, 59
- host qualifier, for filter, 59
- hosts file, 149–150
- hping, 239
- HTTP. *See* Hypertext Transfer Protocol (HTTP)
- HTTPS, 134
- hubbing out, 22–23, 32
- hub network, collisions on, 20
- hubs, 10–11
 - finding “true,” 23
 - sniffing on network with, 19–20
- Hypertext Transfer Protocol (HTTP), 8–9, 129–132
 - browsing with, 129–131
 - posting data with, 131–132
 - viewing requests, 143–144

I

- IANA (Internet Assigned Numbers Authority), 240
- ICMP. *See* Internet Control Message Protocol (ICMP)
- Ident protocol, 193
- idle/busy traffic, in host baseline, 185
- IDS (intrusion detection system), 206
- IEEE (Institute of Electrical and Electronics Engineers), 216
- <iframe> tag (HTML), 200
- in-lease renewal for DHCP, 119–120
- incremental zone transfer (IXFR), 127

- installing Wireshark, 37–41
 - on Linux, 39–40
 - on Mac OS X, 40–41
 - on Microsoft Windows, 37–39
- Institute of Electrical and Electronics Engineers (IEEE), 216
- interference, between wireless channels, 217
- International Organization for Standardization (ISO), 5
- Internet access, troubleshooting configuration problems, 144–147
 - unwanted redirection, 147–150
- Internet Assigned Numbers Authority (IANA), 240
- Internet Control Message Protocol (ICMP), 107–112
 - echo requests and responses, 108–110
 - header, 107
 - ping, 95
 - types and messages, 107
- Internet Explorer, vulnerability in, 197
- Internet Protocol (IP), 9, 91–97
 - addresses, 26, 91–92
 - assignments, 70
 - dynamic assignment, 113–120
 - filtering packets with specific address, 64
 - finding. *See* Domain Name System (DNS)
 - fragmentation, 95–97
 - Time to Live (TTL), 93–95
 - v4 header, 92–93
- intrusion detection system (IDS), 206
- IO graphs, 79–80, 209–210
- IP. *See* Internet Protocol (IP)
- IP-to-MAC address mapping, updating cache with, 89–90
- IPv6 address, filter based on, 59
- ISO (International Organization for Standardization), 5
- iwconfig command, 222–223
- IXFR (incremental zone transfer), 127

J

- JFIF string, 209
- JPG file
 - extracting data from Wireshark, 211–212
 - to initiate attack communication, 209–211

K

- Keep Alive message, 84
- keep-alive packets, 175, 177–178, 179
- keys, for SSL, 135
- Kismet, 216
- Kozierok, Charles, *The TCP/IP Guide*, 240

L

- LAN (local area networks), 91
- latency, 166
 - locating framework, 182–183
 - locating source of high, 179–183
 - client latency, 181
 - normal communications, 180
 - server latency, 182
 - wire latency, 180–181
- layer 2 addresses, 26
- layer 8 issue, 7
- leases, from DHCP, 119–120
- LED lights on AirPcap, blinking, 220
- libpcap/WinPcap driver, 19, 239
- Linux
 - default number of retransmission attempts, 167
 - hosts file examination, 150
 - installing Wireshark on, 39–40
 - sniffing wirelessly, 222–223
 - traceroute utility, 112
- local area networks (LAN), 91
- location, for packet sniffer, 17–18, 31–32
- logical addresses, 9, 86
- logical operators
 - in BPF syntax, 58
 - for combining filter expressions, 64–65

login process
 for Facebook, 138
 for Twitter, 134–135
low latency, 166

M

MAC address, 26, 86
 ARP and, 18
 attempt to find for default
 gateway, 145–146
 filter based on, 59
 name resolution, 73
MAC Address Scanner dialog (Cain
 & Abel), 28
Mac OS X, installing Wireshark on,
 40–41
mailing lists, for program support, 3
make command, 40
malware
 redirecting users to websites
 with malicious code, 150
 risk of infection, 150
man-in-the-middle attacks, 140, 202
managed mode, for wireless NIC,
 218, 219
managed switches, 11
management packets (802.11), 223
mapping path, 110–112
marking packets, 51
master mode, for wireless NIC,
 218, 219
maximum transmission unit
 (MTU), and packet
 fragmentation, 95
MD5 hashes, 162–163
merging capture files, 49
Message Type DHCP option
 field, 116
message types, for DHCP, 120
messaging methods, Twitter vs.
 Facebook, 140
MetaGeek, 217
Microsoft Windows
 command shell, attacker use, 201
 default number of retransmis-
 sion attempts, 167
 hosts file examination, 150

 installing Wireshark on, 37–39
 sniffing wirelessly, 219–222
mission-critical servers, host base-
 line for, 185
monitor mode for wireless NIC,
 218, 219
 enabling in Linux, 222–223
monitor port, for nonaggregated
 taps, 25
More Fragments field, for packets,
 96, 97
MTU (maximum transmission
 unit), and packet
 fragmentation, 95
multicast traffic, 15

N

name resolution, 72–74
Name Resolution section, for Wire-
 shark preferences, 44
namespace, for DNS server
 management, 127
Netdude, 236
netmask (network mask), 91–92
network address, in IP address, 91
network baselining, 183–187
network diagrams, 31
network endpoints, 67–68. *See also*
 endpoints
network hardware, 10–14
 hubs, 10–11
 routers, 12–14
 switches, 11–12
 taps, 24–26
network interface card
 promiscuous mode support,
 18–19
 wireless card modes, 218–219
Network layer (OSI), 6
network maps, 31
network mask (netmask), 91–92
NetworkMiner, 238
network name resolution, 73
networks
 packet level as source of
 problems, 1
 traffic classifications, 14–15

- traffic flow, 14
- understanding normal traffic, 85
- network tap, 24–26, 32
- ngrep, 238
- NMAP tool, 191, 197
- No Error Messages message, 84
- nonaggregated network tap, 24, 25–26
- Nortel, port-mirroring mode
 - mirror-port command, 22
- NOT (!) operator, in BPF syntax, 58
- Note category of expert
 - information, 82, 83
- not filter expression logical
 - operator, 65
- Novak, Judy, 240

O

- Offer packet in DHCP, 117–118
- OmniPeek, 2
- one-way latency, 166
- open ports, identifying, 193–194
- operating systems. *See also* Linux;
Mac OS X; Microsoft
Windows
 - fingerprinting, 194–197
 - sniffer support, 3
 - Wireshark support, 37
- Operation Aurora, 197–202
- OR (||) operator, in BPF syntax,
58, 61
- or filter expression logical
 - operator, 65
- OSI model, 5–8
- out of lease, 119
- Out-of-Order message, 84
- oxid.it, 27

P

- packet analysis, 2
 - tools, 235–239
 - web resources, 239–240
- Packet Bytes pane (Wireshark), 43
- packet capture, 41–42. *See also* cap-
ture file examples

- Packet Details pane (Wireshark),
43, 153
 - Application Data in Info
column, 135
 - retransmission packet
information, 168
- Packet List pane (Wireshark), 43,
74, 153
 - adding columns to, 203,
225–226
 - for filter, 160
 - retransmissions in, 168
- packets
 - color coding, 45–46
 - dropping, 10
 - finding, 50
 - fragmentation, 95–97
 - length, 78–79
 - mapping path, 110–112
 - marking, 51
 - printing, 51–52
 - SYN flag, 148–149
 - term defined, 8
 - wireless types, filtering
specific, 227
- packet sniffers
 - evaluating, 2–3
 - guidelines, 32
 - how they work, 3–4
 - positioning for data capture,
17–18, 31–32
- packet sniffing, 2. *See also* packet
analysis
- Packetstan blog, 240
- packet time referencing, 52, 53
- Parameter Request List DHCP
 - option field, 117
- passive fingerprinting, 194–196
- .pcap file format, 48. *See also* cap-
ture file examples
- pcapr, 237–238
- PDF file, printing packets to, 51
- PDU (protocol data unit), 8
- performance, 165–187. *See also*
latency
network baselining, 183–187
Selective ACK and, 172

- Perl, 239
- physical addresses, 86
- Physical layer (OSI), 5, 6, 9
- ping utility, 108
- plaintext, saving capture file as, 48
- pOf tool, 196
- Poor, Mike, 240
- port mirroring, 21–22, 32
 - for checking for data corruption, 159
 - for troubleshooting printer, 153
- port-mirroring mode mirror-port command (Nortel), 22
- ports
 - attacker research on, 190
 - attackers' efforts to determine open, 190
 - blocking traffic, 158
 - filter based on, 60
 - filter to show all traffic using specific, 192
 - filtering packet capture by, 57
 - filters to exclude, 60
 - for HTTP, 130
 - identifying open and closed, 193–194
 - list of common, 101
 - for TCP, 99–101
- port spanning, 21. *See also* port mirroring
- posting data with HTTP, 131–132
- POST method, 132
 - for Facebook, 139
 - for tweet, 136
- POST packet (HTTP), 131
- PostScript, saving capture file as, 48
- Preferences dialog (Wireshark), 44
 - Name Resolution section, 100
 - Protocols section, 170
- Presentation layer (OSI), 5
- Previous Segment Lost message, 84
- primitives, in BPF syntax, 58
- Print dialog, 51
- printing packets, 51–52
- Printing section, for Wireshark preferences, 44
- privacy, of Twitter direct messages, 137
- private messaging, with Facebook, 139
- problems. *See* troubleshooting
- program support
 - evaluating, 3
 - for Wireshark, 37
- promiscuous mode, 3
 - network interface card support for, 18–19
- protocol analysis, 2. *See also* packet analysis
- protocol data unit (PDU), 8
- protocol field filters, 60–61
- Protocol Hierarchy Statistics, 71–72, 141–142, 184
- protocols, 4
 - in application baseline, 186
 - color coding in Wireshark, 45–46
 - dissection, 74–76
 - filter based on, 60
 - in host baseline, 185
 - lower-layer, 85–112
 - Address Resolution Protocol (ARP), 86–90
 - Internet Control Message Protocol (ICMP), 107–112
 - Internet Protocol (IP), 91–97
 - Transmission Control Protocol (TCP), 98–105
 - User Datagram Protocol (UDP), 105–107
 - and OSI model, 6
- packet sniffer evaluation and, 2
- in site baseline, 184
- support by Wireshark, 37
- upper-layer, 113–132
 - Domain Name System (DNS), 120–129
 - Dynamic Host Configuration Protocol (DHCP), 113–120
 - Hypertext Transfer Protocol (HTTP), 129–132

Protocols section, for Wireshark
 preferences, 44
protocol stack, 4
public forums, for program
 support, 3
Python, 239

Q

qualifiers, in BPF syntax, 58
queries in DNS, 122–123, 142
 conditions preventing, 149

R

rack-mountable Ethernet switch, 11
RAT (remote-access Trojan),
 206–213
reassembly, for packets in FTP-
 DATA stream, 160–161
receive window, 173
 adjusting size, 174, 176
 halting data flow, 175
Received Signal Strength Indica-
 tion (RSSI), 225
reconnaissance by potential
 attacker, 190–197
redirection, troubleshooting
 unwanted, 147–150
remote-access Trojan (RAT),
 206–213
remote server, lack of response, 152
repeating device, hub as, 10
Replay Counter field, 232
report-generation module, free vs.
 commercial sniffers, 3
Request for Comments (RFC)
 791, on Internet Protocol v4, 91
 792, on ICMP, 107
 793, on TCP, 98
 826, on ARP, 86
 DNS-related, 120
request packet, 8
 in DHCP, 118–119
Requested IP Address DHCP option
 field, 117
resource records in DNS servers, 120

retransmission packets, 154,
 166–169, 178–179
retransmission timeout (RTO), 154,
 166, 168
retransmission timer, 166
RFC. *See* Request for
 Comments (RFC)
Ring Buffer With option, 55
RIPE (Europe), 70
Riverbed, 219
RJ-45 ports, 10
round-trip time (RTT), 166
 graphing, 81
routed environment, sniffing on,
 30–31
routers, 12–14
 for connecting LANs, 91
RPM-based Linux distributions,
 installing Wireshark on, 39
RSSI (Received Signal Strength
 Indication), 225
RST flag, 148–149
RTO (retransmission timeout), 154,
 166, 168
RTT (round-trip time), 166
 graphing, 81

S

Sanders, Chris, blog, 240
SANS Security Intrusion Detection
 In-Depth course, 239–240
saving
 capture files, 48
 display filters, 65–66
 file set, 55
Scapy, 236
screen capture, of victim
 computer, 212
<script> tag (HTML), 198–199
secondary DNS server, 127
Secure Socket Layer (SSL), 74
 over HTTP, 134–135
security for wireless, 189–213,
 228–233
 for baseline, 187
 exploitation, 197–213

- security for wireless (*continued*)
 - reconnaissance, 190–197
 - remote-access Trojan, 206–213
 - screen capture by attacker, 212
 - Twitter and, 136–137
- WEP authentication
 - failed, 230
 - successful, 229–230
- WPA authentication
 - failed, 232–233
 - successful, 231–232
- Selective Acknowledgment
 - feature, 172
- sequence numbers, in TCP
 - packet, 169
- server latency, 182
- Session layer (OSI), 5
- set port mirroring create command (Enterasys), 22
- set span command (Cisco), 22
- site baseline, 184
- sliding window mechanism (TCP),
 - 173, 175–178
- slow network. *See* performance
- Sniffer tab (Cain & Abel), 28
- sniffing the wire, 17
- Snort project, 202
- social networking, packets for,
 - 134–140
- source code for dissector,
 - viewing, 76
- source port, for TCP, 99, 100
- tag (HTML), 200
- spear phishing, 197
- spectrum analyzer, 217
- src qualifier, filter based on, 59
- SSL (Secure Socket Layer), 74
 - over HTTP, 134–135
- standard port group, 99
- startup/shutdown
 - in application baseline, 186
 - in host baseline, 185
- Statistics menu
 - Conversations, 69, 140–141
 - Flow Graph, 82, 159
 - HTTP, 143
 - IO Graphs, 79
 - Packet Lengths, 78

- Protocol Hierarchy, 71, 141–142
- Summary, 143
- TCP Stream Graph, Round Trip Time Graph, 81
- Statistics section, for Wireshark
 - preferences, 44
- stealth scan, 190
- Stevens, Richard, *TCP/IP Illustrated*, 240
- Stop Capture settings, 55
- STOR command (FTP), 160
- subnet mask, 91–92
- Summary window, 143–144
- switches, 11–12
 - sniffing on network with, 20–30
 - ARP cache poisoning, 26–30
 - hubbing out, 22–23
 - port mirroring, 21–22
 - using tap, 24–26
- SYN/ACK packet, 102
- SYN packet, 102, 148–149, 151–152
 - lack of response, 158
 - response, 180
- SYN scans, 190–194
 - filters with, 192–193

T

- tar command, 39
- TCP. *See* Transmission Control Protocol (TCP)
- tcpdump, 2, 235–236
- TCP/IP, address resolution
 - process, 86
- TCP/IP Guide* (Kozierok), 240
- TCP/IP Illustrated* (Stevens), 240
- Tcpplay, 238
- terminating TCP connection,
 - 148–149
- three-way handshake for TCP,
 - 101–103
 - initial sequence number, 169
 - and latency, 179
 - in Twitter authentication process, 134
- throughput
 - graphing, 79
 - of ports being mirrored, 22

Time Display Formats, 52
Time to Live (TTL), 93–95
Traceroute, 110–112
traffic signatures, 202
Transmission Control Protocol
 (TCP), 8–9, 98–105
 buffer space, 173
 capturing only packets with RST
 flag set, 61
 DNS and, 127, 157–158
 duplicate acknowledgments,
 169–172
 error-recovery features, 166–172
 retransmission, 166–169
 expert info messages config-
 ured for, 83–84
 flow control, 173–178
 following streams, 76–77
 header, 98
 HTTP and, 129–130
 learning from error- and flow-
 control packets, 178–179
 resets, 104
 retransmission packets, 83, 154
 sliding window mechanism, 173,
 175–178
 SYN scan, 190–194
 teardown, 103–104
 terminating connection,
 148–149
 three-way handshake, 101–103
 initial sequence number, 169
 and latency, 179
 in Twitter authentication
 process, 134
Transmission Rate (TX Rate), for
 wireless, 225
Transport layer (OSI), 6, 8–9
transport name resolution, 73
trigger for exploit code, GIF file
 for, 200
troubleshooting
 branch office connections,
 155–159
 developer tensions, 159–163
 with Endpoints and Conversa-
 tions windows, 70–71
 latency, 178–179

 no Internet access
 from configuration
 problems, 144–147
 from unwanted redirection,
 147–150
 from upstream problems,
 150–153
 printer inconsistency, 153–155
 slow networks, 166
 wireless signal interference, 217
TTL (Time to Live), 93–95
Twitter
 capturing traffic, 134–137
 direct messaging, 137
 vs. Facebook, 140
 login process, 134–135
 sending data, 136–137
TX Rate (Transmission Rate), for
 wireless, 225

U

Ubuntu, installing Wireshark on, 39
UDP. *See* User Datagram
 Protocol (UDP)
unicast packet, 15
unmarking packets, 51
Update List of Packets in Real Time
 option, 56
uploading data to web server,
 131–132
upstream problems, troubleshoot-
 ing lack of Internet access
 from, 150–153
User Datagram Protocol (UDP),
 105–107, 157
 DHCP and, 116
 DNS and, 123
 header, 106–107
 and latency, 182
user-friendliness
 of packet sniffers, 3
 of Wireshark interface, 37
User Interface section, for Wire-
 shark preferences, 44
user privileges, for promiscuous
 mode, 19
USER request command (FTP), fil-
 ter for traffic, 160–161

V

- viewing
 - conversations, 69
 - endpoints, 68–69
- View menu
 - Time Display Format, 52, 53, 154–155
 - Seconds Since Previous Displayed Packet, 179
- visibility window, 20, 21

W

- WAN (wide area network), branch office access, 156
- WAP (Wireless Access Protocol)
 - beacon packet, 231
 - broadcast packet from, 224
- Warning category of expert information, 82, 84
- web resources
 - on DHCP options, 120
 - DNS-related RFCs, 120
 - on DNS resource record types, 124
 - on intrusion detection and attack signatures, 202
 - on packet analysis, 239–240
 - on packet analysis tools, 236–239
 - on wireless capture filters, 228
- web server
 - downloading pages from, 129–131
 - uploading data to, 131–132
- websites, capturing traffic, 140–144
- WEP. *See* Wired Equivalent Privacy (WEP)
- WHOIS utility, 70
- wide area network (WAN), branch office access, 156
- Wi-Fi Protected Access (WPA), 228
 - authentication
 - failed, 232–233
 - successful, 231–232
- Window is Full message, 84
- Windows. *See* Microsoft Windows
- Windows command shell, attacker use, 201

- Windows Size field, 175–176
 - Window Update message, 83
 - Windump, 235–236
 - WinHex, 212
 - WinPcap capture driver, 37
 - Wired Equivalent Privacy (WEP), 228
 - authentication
 - failed, 230
 - successful, 229–230
 - configuration with AirPcap, 220
 - wire latency, 180–181
 - Wireless Access Protocol (WAP)
 - beacon packet, 231
 - broadcast packet from, 224
 - wireless packet analysis, 215–233
 - 802.11 packet structure, 223–225
 - adding columns to Packet List pane, 225–226
 - filters specific to, 226–228
 - NIC modes, 218–219
 - physical considerations, 216–217
 - signal interference, 217
 - sniffing channel at a time, 216
 - security, 228–233
 - failed WEP
 - authentication, 230
 - failed WPA authentication, 232–233
 - successful WEP authentication, 229–230
 - successful WPA authentication, 231–232
 - sniffing
 - in Linux, 222–223
 - in Windows, 219–222
- Wireshark University, 240
 - Wireshark
 - and AirPcap, 221
 - benefits, 36–37
 - fundamentals, 41–46
 - first packet capture, 41–42
 - main window, 42–43
 - preferences, 43–44
 - hardware requirements, 37
 - history, 35–36

home page, 239
installing, 37–41
 on Linux, 39–40
 on Mac OS X, 40–41
 on Microsoft Windows, 37–39
libpcap/WinPcap driver, 19, 239
relative sequence numbers, 170
Wi-Spy, 217
WPA (Wi-Fi Protected Access), 228
 authentication
 failed, 232–233
 successful, 231–232

X

XML, saving capture file as, 48
xor filter expression logical
 operator, 65

Z

Zero Window message, 84
zero window notification, 175,
 176, 179
Zero Window Probe message, 83, 84
zone transfers
 for DNS, 127
 risk from allowing access to
 data, 128
failed, 158