Writing, Hacking, and Modifying Security Tools

Network Security Tools

O'REILLY®

Nitesh Dhanjani & Justin Clarke

CHAPTER 8

Developing Web Assessment Tools and Scripts

Web application vulnerabilities are increasingly becoming the attacker's method of choice for compromising systems and obtaining access to valuable data. Although most organizations have a reliable process in place for identifying and defending perimeter hosts from traditional network-based attacks, often little or no attention is paid to security over custom web applications that are deployed to allow employees, customers, or business partners to access company data. In addition, although a myriad of tools is available to automatically assess and identify network-based vulnerabilities, open source and freeware alternatives for identifying vulnerabilities in custom web applications are lacking. In this chapter, we walk through the process of developing a simple web application scanner using the Perl scripting language and its powerful LWP module.

It is important to define the types of vulnerabilities we identify in this chapter. Many people think CGI scanners, such as Nikto (discussed in Chapter 4), are considered web application scanners. Although these scanners do in fact have the potential to identify "known" vulnerabilities in specific pages or files, they do not identify vulnerabilities that are unique to a given web application. For example, the popular PHP-Nuke application has multiple vulnerabilities for which Nikto contains a signature, but a Nikto signature is unlikely to be available for a vulnerability that might be present in a custom web application your company has built. To identify these unique vulnerabilities, the scanner must be able to dynamically generate test requests that are tailored specifically to a given web application.

This chapter introduces two simple Perl scripts you can use to assess a custom web application for common vulnerabilities. Before we begin developing the scripts, however, you must first understand the nature of web application vulnerabilities and the environment in which these applications operate.

Web Application Environment

The term *web application* typically implies certain attributes an application has. Most often, it means that the application is browser-based—i.e., you can access it using a standard web browser such as Internet Explorer or Netscape Navigator. For the purposes of our discussions in the next two chapters, we assume the web applications communicate using the Hypertext Transfer Protocol (HTTP) and that users access them via a web browser.

HTTP

Most web applications use HTTP to exchange data between the client (typically a web browser such as Internet Explorer or Netscape Navigator) and the server. HTTP works through a series of *requests* from the client and associated server *responses* back to the client. Each request is independent and results in a server response. A detailed familiarity with HTTP requests and responses is critical to effectively test web applications. Example 8-1 shows what a typical raw HTTP request looks like.

Example 8-1. Typical HTTP GET request

```
GET /public/content/jsp/news.jsp HTTP/1.1
Accept: image/gif, image/x-xbitmap, image/jpeg, */*
Accept-Language: en-us
Accept-Encoding: gzip, deflate
User-Agent: Mozilla/4.0 (compatible; MSIE 6.0)
Host: www.myserver.com
Connection: Keep-Alive
```

The first line of the HTTP request typically contains the request method—in this case, the `GET` method—followed by the file or resource being requested. The version of HTTP the client uses is also appended to the first line of the request. Following this line are various request headers and associated values.

Several HTTP request methods are defined in the HTTP RFC; however, by far the two most common are the `GET` and `POST` methods. The primary difference between these methods is in how application parameters are passed to the file or resource being requested. Requests for resources that do not include parameter data are typically made using the `GET` request (as shown in Example 8-1). `GET` requests, however, can also include parameter data in the query string portion of the request. The query string normally consists of at least one parameter name/value pair appended to the end of the resource being requested. Use a question mark (`?`)to separate the resource name from the query string data, and you use an equals sign (`=`) to separate the parameter name/value pair. You can pass multiple parameter name/value pairs in the query string and concatenate them using an ampersand (`&`). Example 8-2 shows the same `GET` request from Example 8-1, but it contains request data in the query string.

Example 8-2. HTTP GET request with query string data

```
GET /public/content/jsp/news.jsp?id=2&view=F HTTP/1.1
Accept: image/gif, image/x-xbitmap, image/jpeg, */*
Accept-Language: en-us
Accept-Encoding: gzip, deflate
User-Agent: Mozilla/4.0 (compatible; MSIE 6.0)
Host: www.myserver.com
Connection: Keep-Alive
```

The `POST` request method is very similar to the `GET` method, with the exception of how parameter name/value pairs are passed to the application. A `POST` request passes name/value pairs with the same syntax as that used in a `GET` request, but it places the data string in the body of the request after all request headers. The Content-Length header is also passed in a `POST` request to indicate to the HTTP server the length of the `POST` data string. The Content-Length header value must contain the exact number of characters in the `POST` data string. Example 8-3 shows the request from Example 8-2, but this time using the `POST` method.

Example 8-3. HTTP POST request with data

```
POST /public/content/jsp/news.jsp HTTP/1.1
Accept: image/gif, image/x-xbitmap, image/jpeg, */*
Accept-Language: en-us
Accept-Encoding: gzip, deflate
User-Agent: Mozilla/4.0 (compatible; MSIE 6.0)
Host: www.myserver.com
Content-Length: 11
Connection: Keep-Alive

id=2&view=F
```

Each HTTP request results in a response from the server. The structure of the HTTP response is somewhat similar to that of a request, consisting of the HTTP version and response code in the first line, followed by a series of response headers and values. The HTML output the browser renders is included in the body of the HTTP response following the response headers. Unlike the HTTP response headers, the HTML output is rendered to the user and can be viewed in its raw state using the View Source option in most web browsers. Example 8-4 shows a typical HTTP response.

Example 8-4. HTTP response

```
HTTP/1.1 200 OK
Date: Sat, 10 Jul 2004 23:45:12 GMT
Server: Apache/1.3.26 (Unix)
Cache-Control: no-store
Pragma: no-cache
Content-Type: text/html; charset=ISO-8859-1

<HTML>
```

Example 8-4. HTTP response (continued)

```
<HEAD>
<TITLE>My News Story</TITLE>
</HEAD>
<BODY>
<H1>My News Story</H1>
<P>This is a simple news story.</P>
</BODY>
</HTML>
```

The response status code consists of a three-digit number returned in the first line of the HTTP response. An HTTP server can return several status codes, all classified based on the first of the three digits. Table 8-1 shows a breakout of the five general status code categories.

Table 8-1. HTTP response codes

Status code	Category
1XX (i.e., 100 Continue)	Informational
2XX (i.e., 200 OK)	Success
3XX (i.e., 302 Object Moved)	Redirection
4XX (i.e., 404 File Not Found)	Client Error
5XX (i.e., 500 Internal Server Error)	Server Error

SSL

You can use Secure Sockets Layer (SSL) to encrypt the communications channel between the web browser client and server. Although this is usually referred to as *HTTPS*, underneath the encryption the HTTP requests and responses still look the same. Many people think that simply because HTTPS is used, the application or server is “secure” and resilient to attack. It is important to realize that SSL merely protects the request and response data while in transit so that someone eavesdropping on the network or otherwise intercepting the data cannot read it. The underlying data and associated application, however, are still susceptible to end-user attack.

Common SSL Misconceptions

- The web server is secure because SSL is used.
- SSL secures the web application.
- HTTP exploits do not work over SSL.

Perl and LWP

We will use the Perl scripting language to develop the web application scanner outlined in this chapter. Perl's extensive support of regular expressions and platform independence makes it a great language with which to develop our scanner. We have kept the code syntax as straightforward and easy-to-follow as possible, and we will explain each block of code as we develop it. We will use the Libwww-perl user agent module (`LWP::UserAgent`) native to many Perl installations. LWP is essentially a WWW client library that allows you to easily make HTTP requests from a Perl script. If you want to learn more about LWP, read *Perl and LWP*, by Sean Burke (O'Reilly).

Got LWP?

If you're not sure whether LWP is included in your PERL installation, use the following command to check:

```
% perl -MLWP -le "print(LWP->VERSION)"
```

If LWP is not already installed, you should obtain and install the most recent version from the Comprehensive Perl Archive Network (CPAN). Use the following commands to install LWP using CPAN:

```
% perl -MCPAN -eshell
cpan> install Bundle::LWP
```

Another nice thing about LWP is that it supports HTTP requests over SSL as long as the `Crypt::SSLeay` Perl module and OpenSSL libraries are installed. If you want to use the scanner on HTTPS web applications, ensure that the `Crypt::SSLeay` module and OpenSSL libraries are installed and working.

Web Application Vulnerabilities

When we use the term *web application vulnerabilities*, we are referring to a vulnerability that is the result of poorly written application code. These vulnerabilities can range from application components that do not properly validate external input before processing (such as SQL injection), to flaws in the code that do not properly authenticate users before allowing access. The nature and classifications of web application vulnerabilities are outside the scope of this chapter, but we give a quick overview of these vulnerabilities in the sidebar "Open Web Application Security Project."

Open Web Application Security Project

If you are not familiar with web application vulnerabilities, the Open Web Application Security Project (*www.owasp.org*) is a great resource that can bring you up to speed. OWASP has developed a Top Ten List of the most critical web application vulnerabilities. The list is not all-inclusive, but it represents many of the critical issues present in web-based applications.

Top Ten Most Critical Web Application Vulnerabilities 2004

1. Unvalidated input
2. Broken access control
3. Broken authentication and session management
4. Cross-Site Scripting (XSS)
5. Buffer overflows
6. Injection flaws
7. Improper error handling
8. Insecure data storage
9. Denial of service
10. Insecure configurations management

Designing the Scanner

Before we start actually building the scanner, we need to define the functional requirements and overall structure of how the scanner should operate.

Functional Requirements

The first thing our scanner will do is obtain data about the target application from which to generate its test requests. To run customized testing routines that are designed for a specific web application, you must somehow obtain data about the application. Application *spidering*, or *crawling*, is a very effective technique you can perform to "inventory" or record legitimate application pages and input parameter combinations. You can automatically crawl an application using existing utilities such as Wget, or you can do it manually with the help of a local proxy server such as Odysseus or Burp. Most of the commercial application scanners, such as Sanctum's AppScan and SPI Dynamics' WebInspect, offer users both of these data-collection methods. The goal in either case is to build a collection of request samples to every application page as a basis on which to build the list of test requests for the scanner to make.

Although the automated technique is obviously faster and easier, it has a disadvantage in that it might not effectively discover all application pages for a variety of

reasons. Primarily, the crawl agent must be able to parse HTML forms and generate legitimate form submissions to the application. Many applications present certain pages or functionality to the user only after a successful form submission. Even if the spidering agent can generate form parsing and submissions, many applications require the submissions to contain legitimate application data; otherwise, the application's business logic prevents the user from reaching subsequent pages or areas of the application. Another thing to consider with automated spidering agents is that because they typically follow every link and/or form a given web application presents, they might cause unanticipated events to occur. For example, if a hyperlink presented to the user allows certain transactions to be processed or initiated, the agent might inadvertently delete or modify application data or initiate unanticipated transactions on behalf of a user. For these reasons, most experienced testers normally prefer the manual crawling technique because it allows them to achieve a thorough crawl of the application while maintaining control over the data and pages that are requested during the crawl and ultimately are used to generate test requests.

Our scanner will rely on a manual application crawl to discover all testable pages and requests. To accomplish this, we will use one of the many available freeware local proxy server utilities to record all application requests in a log file as we manually crawl the application. To extract the relevant data from the log file, first we will need to create a log file parsing script. The parsing script is used to generate a reasonably simple input file that our scanner will use. By developing a separate script for log file parsing, our scanner will be more flexible because it will not be tied to a specific local proxy utility. Additionally, this will give us more control over the scan requests because we will be able to manually review the requests that are used to perform testing without having to sift through a messy log file. Keep in mind that the input file our scanner will use should contain only legitimate, or untainted, application requests. The specific attack strings used to test the application will be generated on-the-fly by our scanner.

Now that we know how the scanner will obtain data about the application (an input file generated from a manual crawl proxy log file), we must decide what tests our scanner will conduct and how it will perform its testing. For web applications, we can perform a series of common tests to identify some general application and/or server vulnerabilities. First we will want to perform input validation testing against each application input parameter. At a minimum we should be able to perform tests for SQL injection and XSS, two common web application vulnerabilities. Because we will be performing these tests against each application input parameter, we refer to them as *parameter-based tests*.

In addition to parameter-based testing, we will want to perform certain tests against each application server directory. For example, we will want to make a direct request to each server directory to see if it permits a directory listing that exposes all files contained within it. We also will want to check to see if we can upload files to each directory using the HTTP `PUT` method because this typically allows an attacker to

upload his own application pages and compromise both the application and the server. Going forward we refer to these tests as *directory-based tests*.

For reporting purposes, our scanner should be able to report the request data used for a given request if it discovers a potential vulnerability, and report some information regarding the type of vulnerability it detected. This information will allow us to analyze and validate the output to confirm identified issues. As such, our scanner should be able to generate an output file in addition to printing output to the screen. The final requirement for our scanner is the ability to use HTTP cookies. Most authenticated applications rely on some sort of authentication token or session identifier that is passed to the application in the form of a cookie. Even a simple scanner such as the one we are building needs to have cookie support to be useful.

Scanner Design

Now that we have defined the basic requirements for our scanner, we can start to develop an overview of the scanner's overall structure. Based on our requirements, two separate scripts will be used to perform testing. We will use the first script to parse the proxy log file and generate an input file with the request data to be used for testing. The second script will accept the input file and perform various tests against the application based on the pages and parameter data contained in the file.

parseLog.pl

Our first script is called *parseLog.pl*, and it is used to parse the proxy server log file. This script accepts one mandatory input argument containing the name of the file to be parsed. The script's output is in a simple format that our scanner can use as input. At this point, it probably makes sense to define the actual structure of the input file and the requests contained within it. We must keep in mind here that we most likely will see the following types of requests in our log file:

- `GET` requests (without a query string)
- `GET` requests (with a query string)
- `POST` requests

To handle these request types, we generate a flat text file with one line for each request, as shown in Example 8-5. The first portion of the line contains the request method (`GET` or `POST`), followed by a space, and then by the path to the resource being requested. If the request uses the `GET` method with query string data, it is concatenated to the resource name using a question mark (?). This is the same syntax used to pass query string data as defined by HTTP, so it should be fairly straightforward. For `POST` requests, the `POST` data string is concatenated to the resource name using the same convention (a question mark). Because it is a `POST` request, the scanner knows to pass the data to the server in the body of the HTTP request rather than in the query string.

Example 8-5. Sample input file entries

```
GET /public/content/jsp/news.jsp?id=2&view=F
GET /public/content/jsp/news.jsp?id=8&view=S
GET /images/logo.gif
POST /public/content/jsp/user.jsp?fname=Jim&Lname=Doe
POST /public/content/jsp/user.jsp?fname=Jay&Lname=Doe
GET /images/spacer.gif
GET /content/welcome.jsp
```

Another nice thing about using this input file format is that it enables us to easily edit the entries by hand, as well as easily craft custom entries. Because the script's only purpose is to generate input file entries, we don't need it to generate a separate output file. Instead, we simply use the greater-than (>) character to redirect the script's output to a local file when we run it to save it to a file. You will also notice that the input file contains no hostname or IP address, giving us the flexibility to use the input file against other hostnames or IP addresses if our application gets moved.

As for the proxy server that our parsing script supports, we are using the Burp freeware proxy server (*http://www.portswigger.net*). We chose Burp because of its multiplatform support (it's written in Java) and because, like many local proxy tools, it logs the raw HTTP request and response data. Regardless of which proxy tool you use, as long as the log file contains the raw HTTP requests the parsing logic should be virtually identical. We will more closely examine the Burp proxy and its log format a bit later in the chapter.

simpleScanner.pl

Now that we have a basic design of our log file parsing script we can start designing the actual scanner, which is called *simpleScanner.pl*. We have already stated that the script needs to accept an input file, and based on the format of the input file we just defined, the script also needs to include a second mandatory input argument consisting of the hostname or IP address to be tested. In addition to these two mandatory input arguments, we also need to have some optional arguments for our scanner. When we defined the scanner requirements, we mentioned that the tool would need to be able to generate an output file and support HTTP cookies. These two features are better left as optional arguments because they might not be required under certain circumstances. We also will add an additional option for verbosity so that our scanner has two levels of reporting.

At the code level, we will develop a main script routine that controls the overall execution flow, and we will call various subroutines for each major task the scanner needs to perform. This allows us to segregate the code into manageable blocks based on overall function, and it allows us to reuse these routines at various points within the execution cycle. The first task our scanner needs to do is to read the entries from the input file. Once the file has been parsed, each individual request is parsed to perform our parameter- and directory-based testing.

A common mistake when testing application parameters for input validation is to *fuzz*, or alter, several parameters simultaneously. Although this approach allows us to test multiple parameters at once, contaminated data from one parameter might prevent another from being interpreted by the code. For our parameter-based testing, only one parameter will be tested at a time while the remaining parameters contain their original values obtained from the log file entry. In other words, there will be one test request for each variable on each application page. To minimize the number of unnecessary or redundant test requests, we also track each page and the associated parameter(s) that are tested. Only unique page/parameter combinations will be tested to avoid making redundant test requests.

Once every parameter of a given request has been tested, all parameter values are stripped from the request and the URL path is truncated at each directory level to perform directory-based testing. Again, one request is made for each directory level of the URL path, and we keep track of these requests to avoid making duplicate or redundant requests. Figure 8-1 visually represents the logic of our tool.

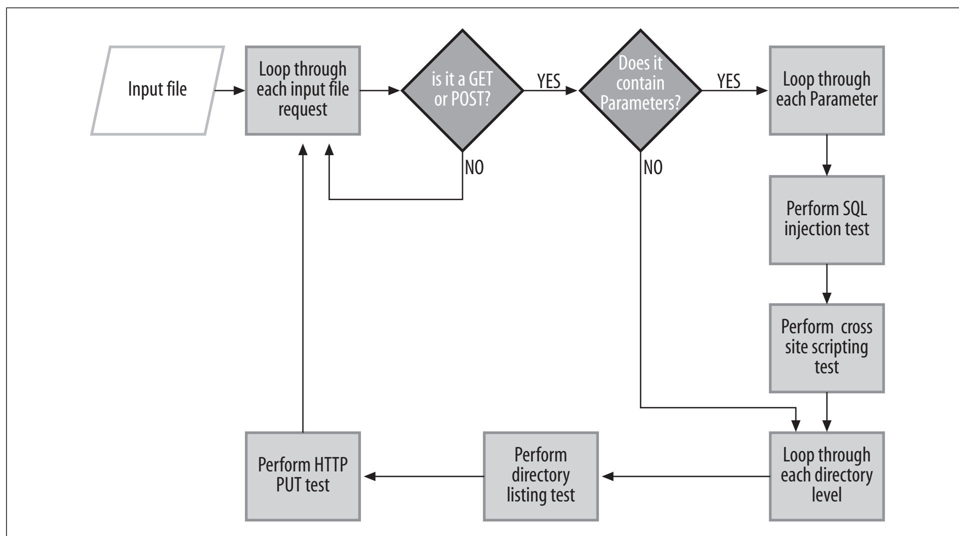


Figure 8-1. Visual representation of scanner logic

Now we are almost ready to begin coding our scanner, but first we should quickly review the process of generating test data using a local proxy server.

Generating Test Data

You can use any local proxy server to record a manual crawl of an application, provided it supports logging of all HTTP requests. Most proxy servers of this type also natively support SSL and can log the plain-text requests the browser makes when using HTTPS. Once the manual crawl is complete we should have a log file containing all the raw HTTP requests made to the application.

Our *logParse* script is designed to work with the Burp proxy tool. Burp is written in Java and you can freely download it from the PortSwigger web site mentioned earlier. You will also need a Java Runtime Environment (JRE), preferably Sun's, installed on the machine on which you want to run Burp. You can download the most recent version of the Sun JRE from *http://www.java.com/en/download/*.

Once you download and run Burp, you need to make sure logging to a file has been enabled and you are not intercepting requests or responses. By logging without intercepting, the proxy server seamlessly passes all HTTP requests back and forth without requiring any user interaction, and it logs all requests to the log file. Figure 8-2 shows the Burp options necessary to generate the activity log.

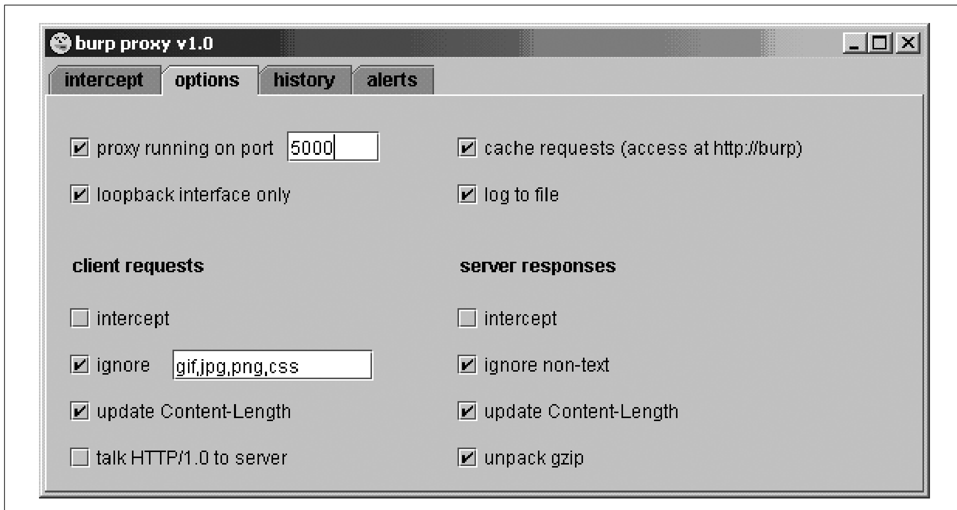
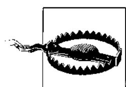


Figure 8-2. Burp options screen

You also need to set your web browser to use Burp as a proxy server (by default the hostname is `localhost` and the port number is `5000`). Because the goal of this phase is to inventory all application pages and parameters, no testing or parameter manipulation should be done during the crawl. The log file should ideally contain only legitimate application requests and legitimate parameter values. We want to ensure that when crawling the web application all application links are followed and all application forms are submitted successfully. Once you have successfully crawled the entire application, you should make a copy of the log file to use for testing.



The log file generated during the application crawl contains a plaintext record of all data, including potentially sensitive information, passed to the application. This will likely include the username and password used to authenticate to the application.

Building the Log Parser

We are finally ready to start writing some code. The first thing we do is open our script and check whether a log filename was passed (the only mandatory argument). If not, the script dies and prints the script usage; otherwise, it continues:

```
#!/usr/bin/perl

use strict;

# Check for mandatory arguments or print out usage info
unless (@ARGV) {
 die "Usage: $0 LogFile\n";
}
```

Now that we know a command-line argument was passed, we assume it was the log file name and attempt to open the file. If we cannot open the file, the script dies and prints an error message:

```
# Attempt to open the input file
open(IN, "<", $ARGV[0]) or die"ERROR: Can't open file $ARGV[0].\n";
```

Before we go any further, it is imperative that we be familiar with the structure and format of the log file we are parsing. Provided that the proxy server you are using is logging the raw HTTP requests and responses (most of them do), the logic to generate test requests from our Perl script should be virtually identical, with the exception of the delimiter used to separate each log file entry. Looking at the Burp log file shown in Example 8-6, notice that each request and response is separated with a consistent delimiter (`"=" 54 x`).

Example 8-6. Excerpt from Burp proxy log file

```
======================================================
http://www.myserver.com/192.168.0.1:80
======================================================
GET /blah.jsp HTTP/1.0
Accept: */*
Accept-Language: en-us
Pragma: no-cache
User-Agent: Mozilla/4.0 (compatible; MSIE 6.0; Windows NT 5.2)
Host: www.myserver.com
Proxy-Connection: Keep-Alive


======================================================
HTTP/1.1 200 OK
Server: Apache/1.3.27 (Unix)
Date: Sun, 11 Jul 2004 17:21:01 GMT
Content-type: text/html; charset=iso-8859-1
Connection: close

<html>
```

Example 8-6. Excerpt from Burp proxy log file (continued)

```
  <head>
    <title>Test Page</title>
  </head>
  <body>
     <P>Hello World!</P>
  </body>
</html>
======================================================
```

Going back to the script, now that the file is open we place its contents into an array (`@logData`). We also change the default record separator (`$/`) to be the delimiter of our log file entries. That way, each array member in `@logData` is a separate log entry.

```
# Populate logData with contents of input file
my @logData = <IN>;

# Change the input record separator to select entire log entries
$/ = "=" x 54;
```

Next, we loop through each log file entry and parse the first line of the request to determine if it is a `GET` or a `POST` request:

```
# Loop through each request and parse it
my ($request,$logEntry, @requests);
foreach $logEntry (@logData) {

 # Create an array containing each line of the raw request
 my @logEntryLines = split(/\n/, $logEntry);

 # Create an array containing each element of the first request line
 my @requestElements = split(/ /, $logEntryLines[1]);

 # Only parse GET and POST requests
 if ($requestElements[0] eq "GET" || $requestElements[0] eq "POST" ) {
```

For `GET` requests, we simply parse the first two members of the `@requestElements` array. These two elements should consist of the method (`GET`) and the resource being requested. Because all spaces in the `GET` request must be URL-encoded, the query string (if present) should be included in the second member of the array, along with the filename or application resource name. For `GET` requests, we go ahead and print this string as output and follow it with a new line:

```
  if ($requestElements[0] eq "GET" ) {
   print "$requestElements[0]  $requestElements[1]\n";
  }
```

For `POST` requests, we need to do a bit more processing. Specifically, we parse the same two data elements we parsed for the `GET` requests (except here the method should be equal to `POST`), but we also have to parse out the `POST` data string from the body of the request. Based on our log file format, the `POST` data string should be the second-to-last data element in the `@logEntryLines` array (this is the array that

contains each line of the specific log entry we are parsing). Then we append the POST data to the resource name as though it were a query string, and we print the line:

```
# POST request data is appended after the question mark
if ($requestElements[0] eq "POST" ) {
 print $requestElements[0]." ".$requestElements[1]."?".$logEntryLines[-2]."\n";
}
```

Finally, we close our if and for statements, and the script exits:

```
 } # End check for GET or POST
} # End loop for input file entries
```

Now we can use our *parseLog.pl* script to print out a listing of test request data in a very simple and consistent format. The complete *parseLog.pl* code is included at the end of this chapter.

Building the Scanner

Next we begin crafting the code for our scanner. The first thing we need to do is open our script and set up our command-line options. We use the Getopt::Std Perl module to parse the three command-line options outlined in Table 8-2.

Table 8-2. simpleScanner.pl options

Switch	Argument	Description
-c	Cookie data string	Use these HTTP cookies for all test requests.
-o	Filename	Log all output to this filename.
-v	N/A	Generate verbose output.

We also need to check whether at least two arguments have been passed to the script (the two mandatory arguments of the input filename and hostname). If two arguments have not been passed, the script dies and prints out some basic syntax info:

```
#!/usr/bin/perl

use LWP::UserAgent;
use strict;
use Getopt::Std;

my %args;
getopts('c:o:v', \%args);

printReport("\n** Simple Web Application Scanner **\n");

unless (@ARGV)  {
 die "\nsimpleScanner.pl [-o <file>] [-c <cookie data>] [-v] inputfile http://
hostname\n\n-c: Use HTTP Cookie\n-o: Output File\n-v: Be Verbose\n";
}
```

Notice in the preceding code that we already called a custom subroutine, `printReport`. This subroutine is an extremely simple routine for printing output to the screen and/or log file. Let's jump down and take a look at it.

Printing Output

We have developed a custom subroutine that our script uses for printing output. We have done this because we have a command-line option (`-o`) that allows all output to be sent to an output file, so we can send everything through one subroutine that handles output to both the screen and a file, if necessary.

printReport subroutine

As we just mentioned, we use the `printReport` subroutine to manage the printing of output to both the screen and output file, if necessary. Let's take a quick look at this routine's contents:

```
sub printReport {
 my ($printData) = @_;
 if ($args{o}) {
  open(REPORT, ">>", $args{o}) or die "ERROR => Can't write to file $args{o}\n";
  print REPORT $printData;
  close(REPORT);
 }
 print $printData;
}
```

As we mentioned, this routine is pretty simple. It takes one parameter as input (the data to be printed), appends the data to a file if the user specified the `-o` option (`$args{o}`), and prints the data to the screen. If the script cannot open the log file for writing, it dies and prints the error to the screen. Now all we have to do when we want to print something is send it to `printReport`, and we know it ends up printing in the right place(s). Now that we have finished the first subroutine, let's go back to the main body of the script.

Parsing the Input File

If we have made it this far in the execution cycle, we know the user has provided two arguments, so we assume the first one is the input file and we attempt to open it. If the open fails, the script dies and prints the error to the screen. If the open succeeds, we populate the `@requestArray` array with the contents of the input file:

```
# Open input file
open(IN, "<", $ARGV[0]) or die"ERROR => Can't open file $ARGV[0].\n";
my @requestArray = <IN>;
```

Now that we have opened our input file, the `@requestArray` array contains all the requests that were extracted from the input file. At this point, we can begin to process each request in the array by performing a `foreach` loop on the array members.

We use the following request for all our examples:

```
GET /public/content/jsp/news.jsp?id=2&view=F
```

At this point in the script, we also declare a few other variables: specifically, `$oResponse` and `$oStatus` (the response content and status code generated by our request), and two hashes for storing a log of all directory- and parameter-based test combinations we perform. We use the log hashes primarily to ensure that we do not make duplicate test requests (we discuss this in greater detail later in the chapter). As we perform each loop, we assign the original request from the input file to the `$oRequest` variable:

```
my ($oRequest,$oResponse, $oStatus, %dirLog, %paramLog);
printReport("\n** Beginning Scan **\n\n");

# Loop through each of the input file requests
foreach $oRequest (@requestArray) {
```

Once we start the loop, the first thing we do is to remove any line-break characters from the input entry and ensure that we are dealing with a `GET` or a `POST` request; otherwise, there is no need to continue. Although every line in our input file should contain only one of these two request types, because we are accepting an external input file we need to validate this fact:

```
# Remove line breaks and carriage returns
 $oRequest =~ s/\n|\r//g;

 # Only process GETs and POSTs
 if ($oRequest =~ /^(GET|POST)/) {
```

Next, we determine whether the request contains input parameters (either in the query string of a `GET` request or in a `POST` request) by inspecting the line for the presence of a question mark (?). If we find one, we need to parse the parameters and perform input parameter testing; otherwise, we skip parameter testing and move directly to directory testing:

```
 # Check for request data
 if ($oRequest =~ /\?/) {
```

For requests that contain parameter data, we perform parameter-based testing to identify a couple of common input-based vulnerabilities. Within the parameter-based testing block, the first action we perform on the request is to replay the original request (without altering any data):

```
  # Issue the original request for reference purposes
  ($oStatus, $oResponse) = makeRequest($oRequest);
```

The reason we do this, although perhaps not immediately obvious, is quite simple. Our scanning tool is testing the application based on a series of specific "test

requests" made to the application. The responses generated by each test request are analyzed for particular signatures indicating whether the specific vulnerability we are testing for is present. Because our findings are based on the output generated by each test request, we must be sure the presence of the vulnerability signature we are using is a direct result of our test request and not merely an attribute of a normal response.

For example, let's say we are looking for the string `SQL Server` in the test response to identify the presence of a database error message. However, the page we are testing contains a product description for software that is "designed to integrate with SQL Server." If we aren't careful, we might mistakenly identify this page as being vulnerable simply because the string `SQL Server` was contained in every response. To mitigate this risk, we preserve the original "valid" responses for each page before we begin our testing to validate that our signature matches are a result of the test we are performing and not a result of the scenario just described. This helps to ensure that we do not report false positives based on the content of the page or application we are testing.

Making an HTTP Request

This brings us to our next subroutine, `makeRequest`, which is responsible for making the actual requests during our scanning. As you can see in the last piece of code, the `makeRequest` subroutine is called to make the request, and it returns two variables (the status code and the response content). Let's jump down to this subroutine and take a closer look at exactly what is happening.

makeRequest subroutine

This subroutine is used to make each request we want to generate while testing the application. Keep in mind that this routine is not responsible for manipulating the request for testing purposes; it merely accepts a request and returns the response. Manipulating data for testing occurs outside of this subroutine, depending on the test being performed.

We need to consider several things here, specifically the inputs and outputs of the routine. Because we have already developed a fairly simple and consistent format for storing requests in our input file, it makes sense to pass off requests to this routine using the same syntax. As such, this subroutine expects one variable to be passed to it that contains an HTTP request in the same format as our input log entries. The output requirements for this routine will directly depend on the information we need to identify, regardless of whether the test is successful. At a minimum, the request body (typically HTML) is returned so that we can analyze the contents of the response output. In addition to the response body, we need to check the status code returned by the server to determine whether certain tests resulted on success or failure.

Another feature we discussed earlier was the ability for our scanner to use HTTP cookies when making test requests. Most web applications use HTTP cookies as a means of authenticating requests once the user has logged in (using a Session ID, for example). To effectively test the application, our tool needs to send these cookie(s) with each test request. To keep things simple, we assume these cookie values remain static throughout the testing session.

Now we can take a close look at this subroutine. The first thing it does is declare some variables and accept one input variable (the request):

```
sub makeRequest {
 my ($request, $lwp, $method, $uri, $data, $req, $status, $content);

 ($request)=@_;
 if ($args{v}) {
  printReport("Making Request: $request\n");
 } else {
  print ".";
 }
```

You can see we are also printing some output based on the presence of the `-v` (verbose) option. Note, however, that for nonverbose output we are using `print` instead of `printReport`. This is because we are printing consecutive periods (`.`) to the screen each time a request is made to indicate the script's progress during nonverbose execution. Although we want the verbose message to appear in the output file, we do not want these periods to appear there. Next, we set up a new instance of LWP to make the HTTP request:

```
 # Setup LWP UserAgent
 $lwp = LWP::UserAgent->new(env_proxy => 1,
             keep_alive => 1,
             timeout => 30,
             );
```

Now we need to parse the request data. Because we plan on performing upload testing via the HTTP `PUT` method, we need to support the `GET`, `POST`, and `PUT` methods. Both the `POST` and `PUT` methods need to pass some data in the body of the request, and as such, we need to perform a bit more processing for these two request methods. First, we split the input variable (`$request`) on the first space to parse out the method (`$method`) from the actual request data (`$uri`). For the `POST` and `PUT` requests, we can go ahead and parse out the data portion of the request (`$data`) as well by splitting the `$uri` variable based on a question mark:

```
# Method should always precede the request with a space
 ($method, $uri) = split(/ /, $request);

 # PUTS and POSTS should have data appended to the request
 if (($method eq "POST") || ($method eq "PUT")) {
  ($uri, $data) = split(/\?/, $uri);
 }
```

Now that we have our essential request data parsed into separate variables, we can set up the actual HTTP request. We know the hostname and cookie values being used for testing are available via the `$ARGV[1]` and `$args{c}` values, respectively (both of these are provided as inputs to the script). You'll notice here that we manually add our own custom "cookie" header value only if the `$args{c}` variable is populated because this is an optional switch. Although LWP does have an additional module designed specifically for handling HTTP cookies (`LWP::Cookies`), we don't really need the robust level of functionality this module provides because our cookie values remain static across all test requests.

```
# Append the uri to the hostname and set up the request
$req = new HTTP::Request $method => $ARGV[1].$uri;

# Add request content for POST and PUTs
if ($data) {
 $req->content_type('application/x-www-form-urlencoded');
 $req->content($data);
}

# If cookies are defined, add a Cookie: header
if ($args{c}) {
 $req->header(Cookie => $args{c});
}
```

Now that the request has been constructed, we pass it to LWP and parse the response that is sent back. We already decided the two pieces of the response we are most interested in are the status code and the response content, so we extract those two pieces of the response and assign them to the `$status` and `$content` variables accordingly:

```
my $response = $lwp->request($req);

# Extract the HTTP status code and HTML content from the response
$status = $response->status_line;
$content = $response->content;
```

It should be noted that the hostname or IP address (`$ARGV[1]`) supplied to LWP *must* be preceded with `http://` or `https://` and can optionally be followed by a nonstandard port number appended with a colon (i.e., *http://www.myhost.com:81*). Note in the next and final piece of this subroutine that we check for a `400` response status code. LWP returns a `400` (Bad Request) response when it is passed an invalid URL, so this response likely indicates the user did not supply a well-formed hostname. If this error occurs, the script dies and prints the error to the screen. Provided this is not the case, we return the `$status` and `$content` variables and close the subroutine:

```
 if ($status =~ /^400/) {
  die "Error: Invalid URL or HostName\n\n";
 }
 return ($status, $content);
}
```

As you can see, the routine accepts one input parameter, the request, and returns two output parameters, the response status code and the response content.

Parameter-Based Testing

Now let's go back to where we left off before we dove into makeRequest. You recall that we had just started our loop through the input file requests and had checked to see if the requests contained parameters. Now that we have replayed the original unaltered request, let's start dicing up the input file entry and generate our parameter-based test requests. Because we are within the if statement that checks for the presence of request parameters, we know any request that hits this area of the code has input parameters. As such, we perform a split on the first question mark to separate the data from the method and resource name. We assign the method and resource name (typically a web server script or file) to the $methodAndFile variable and the parameter data to the $reqData variable:

```
#Populate methodAndFile and reqData variables
my ($methodAndFile, $reqData) = split(/\?/, $oRequest, 2);
```

Next, we split the $reqData variable into an array based on an ampersand (&). Because this character is used to join parameter name/value pairs, we should be left with an array containing each parameter name/value pair:

```
my @reqParams = split(/\&/, $reqData);
```

Now that @reqParams is populated with our parameter name/value pairs, we are ready to start testing individual parameters. For efficiency, our scanner tests only unique page/parameter combinations that have not yet been tested. This is important if we have a large application that makes multiple requests to a common page throughout a user's session using the same parameters. As such, the first thing we do is craft a log entry for %paramLog and add it to the hash. Because we are interested in only the page and parameter names, and not the parameter values, we loop through the parameter name/value pairs and add only the parameter name(s) to our log entry ($pLogEntry):

```
my $pLogEntry = $methodAndFile;

# Build parameter log entry
my $parameter;
foreach $parameter (@reqParams) {
 my ($pName) = split("=", $parameter);
 $pLogEntry .= "+".$pName;
}
$paramLog{$pLogEntry}++;
```

Notice that in the last line of the preceding code, we are incrementing the value of the %paramLog hash member. If the hash member does not exist, it is added with a value of 1. If a subsequent page/parameter combination is identical, the value is incremented to 2, and so forth. To ensure that no duplicate requests are made, we test this page/parameter combination only if the log entry is equal to 1. Table 8-3 shows the current value of $pLogEntry and other key variables at this point in the script.

Table 8-3. Variable and array values

Variable/array	Value(s)
$oRequest	GET /public/content/jsp/news.jsp?id=2 &view=F
$methodAndFile	GET /public/content/jsp/news.jsp
$reqData	id=2&view=F
@reqParams	id=2 view=F
$pLogEntry	GET /public/content/jsp/news.jsp+id+view

Once we verify that the page/parameter combination has not already been tested, we must perform two nested loops through the @reqparams array. The first loop cycles through and tests each parameter. The second loop loops through the parameter/value list and reassembles it back into a query string while replacing the value of the parameter to be tested with a placeholder value. We use the counter variable from the first loop to determine the current array member to be altered in the second loop.

We use the placeholder string "--PLACEHOLDER--" in the parameter to be tested because we have more than one input validation test to perform. This allows our individual testing routines to substitute the placeholder based on their individual testing needs. At the end of each inner loop we can call the input validation testing routines. We also chop the last character off of the request because it always consists of an unnecessary ampersand (&):

```
if ($paramLog{$pLogEntry} eq 1) {

# Loop to perform test on each parameter
for (my $i = 0; $i <= $#reqParams; $i++) {
 my $testData;

 # Loop to reassemble the request parameters
 for (my $j = 0; $j <= $#reqParams; $j++) {
  if ($j == $i) {
   my ($varName, $varValue) = split("=",$reqParams[$j],2);
   $testData .= $varName."="."---PLACEHOLDER---"."&";
  } else {
   $testData .= $reqParams[$j]."&";
  }
 }

 # Remove the extra &
 chop($testData);
 my $paramRequest = $methodAndFile."?".$testData;

 ## Perform input validation tests
```

At this point in our loop, we can insert the individual input parameter testing routines we want to perform. As you can see, we have one test request for each request parameter, and we have replaced the parameter value to be tested with our placeholder.

Values Assigned to $testData

For our sample request, any code placed here executes twice with the following two values assigned to the `$testData` variable:

```
id=--PLACEHOLDER--&view=F
id=2&view=--PLACEHOLDER--
```

Now that we have our parameter parsing logic in place, we can call whichever specific input validation tests we want to perform. The first of these tests, called `sqlTest`, detects potential SQL injection points. This subroutine accepts one variable (the request to be used for testing) and returns `1` if the test detects a potential vulnerability or `0` if no vulnerability is detected. We assign the output of `sqlTest` (the `0` or `1`) to a variable called `$sqlVuln`:

```
my $sqlVuln = sqlTest($paramRequest);
```

sqlTest subroutine

Before we start building the SQL injection testing routine, we must decide what the test should consist of. The most common technique for SQL injection testing involves the use of a single quote (`'`) character inserted into a parameter value. In the absence of any input validation, a single quote, when passed to a database server within a query, typically generates an SQL syntax error unless it is properly escaped. The ability to invoke a database syntax error by inserting a single quote into an application parameter is a very good indication that an SQL injection point might exist. From a testing perspective, any database error message that the user can invoke is something that should be followed up on. As such, our SQL injection test consists of passing a single quote within the parameter being tested to see if the application returns a database error.

Recall that the specific parameter value to be tested in each request is prepopulated with a placeholder string before the parameter parsing logic calls the test routine. This saves us some effort because the subroutine automatically knows which parameter value to test based on the presence of the placeholder string. The first thing this subroutine does is accept an input variable (the request) and substitute the placeholder string with our SQL injection string. Because all we need to do is to pass in a single quote, our test string can be something simple, such as `te'st`:

```
sub sqlTest {
 my ($sqlRequest, $sqlStatus, $sqlResults, $sqlVulnerable);
 ($sqlRequest) = @_;

 # Replace the "---PLACEHOLDER---" string with our test string
 $sqlRequest =~ s/---PLACEHOLDER---/te'st/;
```

Now that the SQL injection test request is ready, we can hand it off to the makeRequest subroutine and inspect the response. We must define the criteria used to determine whether the response indicates the presence of a vulnerability. We previously decided that the ability to invoke a database error message using our test string is a good indicator that a potential injection point might exist. As such, the easiest way to test the response is to develop a regular expression designed to identify common database errors. We must ensure that the regular expression can identify database error messages from a variety of common database servers. Figure 8-3 shows what one of these error messages typically looks like.

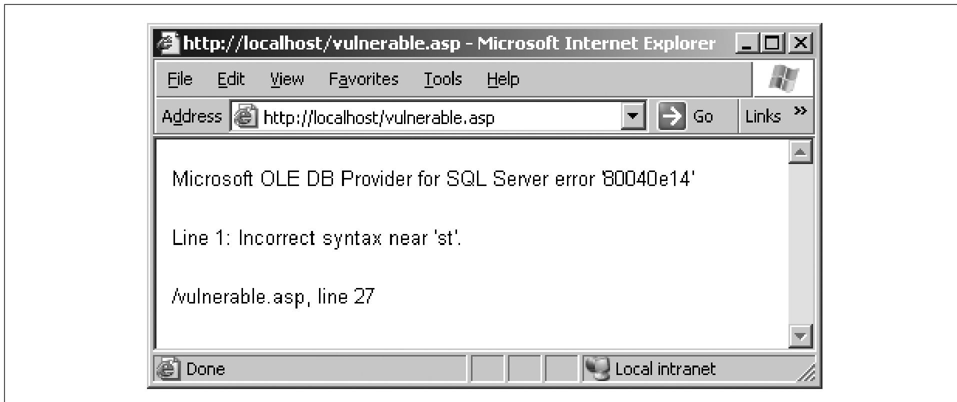


Figure 8-3. Common SQL server error message

The regular expression used in the following code was designed to match common database server error messages. As you can see, if the response matches our regular expression, we consider the page vulnerable and report the finding:

```
 # Make the request and get the response data
 ($sqlStatus, $sqlResults) = makeRequest($sqlRequest);

 # Check to see if the output matches our vulnerability signature.
 my $sqlRegEx = qr /(OLE DB|SQL Server|Incorrect Syntax|ODBC Driver|ORA-|SQL command
not|Oracle Error Code|CFQUERY|MySQL|Sybase| DB2 |Pervasive|Microsoft Access|MySQL|CLI
Driver|The string constant beginning with|does not have an ending string
delimiter|JET Database Engine error)/i;
 if (($sqlResults =~ $sqlRegEx) && ($oResponse !~ $sqlRegEx)) {
  $sqlVulnerable = 1;
 printReport("\n\nALERT: Database Error Message Detected:\n=> $sqlRequest\n\n");
 } else {
  $sqlVulnerable = 0;
 }
```

Additionally, note that we are also ensuring that the original response, made before we started testing (the $oResponse variable), does not match our regular expression. This helps to reduce the likelihood of reporting a false positive, because the normal

request content matches our regular expression (recall the scenario involving the product description page for software "designed to integrate with SQL Server").

Now that we have performed our test, we assign a value to the `$sqlVulnerable` variable to indicate whether the request detected a database error message. The final action for our subroutine is to return this variable. Returning `1` indicates that the request is potentially vulnerable; `0` indicates it is not:

```
 # Return the test result indicator
 return $sqlVulnerable;
}
```

Now that our SQL injection testing has been performed, we continue with our per-variable tests. Turning back to our main script routine, you'll recall we are in the midst of looping through each request variable, so we must perform the remaining per-variable tests before we continue. The next and last per-variable test to be performed is designed to detect possible XSS exposures. The subroutine for this test is called `xssTest` and it is structured in a way that is very similar to `sqlTest`. As before, we declare a new variable (`$xssVuln`) to assign the value returned (`0` or `1`) by `xssTest`:

```
    my $xssVuln = xssTest($paramRequest);
```

xssTest subroutine

To test for XSS, we inject a test string containing JavaScript into every test variable and check to see if the string gets returned in the HTTP response. A simple JavaScript alert such as the one shown here produces an easily visible result in the web browser if successful:

```
<script>alert('Vulnerable');</script>
```

One thing we must consider is that many XSS exposures result from HTML form fields that are populated with request parameter values. These values are typically embedded within an existing HTML form control, so any effective exploit string needs to "break out" of the existing HTML tag. To compensate for this, we modify our test string as follows:

```
"><script>alert('Vulnerable');</script>
```

Now that we have designed our test string, we can build the XSS testing routine. Like the other parameter test routines, it accepts a request containing a placeholder that must be replaced by our test string:

```
sub xssTest {
 my ($xssRequest, $xssStatus, $xssResults, $xssVulnerable);
 ($xssRequest) = @_;

 # Replace the "---PLACEHOLDER---" string with our test string
 $xssRequest =~ s/---PLACEHOLDER---/"><script>alert('Vulnerable');<\/script>/;
 # Make the request and get the response data
 ($xssStatus, $xssResults) = makeRequest($xssRequest);
```

Once again, we hand off the test request to makeRequest and inspect the HTTP response data for the presence of our test string. If the application returns the entire string (unencoded), an exploitable XSS vulnerability is likely to be present. If that is the case we assign a value of 1 to the $xssVulnerable variable and report the finding; otherwise, we set it to 0:

```
 # Check to see if the output matches our vulnerability signature.
 if ($xssResults =~ /"><script>alert\('Vulnerable'\);<\/script>/i) {
  $xssVulnerable = 1;

  # If vulnerable, print something to the user
  printReport("\n\nALERT: Cross-Site Scripting Vulnerability Detected:\n=>
$xssRequest\n\n");
 } else {
  $xssVulnerable = 0;
 }
```

Note that for this test, we did not check to see whether the original response contained our test string. This is because we want to flag any page that contains this test string because there is a chance it could be the result of a previous test request made by our scanner. Additionally, unlike the SQL injection test, the odds of generating a false hit using this string are fairly low.

Now that we have performed our test, the final action for our subroutine is to return the value of $xssVulnerable. Returning 1 indicates that the request is vulnerable; 0 indicates it is not:

```
 # Return the test results
 return $xssVulnerable;
}
```

Turning back to our main script routine, we now have completed all our parameter-based testing for the current request. We can close out the loop for each parameter value, as well as the if statements checking for unique parameter combos and request data:

```
   } # End of loop for each request parameter
  } # End if statement for unique parameter combos
 } # Close if statement checking for request data
```

Directory-Based Testing

Now it's time to move on to directory-based testing. You'll recall that we had previously determined the scanner tests would consist of parameter-based and directory-based testing routines. To perform directory-based testing, we must develop some logic that loops through each directory level within the test request and calls the appropriate testing subroutines at each level. Because we want to test every directory regardless of its content, we do not discriminate against any attributes of the test request (i.e., request method, presence of parameter data, etc.).

The first thing we do is isolate the path and file information from the rest of the test entry. Specifically, we strip out the request method at the beginning of the current test request (`$oRequest`) and any parameter data appended to it. For simplicity, we declare a trash variable (`$trash`) for allocating unnecessary data and keep the portion of the test request to be used in the `$oRequest` variable:

```
my $trash;
($trash, $oRequest, $trash) = split(/\ |\?/, $oRequest);
```

Now that we have isolated our path and file data, we create an array containing each directory and subdirectory from the `$oRequest` variable. We can do this by performing a split using a forward slash (`/`):

```
my @directories = split(m{/}, $oRequest);
```

Before we start looping through each directory level, we need to determine whether the last member of our `@directories` array is a filename. If the request was to a directory containing a default web server document, there is a good chance the request won't contain a filename. It is also likely that most of our requests will, in fact, contain a filename, so we need to determine this up front so that we do not confuse the two.

Because most web servers require a trailing forward slash (`/`) when making a request to a directory with no document, we can check the last character in the test request to see if it is a forward slash. If it is, we know no filename is in the request. If it is not, we assume the last portion of the request includes a file or servlet name, and this value is the last member of our `@directories` array. To check the last character, we break out each character in the request to an array (`@checkSlash`) and refer to the last member of the array:

```
my @checkSlash = split(//, $oRequest);
 my $totalDirs = $#directories;

 # Start looping through each directory level
 for (my $d = 0; $d <= $totalDirs; $d++) {
  if ((($checkSlash[(-1)] ne "/") && ($d == 0)) || ($d != 0)) {
   pop(@directories);
  }
```

As you can see in the preceding code, we assign the member count from the `@directories` array to the `$totalDirs` variable, then we perform a loop starting with a counter variable (`$d`) at `0` and continually increment the counter by 1 until it and the `$totalDirs` variable are equal. Each time we loop, we remove the last member of the `@directories` array, effectively truncating up one level every time. The exception to this is on the first loop (`$d = 0`), where the last member of the `$checkSlash` array is equal to a forward slash (`/`). This condition indicates that the test request did not contain a filename (the request ended with a forward slash), thus the last member is not removed. Subsequent requests (`$d != 0`), however, always result in the removal of the last array member. We assigned the member count from the `@directories` array to the `$totalDirs` variable because this number changes after each loop iteration.

Now that we have our directory truncation loop in place, we can create the actual request to be used by our testing subroutines. We are not particularly interested in the original request method, so we reassemble the current members of the `@directories` array into a `GET` request as follows:

```
my $dirRequest = "GET ".join("/", @directories)."\/";
```

At this point in the loop, we can insert the individual directory testing routines we want to perform. For our sample request, any code placed here is hit three times, with the values in Example 8-7 assigned to the `$dirRequest` variable.

Example 8-7. Values assigned to $dirRequest

```
GET /public/content/jsp/
GET /public/content/
GET /public/
```

As you can see, we have one test request for each directory level. Just as we did with the parameter-based test requests, we keep track of each request we make to ensure that we do not make duplicate requests. We had previously declared the `%dirLog` hash with this specific purpose in mind, so we can use the same technique we used with `%paramLog` to determine if the request is unique:

```
# Add directory log entry
$dirLog{$dirRequest}++;
if ($dirLog{$dirRequest} eq 1) {
```

Now we call whichever specific directory-based tests we want to perform. The first of these testing subroutines, `dirList`, is used to detect whether directory listings are permitted when requesting the directory without a document:

```
my $dListVuln = dirList($dirRequest);
```

Let's jump down and take a peek at the `dirList` subroutine.

dirList subroutine

Because this subroutine is called once at each directory level, it accepts a request that is already properly formed with no default document. This makes this routine relatively simple because all it needs to do is make the request and decide whether the response contains a directory listing:

```
sub dirList {
 my ($dirRequest, $dirStatus, $dirResults, $dirVulnerable);
 ($dirRequest) = @_;

 # Make the request and get the response data
 ($dirStatus, $dirResults) = makeRequest($dirRequest);

 # Check to see if it looks like a listing
 if ($dirResults =~ /(<TITLE>Index of \/|(<h1>|<title>)Directory Listing For|<title>
Directory of|\"\?N=D\"|\"\?S=A\"|\"\?M=A\"|\"\?D=A\"| - \/<\/title>|&lt;dir&gt;| - \/
<\/H1><hr>|\[To Parent Directory\])/i) {
  $dirVulnerable = 1;
```

```
# If vulnerable, print something to the user
 printReport("\n\nALERT: Directory Listing Detected:\n=> $dirRequest\n\n");
} else {
 $dirVulnerable = 0;
}
```

The regular expression used in the preceding code was designed to detect IIS, Apache, and Tomcat directory listings. As with the other testing routines, we assign a value of `1` to the `$dirVulnerable` variable and report the finding if the expression matches; otherwise, we assign a `0` to the variable. Finally, we return this value and close the subroutine:

```
 # Return the test results.
 return $dirVulnerable;
}
```

Let's jump back up to our main script routine and move on to our next and final testing subroutine, `dirPut`, to determine if the directory permits uploading of files using the HTTP `PUT` method:

```
 my $dPutVuln = dirPut($dirRequest);
```

dirPut subroutine

The last of our testing routines is responsible for determining whether files can be uploaded using the HTTP `PUT` method. Like `dirList`, this subroutine accepts a request that is already properly formed with no default document:

```
sub dirPut {
 my ($putRequest, $putStatus, $putResults, $putVulnerable);
 ($putRequest) = @_;
```

Unlike the `dirList` routine, we need to format our request a bit more before handing it off to `makeRequest`. Specifically, we need to change the request method from `GET` to `PUT`, and add request data to the end of the request. Once we have done that we issue the request:

```
 # Format the test request to upload the file
 $putRequest =~ s/^GET/PUT/;
 $putRequest .= "uploadTest.txt?ThisIsATest";

 # Make the request and get the response data
 ($putStatus, $putResults) = makeRequest($putRequest);
```

Now that we have issued the `PUT` request we reformat the request to check whether the new document is in the directory. The reformatting includes changing the request method back to `GET`, and removing the request parameter data:

```
 # Format the request to check for the new file
 $putRequest =~ s/^PUT/GET/;
 $putRequest =~ s/\?ThisIsATest//;

 # Check for the uploaded file
 ($putStatus, $putResults) = makeRequest($putRequest);
```

Once we issue the second request, we can check to see if our test string was returned in the content. If so, we can be sure the file was created successfully, so we set the `$dirVulnerable` variable to `1` and report the finding; otherwise, we set this variable to `0`:

```
if ($putResults =~ /ThisIsATest/) {
 $putVulnerable = 1;

# If vulnerable, print something to the user
 printReport("\n\nALERT: Writeable Directory Detected:\n=> $putRequest\n\n");
} else {
 $putVulnerable = 0;
}
```

Last but not least, we return the `$dirVulnerable` value and close the subroutine:

```
 # Return the test results.
 return $putVulnerable;
}
```

At this point, we have completed all our directory-level testing routines, so we jump back up to our main script routine and close out all our loops as follows:

```
   } # End check for unique directory
  } # End loop for each directory level
 } # End check for GET or POST request
} # End loop on each input file entry

printReport("\n\n** Scan Complete **\n\n");
```

Finally, we report a message stating that testing is complete. With that, we have completed our simple web application vulnerability scanner.

Using the Scanner

Hopefully, by now you are familiar enough with the scanner to know how to use it effectively. If not, let's quickly review the process of running the scanner against an application. We have already gone through the process of how to manually crawl and log data from a web application. Assuming we have the log file from the proxy server, we can call the *parseLog.pl* script to format the log data and redirect the script's output to our input file:

```
ParseLog.pl proxylog.txt > inputfile.txt
```

Next, assuming the application requires authentication, we need to reauthenticate to the application and intercept a request subsequent to successful authentication (we can use our Burp proxy server to do this by checking the Intercept box under Client Requests on the Options tab). The intercepted request should contain a fresh Session ID or authentication token for us to provide our script for testing. If the application is anonymously accessible and doesn't require state management, we can probably skip this step.

Before we actually begin testing an authenticated application, we also want to identify the login and logout requests within the input file and manually delete them. If we do not do this, the scanner will issue these requests during its execution, invalidating our Session ID or authentication token. Because of this issue, it's best that we test these pages manually.

Now we are ready to run the scanner. We pass the scanner our input filename and hostname to be tested, along with the -c option and including the HTTP cookie value(s) we want to use for testing:

```
simpleScanner.pl -c "ASPSESSIONIDQARRTRQC= FGCBFJBABN NLNLKNCLJBPBGE;" inputfile.txt
http://www.myhost.com
```

It's that simple. We can optionally use the -v option to have the script print each request it makes; otherwise, it notifies us only when it detects a vulnerability. Keep in mind that we have merely scratched the surface as far as the potential for identifying web application vulnerabilities goes. In addition to identifying these vulnerabilities, we could extend the scanner to perform automated attacks and/or exploits in the event that a vulnerability is detected. In the next chapter, we will look at some examples of how to do that using the simple scanner we just developed.

Complete Source Code

The rest of this chapter contains the complete source code for the two scripts developed and outlined in this chapter.

simpleScanner.pl

Example 8-8 shows the full source code for the *simpleScanner.pl* script.

Example 8-8. Code for simpleScanner.pl

```
#!/usr/bin/perl

use LWP::UserAgent;
use strict;
use Getopt::Std;

my %args;
getopts('c:o:v', \%args);

printReport("\n** Simple Web Application Scanner **\n");

if ($#ARGV < 1) {
 die "\n$0 [-o <file>] [-c <cookie data>] [-v] inputfile http://hostname\n\n-c: Use HTTP
Cookie\n-o: Output File\n-v: Be Verbose\n";
}
```

Example 8-8. Code for simpleScanner.pl (continued)

```
# Open input file
open(IN, "< $ARGV[0]") or die"ERROR => Can't open file $ARGV[0].\n";
my @requestArray = <IN>;

my ($oRequest,$oResponse, $oStatus, %dirLog, %paramLog);

printReport("\n** Beginning Scan **\n\n");

# Loop through each of the input file requests
foreach $oRequest (@requestArray) {

 # Remove line breaks and carriage returns
 $oRequest =~ s/(\n|\r)//g;

 # Only process GETs and POSTs
 if ($oRequest =~ /^(GET|POST)/) {

  # Check for request data
  if ($oRequest =~ /\?/) {

   # Issue the original request for reference purposes
   ($oStatus, $oResponse) = makeRequest($oRequest);

   #Populate methodAndFile and reqData variables
   my ($methodAndFile, $reqData) = split(/\?/, $oRequest, 2);
   my @reqParams = split(/\&/, $reqData);

   my $pLogEntry = $methodAndFile;

   # Build parameter log entry
   my $parameter;
   foreach $parameter (@reqParams) {
    my ($pName) = split("=", $parameter);
    $pLogEntry .= "+".$pName;
   }
   $paramLog{$pLogEntry}++;
   if ($paramLog{$pLogEntry} eq 1) {

    # Loop to perform test on each parameter
    for (my $i = 0; $i <= $#reqParams; $i++) {
     my $testData;

     # Loop to reassemble the request parameters
     for (my $j = 0; $j <= $#reqParams; $j++) {
      if ($j == $i) {
       my ($varName, $varValue) = split("=",$reqParams[$j],2);
       $testData .= $varName."="."---PLACEHOLDER---"."&";
      } else {
       $testData .= $reqParams[$j]."&";
      }
     }
     chop($testData);
```

Example 8-8. Code for simpleScanner.pl (continued)

```
     my $paramRequest = $methodAndFile."?".$testData;

     ## Perform input validation tests
     my $sqlVuln = sqlTest($paramRequest);
     my $xssVuln = xssTest($paramRequest);

    } # End of loop for each request parameter
   } # End if statement for unique parameter combos
  } # Close if statement checking for request data

  my $trash;
  ($trash, $oRequest, $trash) = split(/\ |\?/, $oRequest);
  my @directories = split(/\//, $oRequest);

  my @checkSlash = split(//, $oRequest);
  my $totalDirs = $#directories;

  # Start looping through each directory level
  for (my $d = 0; $d <= $totalDirs; $d++) {
   if ((($checkSlash[(-1)] ne "/") && ($d == 0)) || ($d != 0)) {
    pop(@directories);
   }

   my $dirRequest = "GET ".join("/", @directories)."\/";

   # Add directory log entry
   $dirLog{$dirRequest}++;
   if ($dirLog{$dirRequest} eq 1) {
    my $dListVuln = dirList($dirRequest);
    my $dPutVuln = dirPut($dirRequest);

   } # End check for unique directory
  } # End loop for each directory level
 } # End check for GET or POST request
} # End loop on each input file entry

printReport("\n\n** Scan Complete **\n\n");

sub dirPut {
 my ($putRequest, $putStatus, $putResults, $putVulnerable);
 ($putRequest) = @_;
 # Format the test request to upload the file
 $putRequest =~ s/^GET/PUT/;
 $putRequest .= "uploadTest.txt?ThisIsATest";

 # Make the request and get the response data
 ($putStatus, $putResults) = makeRequest($putRequest);
 # Format the request to check for the new file
 $putRequest =~ s/^PUT/GET/;
 $putRequest =~ s/\?ThisIsATest//;

 # Check for the uploaded file
```

Example 8-8. Code for simpleScanner.pl (continued)

```
 ($putStatus, $putResults) = makeRequest($putRequest);
 if ($putResults =~ /ThisIsATest/) {
  $putVulnerable = 1;

 # If vulnerable, print something to the user
  printReport("\n\nALERT: Writeable Directory Detected:\n=> $putRequest\n\n");
 } else {
  $putVulnerable = 0;
 }
 # Return the test results.
 return $putVulnerable;
}

sub dirList {
 my ($dirRequest, $dirStatus, $dirResults, $dirVulnerable);
 ($dirRequest) = @_;

 # Make the request and get the response data
 ($dirStatus, $dirResults) = makeRequest($dirRequest);

 # Check to see if it looks like a listing
 if ($dirResults =~ /(<TITLE>Index of \/|(<h1>|<title>)Directory Listing For|<title>
Directory of|\"\?N=D\"|\"\?S=A\"|\"\?M=A\"|\"\?D=A\"| - \/<\/title>|&lt;dir&gt;| - \/<\/
H1><hr>|\[To Parent Directory\])/i) {
  $dirVulnerable = 1;

  # If vulnerable, print something to the user
  printReport("\n\nALERT: Directory Listing Detected:\n=> $dirRequest\n\n");
 } else {
  $dirVulnerable = 0;
 }
 # Return the test results.
 return $dirVulnerable;
}

sub xssTest {
 my ($xssRequest, $xssStatus, $xssResults, $xssVulnerable);
 ($xssRequest) = @_;

 # Replace the "---PLACEHOLDER---" string with our test string
 $xssRequest =~ s/---PLACEHOLDER---/"><script>alert('Vulnerable');<\/script>/;
 # Make the request and get the response data
 ($xssStatus, $xssResults) = makeRequest($xssRequest);

 # Check to see if the output matches our vulnerability signature.
 if ($xssResults =~ /"><script>alert\('Vulnerable'\);<\/script>/i) {
  $xssVulnerable = 1;

  # If vulnerable, print something to the user
  printReport("\n\nALERT: Cross-Site Scripting Vulnerability Detected:\n=> $xssRequest\n\
n");
 } else {
```

Example 8-8. Code for simpleScanner.pl (continued)

```
  $xssVulnerable = 0;
 }
 # Return the test results
 return $xssVulnerable;
}

sub sqlTest {
 my ($sqlRequest, $sqlStatus, $sqlResults, $sqlVulnerable);
 ($sqlRequest) = @_;

 # Replace the "---PLACEHOLDER---" string with our test string
 $sqlRequest =~ s/---PLACEHOLDER---/te'st/;
 # Make the request and get the response data
 ($sqlStatus, $sqlResults) = makeRequest($sqlRequest);

 # Check to see if the output matches our vulnerability signature.
 my $sqlRegEx = qr /(OLE DB|SQL Server|Incorrect Syntax|ODBC Driver|ORA-|SQL command
not|Oracle Error Code|CFQUERY|MySQL|Sybase| DB2 |Pervasive|Microsoft Access|MySQL|CLI
Driver|The string constant beginning with|does not have an ending string delimiter|JET
Database Engine error)/i;
 if (($sqlResults =~ $sqlRegEx) && ($oResponse !~ $sqlRegEx)) {
  $sqlVulnerable = 1;
  printReport("\n\nALERT: Database Error Message Detected:\n=> $sqlRequest\n\n");
 } else {
  $sqlVulnerable = 0;
 }
 # Return the test result indicator
 return $sqlVulnerable;
}

sub makeRequest {
 my ($request, $lwp, $method, $uri, $data, $req, $status, $content);

 ($request)=@_;
 if ($args{v}) {
  printReport("Making Request: $request\n");
 } else {
  print ".";
 }

 # Setup LWP UserAgent
 $lwp = LWP::UserAgent->new(env_proxy => 1,
             keep_alive => 1,
             timeout => 30,
             );
 # Method should always precede the request with a space
 ($method, $uri) = split(/ /, $request);

 # PUTS and POSTS should have data appended to the request
 if (($method eq "POST") || ($method eq "PUT")) {
  ($uri, $data) = split(/\?/, $uri);
 }
```

Example 8-8. Code for simpleScanner.pl (continued)

```
 # Append the URI to the hostname and setup the request
 $req = new HTTP::Request $method => $ARGV[1].$uri;

 # Add request content for POST and PUTS
 if ($data) {
  $req->content_type('application/x-www-form-urlencoded');
  $req->content($data);
 }

 # If cookies are defined, add a COOKIE header
 if ($args{c}) {
  $req->header(Cookie => $args{c});
 }
 my $response = $lwp->request($req);

 # Extract the HTTP status code and HTML content from the response
 $status = $response->status_line;
 $content = $response->content;
 if ($status =~ /^400/) {
  die "Error: Invalid URL or HostName\n\n";
 }
 return ($status, $content);
}

sub printReport {
 my ($printData) = @_;
 if ($args{o}) {
  open(REPORT, ">>$args{o}") or die "ERROR => Can't write to file $args{o}\n";
  print REPORT $printData;
  close(REPORT);
 }
 print $printData;
}
```

parseLog.pl

Example 8-9 contains the full source for the *parseLog.pl* script.

Example 8-9. Code for parseLog.pl

```
#!/usr/bin/perl

use strict;

if ($#ARGV < 0) {
 die "Usage: $0 LogFile\n";
}

open(IN, "< $ARGV[0]") or die"ERROR: Can't open file $ARGV[0].\n";

# Change the input record separator to select entire log entries
$/ = "=" x 54;
```

Example 8-9. Code for parseLog.pl (continued)

```
my @logData = <IN>;

# Loop through each request and parse it
my ($request,$logEntry, @requests);
foreach $logEntry (@logData) {

 # Create an array containing each line of the raw request
 my @logEntryLines = split(/\n/, $logEntry);

 # Create an array containing each element of the first request line
 my @requestElements = split(/ /, $logEntryLines[1]);

 # Only parse GET and POST requests
 if ($requestElements[0] eq "GET" || $requestElements[0] eq "POST" ) {
  if ($requestElements[0] eq "GET" ) {
   print $requestElements[0]." ".$requestElements[1]."\n";
  }

  # POST request data is appended after the question mark
  if ($requestElements[0] eq "POST" ) {
   print $requestElements[0]." ".$requestElements[1]."?".$logEntryLines[-2]."\n";
  }
 } # End check for GET or POST
} # End loop for input file entries
```