Chapter 1

Web Hacking & Penetration Methodologies

IN THIS CHAPTER:

- Threats and Vulnerabilities
- Profiling the Platform
- Profiling the Application
- Summary

The "revolution" part of the "Internet revolution" slogan has not been around nearly as long as the Internet itself, whose lineage dates back to the 1960s. While the beneficiaries of the revolution are debatable, the amount of information that has been put "on the Web" has obviously grown immensely. Today, anyone can post stories about their cat, write insightful articles, chat on message boards, sell widgets, sell used widgets, manage their collection of widgets, and more. One of the common factors among these activities is the use of web applications. Web applications may be static HTML files or complex, dynamic, and database-driven web sites. In all cases, security is paramount to maintaining the application's integrity, privacy of its users, confidentiality of its data, and uptime of its servers.

This chapter describes the techniques you can use to assess the (in)security of your application. It steps through the major categories of attacks employed by malicious Internet users. In some cases, the attack may appear innocuous, such as gathering line numbers from error messages or identifying all of the <form> fields in a web site. On the other hand, the attacker may find the chink in the application's armor that enables arbitrary access to database information. In all cases, a comprehensive review of a web application requires a methodical approach. Here is where you will find that approach.

THREATS AND VULNERABILITIES

There are two categories into which web vulnerabilities can be categorized. One category contains vulnerabilities within the platform—the components that many web applications share, such as Linux, Windows, Apache, and Oracle. The other category of vulnerabilities targets the application itself. In other words, programming errors in the web site might expose a user's credit card details, enable a malicious user to execute arbitrary database queries, or even enable remote command-line access to the server.

Consequently, any web application faces a variety of threats. Many tools are available to check for vulnerabilities in an operating system or web server, and exploit code for those vulnerabilities is common. Application attacks, such as SQL injection or session hijacking, are more difficult to automate, but the most common vulnerabilities can be codified so that a few lines of Perl can check for their presence, as in the case of basic input validation checks. In short, many high-risk vulnerabilities can be identified and exploited by the least competent of individuals. That is not to say that other high-risk vulnerabilities require an elite skill set; it merely points out that greatest common denominator of threats to a web application has a very large set of tools and information available.

PROFILING THE PLATFORM

A web application consists of more than a shopping cart, a marketing opt-out page, and a flashing graphic to capture your attention. The majority of e-commerce applications use a three-tier architecture. So, when we say "application" we really mean one or more servers that perform the following roles:

- **Web Server** This component serves web pages to the user's browser. Apache and IIS are the most common examples. Every web server has a collection of vulnerabilities.
- **Application Server** This component manipulates, interprets, and presents data for the user. The application server can be part of the web server, as in the case of PHP and Apache, or ASP.NET and IIS. On the other hand, the application server could be a physically separate server, such as a Tomcat servlet engine. Every web application server has a collection of vulnerabilities.
- **Database** This component stores all of the data required by the application. Whereas users interact with the web and application servers, they usually cannot access the database server. Most of the time, the application server brokers data between the user and the database, formatting data so that they are stored correctly. Every database server has a collection of vulnerabilities.

It may seem pedantic to repeat that each component has a potential security problem; however, it should illustrate the number of threats a web application faces—all before a single line of code has even been written!

Port Scanning and Service Identification

This is the basic step in a security review. After all, in order to test a system, there must be a service (open port) listening. There are several port scanners for Windows- and Unix-based operating systems that not only act as port scanners, but have quite a bit of extra functionality.

Nmap is probably the best-known port scanner. It compiles on just about all Unix operating systems and has recently been ported to the Windows platform.

```
[localhost:~]% nmap 192.168.0.43
Starting nmap V. 3.20 ( www.insecure.org/nmap/ )
Interesting ports on target (192.168.0.42):
(The 1596 ports scanned but not shown below are in
```

```
 state: closed)
Port       State       Service
22/tcp     open        ssh
80/tcp     open        http
Nmap run completed -- 1 IP address (1 host up) scanned
 in 0.481 seconds
```

Other uses for nmap include operating system identification, the ability to save output in different formats, and a wide range of different port scanning methods.

If you have trouble compiling nmap on Apple OSX, try passing the "--build=powerpc-apple-macosx" flag to the ./configure script.

Scanline is a Windows-based port scanner that, unlike nmap, does not require the installation of WinPCAP drivers. It is more basic than nmap, meaning that it only performs SYN, ICMP, and UDP scans, but it is extremely fast and especially reliable for UDP scans. One of its best features is the "banner" option (-b) that collects the service banner, if present, from each port it scans.

```
C:\>sl -bp -o website.sl 192.168.0.43
192.168.0.43
TCP ports: 80
UDP ports:
TCP 80:
[HTTP/1.0 200 OK Connection: Keep-Alive
 Date: Wed, 19 Mar 2003 00:18
:38 GMT Set-Cookie:]
```

Netcat is a cumbersome tool for port scanning, but extremely useful for banner grabbing. It will also make an appearance in Chapter 2 as a tool for application attacks. Banner grabbing with netcat is simple. Either connect to the target site and type in the http request or echo the request into netcat:

```
echo -e "GET / HTTP/1.0\n\n" | nc -vv website 80
```

We'll make more mention of this later on in the book, but it's important to realize that any http request can be piped through netcat. For example, a HEAD request doesn't return HTML source when all you're looking for is the server's banner. Also, some sites might respond differently to HTTP 1.1 or WebDAV requests.

```
echo -e "GET / HTTP/1.1\nHost:\n" | nc -vv website 80
```

The Windows command shell (cmd.exe) does not support a proper echo. You will have to create a nudge.txt file that contains:

```
GET / HTTP/1.0
<blank line>
<blank line>
```

and use the command:

```
c:\> type nudge.txt | nc -vv website 80
```

You can also use the Cygwin utility on Windows platforms to obtain a Unix-like echo.

Netcat works great for HTTP connections, but won't help when you need to gather information and connect to sites using HTTPS. In that case, use the openssl command to make connections:

```
[localhost:~]% echo -e "HEAD / HTTP/1.0\n\n" | \
 openssl s_client -quiet -connect 192.168.0.43:443
depth=0 /C=FR/ST=Paris/L=Paris/O=roliste/OU=jdr/CN= website
verify error:num=20:unable to get local issuer certificate
verify return:1
depth=0 /C=FR/ST=Paris/L=Paris/O=roliste/OU=jdr/CN=website
verify error:num=20:unable to get local issuer certificate
verify return:1
depth=0 /C=FR/ST=Paris/L=Paris/O=roliste/OU=jdr/CN=website
verify error:num=20:unable to get local issuer certificate
verify return:1
HTTP/1.1 302 Found
Date: Fri, 15 Nov 2002 08:43:17 GMT
Server: Stronghold/2.4.2 Apache/1.3.6 C2NetEU/2412 (Unix)
Location: http://www.website.com/
Connection: close
Content-Type: text/html; charset=iso-8859-1
```

OpenSSL can also be used to identify the encryption strength of the target web server.

```
openssl s_client -connect website:443 -cipher EXPORT40
openssl s_client -connect website:443 -cipher NULL
openssl s_client -connect website:443 -cipher HIGH
```

The idea is to use openssl to try and negotiate a downgraded session. In most cases, this should not work; however, you might run into an embedded device or legacy server that supports a very weak encryption scheme. If the server supports the selected encryption strength, then you will see the certificate information. Otherwise, you will receive an error similar to the following:

```
CONNECTED(00000003)
27249:error:14077410:SSL routines:SSL23_GET_SERVER_HELLO:
sslv3 alert handshake failure:s23_clnt.c:455:
```

If you're a fan of nessus, the ssl_ciphers.nes plug-in will perform the SSL strength check for you and report all of the server's supported algorithms.

Vulnerability Scanning

Vulnerability scanning is the trivial part of web application security testing. Anyone with a little knowledge of the command line can perform these checks.

Nikto is based on the libwhisker Perl library, which is an evolution of the Whisker web vulnerability scanner. As such, Nikto is a vulnerability checker that focuses on known vulnerabilities within web servers and CGI scripts. The list of known vulnerabilities is continuously maintained and the tool even allows for quick updates:

```
[localhost:~]%./nikto.pl –update
---------------------------------------------------------------
- Nikto v1.23  - www.cirt.net - Mon Mar 17 23:30:46 2002
+ No updates required.
+ www.cirt.net message: Please report bugs and new tests.
```

To use Nikto, point it at a web server and examine the output for HTTP 200 messages and other important notes.

```
[localhost:~] mike% ./nikto.pl -p 80 -host dusk
---------------------------------------------------------------
- Nikto v1.23  - www.cirt.net - Tue Mar 18 20:40:45 2003
---------------------------------------------------------------
+ Target IP:       192.168.0.175
+ Target Hostname: dusk
+ Target Port:     80
---------------------------------------------------------------
+ Server: Microsoft-IIS/5.0
+ No CGI Directories found (use -a to force check...)
+ /xxxxxxxxxxabcd.html - The IIS server may be vulnerable
 to Cross Site Scripting (XSS) in error messages, see
 MS02-018,CVE-2002-0075,SNS-49,MS02-018,CA-2002-09 (GET)
+ /_vti_bin/_vti_aut/author.dll?method=list+documents%3a
3%2e0%2e2%2e1706&service%5fname=&listHiddenDocs=true&
listExplorerDocs=true&listRecurse=false&listFiles=true&
listFolders=true&listLinkInfo=true&listIncludeParent=true&
listDerivedT=false&listBorders=false
 Needs Auth: (realm NTLM)
+ /_vti_inf.html - FrontPage may be installed. (GET)
- 1106 items checked, 3 items found on remote host
```

Nessus is a more complete tool than Nikto because it combines port scanning and vulnerability checking, not limited to web checks, into a

single application. Chapter 3 provides more detail and instructions on how to use these tools.

As you begin the application assessment, create a matrix similar to Table 1-1 to track the data you acquire.

PROFILING THE APPLICATION

The next step is to profile the actual web site by systematically cataloging all of its pages, functions, and parameters. This is where you'll be able to identify common problems such as poor input validation, inadequate session handling, and other programming errors. Consequently, it is important to maintain a descriptive record of the site. You will most likely uncover some obvious application-level vulnerabilities in this

Step (Repeat for Each Server)	Subsequent Steps and Potential Attacks
Identify the server's role	What is its function? (Web, application, database, firewall, proxy, administration) What data does it handle? With which servers does it interact? (For example, does the web server contact the database, or is there an intermediate application server?)
Determine the operating system and version	Identify the OS using banner information, educated guesses, and "nmap –O" results.
Determine the operating system and application patch level	Check the OS and application vendor's web site for the latest patch information.
Scan for open ports	Perform a TCP and UDP port scan Application server ports (7000, 8000, etc.) Administration ports (22, 23, 2301, 3389, 10000) Proxy ports (8080) Sytem ports (79, 111, 139, 445, 512)
Record the web server type, patch level, and additional components	Apache mod_* modules IIS ISAPI filters This information will be useful for finding known vulnerabilities, testing functionality (such as WebDAV), and searching for common HTML files.
Research known vulnerabilities	Good resources are packetstormsecurity.org and www.securityfocus.com. Application-level vulnerability information can be found at www.cgisecurity.com.

Table 1-1. Platform Profile Checklist

phase. Resist the urge to immediately branch off and begin hacking the application. Collect a complete picture. Then, take advantage of vulnerabilities to gather more information or gain additional access.

Complete a matrix similar to Table 1-3 as you visit each page of the application.

Enumerate the Directory Structure and Files

In one way, this step is trivial, easy to perform, and can be readily automated. After all, in order to profile the application you need to know what files make up the web site. The easy part is going through the application and recording each file name and its full path from the web root.

The other portion of directory enumeration involves making educated guesses about files or directories that *might* exist. To be successful at directory prognostication takes a little bit of luck and an eye for patterns. For example, perhaps the application has three directories from the root: /scripts, /users, and /manage. Now, if you observe /users/includes and /scripts/includes directories, then it's probably a good guess that there will also be a /manage/includes directory. Often, subdirectories have incorrect authorization settings. So, while /manage might be password protected, /manage/include is not.

A good example is Real Networks RealServer 7 web administration portal. There is an /admin directory that requires a username and password to access; however, files in the /admin/docs/ directory can be accessed directly—not a good situation when the default.cfg file in this directory contains at least one plaintext username and password to the site. This vulnerability also demonstrates that any web-based platform (server, application, or web engine) is susceptible to these types of vulnerabilities.

A tool such as wget or libwhisker's crawl function is helpful for this stage, but manual interaction gives you a better feel for how the programmers designed the application.

Always look for a robots.txt file. This file is intended to serve as a list of directories that search engines should *not* crawl. Thus, a robots.txt file (if present) provides a comprehensive list of directories on the server—especially directories that contain sensitive information that search engines are supposed to ignore.

Identify Authentication Mechanism

If the application supports individual users, then record how users must authenticate to the application:

Anonymous	No authentication required.
HTTP Basic	Username and password are passed in a header that is Base64 encoded of the type base64 (username:password).
HTTP Digest	Username and password are passed in a header that is an MD5 challenge/response.
HTTP NTLM	Username and password use Windows credentials passed in a challenge/response format.
Form-based	Username and password are entered in a form. The user receives some token (cookie value, session ID, etc.) that indicates success.

Keep in mind that challenge/response mechanisms do not protect passwords with 100 percent security. Even though the password is not sent between the client and server, the "hash" passed by the challenge/response is susceptible to brute force. So, any user authentication mechanism should also use an encrypted channel. In other words, use SSL regardless of how users' names and passwords are submitted to the application.

If you're interested in tools that break other challenge/response mechanisms, check out kerbcrack from http://www.ntsecurity.nu/ and anwrap.pl from http://modelm.org/anwrap/. Although these examples are not directly related to web applications, they illustrate the fallacy of relying on "one-hit wonder" algorithms or techniques for your network's security. This doesn't imply that they are totally insecure and useless, it just means that computer security is under continuous escalation.

Identify Authorization Mechanism

In an application that enforces a tiered user model, try to log in with accounts that have varying degrees of access. Compare what functions are available to different user roles. Also, record which tokens change based on user and role. Look at Table 1-2 for an example.

From this example, we have several attacks available to us.

User	URL
John	https://website/index.php?id=john&isadmin=false&menu=basic
Paul	https://website/index.php?id=paul&isadmin=false&menu=basic
George	https://website/index.php?id=george&isadmin=true&menu=full
Ringo	https://website/index.php?id=ringo&isadmin=true&menu=full

Table 1-2. Identify Authorization Tokens

- Log in as user John, then change the URL to

  https://website/index.php?id=**paul**&isadmin=false&menu=basic

  If the request succeeds, then the application is vulnerable to horizontal privilege escalation. A user can modify one token (id) in order to impersonate a peer. If John changes the URL to

  https://website/index.php?id=**george**&isadmin=false&menu=basic

  but doesn't receive administrator rights, then user impersonation still works, but the server tracks authorization in a parameter other than id.

  If John did receive administrator rights, then the application performs the authorization check based on the username, is vulnerable to horizontal and privilege escalation, and uses poor session management. A poor application, indeed!

- Log in as user John, then change the URL to

  https://website/index.php?id=john&isadmin=false&menu=**full**

  If the request succeeds, then the application is vulnerable to vertical privilege escalation. A user can modify one token (menu) in order to gain elevated rights. In this case, the application does not perform any authorization checks after the user has authenticated. It trusts that "menu=basic" will not be changed.

- Log in as user John, then change the URL to

  https://website/index.php?id=john&isadmin=**true**&menu=basic

  If the request succeeds, then the application is vulnerable to vertical privilege escalation. In this case, the application performs an authorization check on the isadmin parameter and provides functionality according to the value.

- Log in as user John, then change the URL to

  https://website/index.php?id=john&isadmin=**true**&menu=**full**

  If the request succeeds, then the application is vulnerable to vertical privilege escalation. The attack required manipulating multiple tokens, but the application still failed to enforce strong authorization checks.

Protect Authorization

Session management and its inherent authorization control is definitely the greatest challenge to a web application. The best defense is to track as many user attributes on the server as possible. For example, if the *isadmin* and *menu* parameters from the previous example had been tracked in a database and verified for each request, then the attacks might not have succeeded. Of course, creating role-based access in a custom database table increases application overhead and maintenance; however, the security requirements of the application may require such a technique. After all, speedy processors and computer hardware have become much more of a commodity. So, adding another five or ten servers to a web farm in order to keep up with user demand should have a better payoff than risking media headlines that include the words "credit card numbers stolen."

Identify All "Support" Files

Most of the time, support files can be identified, recorded, and ignored. Some examples of these files include style sheets (.css) and IIS files that are interpreted by specific ISAPI filters, such as .htr, .htx, .idc, and .idq. These files usually contain layout information or other browser-specific data, or contain a short list of application information. While there might be a buffer overflow against the ISAPI filter itself (.ida, for example), the files rarely contain values or data that can be exploited. Still, they should be reviewed for the presence of developers' comments.

On the other hand, support files such as global.asa and passwd.txt contain authentication credentials for the application. One of the most notorious support files is passwd.txt. As the name implies, it contains usernames and passwords, resides in the web document root (usually in / or /wwwboard), and its file suffix (txt) means that most web servers will happily let users view it in their web browser. Nikto will identify these common files, but only if they are in default locations.

Identify All Include Files

Include files are not usually explicitly called by the user's browser. Instead, they are references by pages that the user visits. For example, a login.asp file might call two include files: footer.inc and validateuser.inc. A user only sees a request for login.asp; both of the include files are called by a file on the web server and executed on the web server.

The easiest way to identify an include file is to search for the server side include (SSI) tag. There are two types of SSI references:

- **Virtual** The virtual SSI uses a path format that begins with the web document root.

  ```
  <!-- #include virtual = "/html/include/header.inc" -->
  ```

- **File** The file SSI uses a path format that is relative to the current directory.

  <!-- #include file = "include/header.inc" -->

In both cases, the SSI will be visible in the HTML source code. On the other hand, a language such as PHP references include files between language tags. Therefore, you'll have to try to find an /include directory and guess some common file names. Also, be sure to check HTML comments for programmer's notes on the presence of include files.

Here is an example of an include file reference in PHP. Since it is between <? and ?> tags, the reference won't be visible in the HTML source available to the user.

```
<?php
include("$DOCUMENT_ROOT/include/db_connect.inc");
include '/include/db_connect.inc';
include $db_connect_file;
?>
```

Include files often contain references to other include files, application variables and constants, database connection strings, or SQL statements. Basic input validation tests often produce errors that reveal include files, or even internal errors give up these files:

```
Warning :  main( include /config.inc) [ function.main ]:
 failed to create stream: No such file or directory in
 /home/snews/documents/
include /page_headers.inc on line 10

Warning :  Supplied argument is not a valid MySQL-Link
 resource in /usr/local/apache/include/db.inc on line 67
```

Protect Include Files

In Chapter 2, we'll talk about countermeasures in detail, but some simple steps can protect the content of include files from prying eyes. Always use the language's file suffix instead of .inc when naming include files. The file's function and execution will not be affected, but users will be prevented from viewing the source code in the file. For example, a database.inc file will not be parsed by the ASP filter and therefore everything between <% and %> will be visible in the HTML source. By renaming the file to database.asp, then only HTML tags that lie outside of the ASP tags will be visible.

```
<%
 'This line will not be visible if the file suffix is .asp
%>
<!-- This line will be visible regardless of the file suffix -->
```

If you're using Apache::ASP, then you can either rename the files to .asp or modify the httpd.conf file to ensure their content is always interpreted as opposed to being sent in source format:

```
<FilesMatch "\.(asp|inc)$">
    SetHandler  perl-script
    PerlModule  Apache::ASP
    PerlHandler Apache::ASP
    PerlSetVar  IncludesDir .;/home/httpd/asp/shared
    PerlSetVar  StateDir /tmp/state
</FilesMatch>
```

The line in bold will match all files that end in .asp or .inc and parse them with the proper module, as opposed to dumping their raw source to a user's browser.

The <FilesMatch> directive is an effective technique to control access on a per-file basis. It uses the standard regex engine, so you could extend the directive to match many custom extensions. Also, try the <Directory> or <Location> directives to implement restrictions based on directory names.

The <FilesMatch> trick can also be used to prevent users from accessing backup files that have been accidentally left in the web document root. For example, the following syntax prevents users from downloading sensitive files such as database.php.old, menu.pl.bak, scripts.tar.gz, or cgi-bin.tgz:

```
<FilesMatch "\.(old|bak|tar\.gz|tgz)$">
    Order Deny,Allow
    Deny from All
</FilesMatch>
```

Enumerate All Forms

Forms are one of the most vulnerable parts of an application. Here, the application requests data from an untrusted and potentially malicious source: the user. When we discuss input validation attacks in Chapter 2, we will demonstrate how any form data can be manipulated. For example, even if a drop-down menu contains three pre-determined choices (such as male, female, other), the application should not trust that it will receive one of those three responses when the form is submitted. Hence, record every parameter that the form uses because these will be used later on for input validation. The obvious indicator of a form is the HTML <form> tag; however, the salient portions are the "input type" definitions:

```
<INPUT TYPE="hidden" NAME="sess_id" VALUE="">
<INPUT TYPE="hidden" NAME="postit" VALUE="TRUE">
```

```
<INPUT TYPE="hidden" NAME="insertinto" VALUE="1">
<INPUT TYPE="hidden" NAME="BoardID" VALUE="1">
<INPUT CLASS="button" TYPE="submit" NAME="new_topic"
 VALUE="Thema posten">
<INPUT CLASS="button" TYPE="submit" NAME="preview_topic"
 VALUE="Vorschau">
```

The preceding form snippet is from an application called APBoard. The APBoard application handles multiple message boards, or "forums" in APBoard parlance. The value of the hidden tag named *insertinto* (meaning insert into the forum ID number of the value) can be changed to enable a user to post to an arbitrary forum—even one to which access is password-protected. ProXy (http://es-crew.de/) discovered this vulnerability. Also note that hidden tags track the session ID and other variables. A user can easily examine and modify hidden tags.

Form-based authentication is also a primary target for brute-force password-guessing attacks. With just a few lines of Perl (or your language of choice), you can craft a brute-force tool to test weak passwords in form-based authentication. We'll address this in more detail in later chapters. For now, we need to finish profiling the application!

Enumerate All GET Parameters

Many applications track variables through URL parameters. The server sets these parameters based on user permission level, a user's action, a session ID, or similar function. Like forms, GET parameters are a high-risk area for input validation and SQL injection attacks.

Certain applications rely on parameter-driven techniques. For example, the main page may be called main.asp?menu=viewprofile. Here, a single ASP file generates different content based on the value of "menu": *viewprofile*, *user*, *welcome*, *admin*, *debug*, and so on.

Once you've enumerated the GET parameters, return to each page and methodically delete each parameter from the URL. Observe how the application reacts. This can point to the parameter's function or its relation to session tracking, or it can generate informational errors. Each GET parameter should also be tested for input validation and SQL injection attacks.

Protect Parameters

If the application uses GET parameters to track important values, such as session IDs or usernames, then you might consider using POST requests more often. The parameters to a POST request will not show up in a browser's history file or bookmarks. However, be aware that POST requests are consequently less reliable for users to bookmark. This does not protect the parameters from being manipulated; it merely protects

them from casual "shoulder-surfing" or retrieval in a shared computing environment (Internet cafés, for example).

Identify Vectors for Directory Attacks

Directory attacks take two forms: traversal and listing. A directory traversal attack is an attempt to access files outside of the web document root, or files within the document root that are otherwise restricted to the user. The primary vector for a directory traversal attack is in the URL. Therefore, this is where to focus checks for these types of vulnerabilities.

Applications that use templating techniques are prime candidates for directory traversals. Such an application has file references within the URL. All three of these examples are vulnerable to directory traversal attacks that can access an arbitrary file:

- http://website/cgi-bin/bb-hostsvc.sh?**HOSTSVC=www,website,com.cpu**
- http://website/servlet/webacc?**User.html=index**
- http://website/ultraboard.pl?action=PrintableTopic&**Post=42**

The typical attack merely involves replacing the problematic parameter with an arbitrary file:

- ../../../../etc/passwd
- ../../conf/httpd.conf
- ../../../../boot.ini
- ../../../../../winnt/repair/sam

At this point, we must emphasize the importance of the profiling the platform step taken earlier in this chapter. It does you no good to attempt to pull the /etc/passwd file from an IIS system vulnerable to directory traversal. Know the operating system and common locations for sensitive files.

Slightly more advanced techniques require a trailing NULL (%00) character in order to properly terminate the string. In the C programming language, a string is represented as an array of characters terminated by a NULL byte. So, while Perl might happily accept "../../etc/passwd%00html" as a string value, the underlying operating system that handles file access sees it only as "../etc/passwd" and ignores the portion after the %00. Try this to bypass scripts that check for file extensions or automatically append characters to file names. Also, see if %0a or %0d perform similar functions in your file parsing.

Identify Areas that Provide File Upload Capability

Not all applications provide or even require a file upload capability. However, if you do encounter this functionality then be sure to note the pages and parameters involved. File upload introduces several threats to the application:

- **Malicious Content** A user might be able to upload an executable file. This could be a cmdasp.asp file that lets the user run arbitrary commands on the IIS server. It could be a PHP file that simply uses the passthru function to run arbitrary commands on the web server. Alternately, the file may contain a virus or Trojan horse that is intended to attack another user.
- **File Overwrite** A user might be able to overwrite a system file such as httpd.conf, /etc/passwd, or .htaccess in order to create a back door into the server. Or, the user could overwrite a file within the web document root such as login.pl in order to gather usernames and passwords or perform some social engineering trick.
- **Denial of Service** A user might be able to upload excessively large files that either cause the application to crash or fill up the server's disk space.

Identify Errors

There are two parts of this step. First, simply try to generate some errors in the application. You can accomplish this by inserting garbage characters, deleting parameters, inserting punctuation (especially single quotes), and doing anything you're not "supposed" to be able to do within the application.

Second, identify what types of errors are generated on the server and how they are displayed to the user's browser. Did it return the server's default HTTP 500 message? Is it a customized error page? Does an error return a custom page, but an HTTP 200 message? What information does the error contain? Can you identify path information? What about internal variables or references to other files? Is the error related to SQL queries? In any of these cases, make a note of the error and record any information it provides.

Protect Error Messages

Like the attack, this defense has two steps. Errors can be caught in two locations. The first location is the web or application server. Most web servers provide the capability to create custom response pages for

HTTP error response codes. Change the content of these pages so that it does not include any server or application information. The second location for error messages is within the application itself. Make sure that the application has proper error-handling routines that default to a simple, innocuous error message.

Determine Which Pages Require SSL

Part of profiling the platform is to identify whether SSL is enabled and determine what encryption algorithms are enabled. As you go through the application, identify which pages are accessible by SSL. In some cases, such as an online banking web site, the entire application should be over SSL. In other cases, such as web-based e-mail, only the login and profile pages might require SSL.

The next test is to replace all of the https:// references with http:// and see if the application still serves the page. Programmers tend to program for the expected. In other words, the assumption might be that the initial login page redirects from port 80 to port 443 and there the user will happily stay. That is not always the case, so the server and application should be designed to ensure that sensitive files are transmitted via SSL.

Table 1-3 summarizes the application profile process.

Step	Subsequent Steps and Potential Attacks
Harvest the web site	Search for comments, e-mail addresses, SQL statements, <script> tags, SSI, etc.
Enumerate the directory structure and files	Obtain additional files by deduction. For example, look for naming trends, additional ../inc, ../include, or ../scripts directories. Try appending .bak, .old, or .txt to these files in order to view previous versions.
Identify authentication mechanism	Target login prompts for brute-force attacks against trivial passwords. Record how many invalid passwords can be entered before an account is locked. How long is it locked? What is the password reminder mechanism? Can the reminder be attacked or spoofed?
Identify authorization mechanism	Record relevant cookies, other headers, GET and POST parameters, and what functions are available to different users. How many tiers of users exist? This will be the focus of horizontal and vertical privilege escalation attacks.
Identify all "support" files	May contain developer comments, but their content does not usually introduce any security vulnerability. On the other hand, validating that certain file extensions such as .htr, .ida, and .idq are in use definitely identifies potential vulnerabilities on an IIS server.

Table 1-3. Application Profile Checklist

Step	Subsequent Steps and Potential Attacks
Identify all include files .inc .inc.php .js config.inc database.inc db_connect.inc footer.inc global.asa header.inc	Search each file for comments, variables, SQL statements, database connection strings, and passwords. Try appending .bak, .old, or .txt to these files in order to view previous versions.
Enumerate all forms type=hidden type=password	Brute-force authentication pages. Brute-force "random" values. Test input validation. Test SQL injection. Test error handling.
Enumerate all GET parameters ?name1=value1&…	Test input validation. Test SQL injection. Test session replay. Test error handling.
Enumerate the effect of absent GET parameters ?name1=value1&…	Delete combinations of parameters to identify which values are related to session management, authentication, authorization, and application functionality.
Identify vectors for directory traversal attacks	Search URL parameters: ?something.html ?index=english.html ?document=filename ?file=name ?load=filename ?image=filename
Identify areas that provide file upload capability	Test for script execution and directory traversal attacks.
Identify errors	Try basic input validation strings: ' -- %00 (nothing, delete the parameter) Record useful information: HTTP response message (200, 401, 403, 404, 500, 501) Full path information File names (and include files) Variables SQL syntax
Determine which pages require SSL	Can the same URL be accessed with HTTP instead? If a site uses frames, are all of the frames accessed via SSL?

Table 1-3. Application Profile Checklist *(continued)*

SUMMARY

In order to fully vet the security of an application, it must first be fully profiled. This basically involves gathering as much information about the platform (operating system, server, database) and the application. Web application security does not necessarily require a web programmer, but it does require a systematic approach and understanding of the underlying technology. As we will demonstrate in later chapters, it is easy to generate an error by inserting a tick (') into a URL parameter, but a good profile of the application and knowledge of SQL can turn an innocuous error into a severe exploit. Once we've donned the deerstalker cap, we're ready to move on to attacking the application.