

# CHAPTER 1

**INTRODUCTION  
TO WEB  
APPLICATIONS  
AND SECURITY**

Remember the early days of the online revolution? Command-line terminals, 300 baud modems, BBS, FTP. Later came Gopher, Archie, and this new, new thing called Netscape that could render online content in living color, and we began to talk of this thing called the World Wide Web...

How far we have come since the early '90s! Despite those few remaining naysayers who still utter the words "dot com" with dripping disdain, the Internet and, in particular, the World Wide Web have radiated into every aspect of human activity like no other phenomenon in recorded history. Today, over this global communications medium, you can almost instantaneously

- ▼ Purchase a nearly unlimited array of goods and services, including housing, cars, airline tickets, computer equipment, and books, just to name a few
- Perform complex financial transactions, including banking, trading of securities, and much more
- Find well-researched information on practically every subject known to humankind
- Search vast stores of information, readily pinpointing the one item you require from amongst a vast sea of data
- Experience a seemingly limitless array of digital multimedia content, including movies, music, images, and television
- Access a global library of incredibly diverse (and largely free) software tools, from operating systems to word processors
- ▲ Communicate in real time with anyone, anywhere, for little or no cost using Web-based e-mail, telephony, or chat

And this is just the beginning. The Web is evolving as we speak into something even more grand than its current incarnation, becoming easier to use, more accessible, full of even more data, and still more functional with each passing moment. Who knows what tomorrow holds in store for this great medium?

Yet, despite this immense cornucopia enjoyed by millions every day, very few actually understand how it all works, even at the most basic technical level. Fewer still are aware of the inherent vulnerability of the technologies that underlie the applications running on the World Wide Web and the ease with which many of them fall prey to online vandals or even more insidious forces. Indeed, it is a fragile Web we have woven.

We will attempt to show you exactly how fragile throughout this book. Like the other members of the Hacking Exposed series, we will illustrate this fragility graphically with examples from our recent experiences working as security consultants for large organizations where we have identified, exploited, and recommended countermeasures for issues exactly as presented in these pages.

Our goal in this first chapter is to present an overview of Web applications, where common security holes lie, and our methodology for uncovering them before someone else does. This methodology will serve as the guiding structure for the rest of the book—each chapter is dedicated to a portion of the methodology we will outline here, covering each step in detail sufficient for technical readers to implement countermeasures, while remaining straightforward enough to make the material accessible to lay readers who don't have the patience for a lot of jargon.

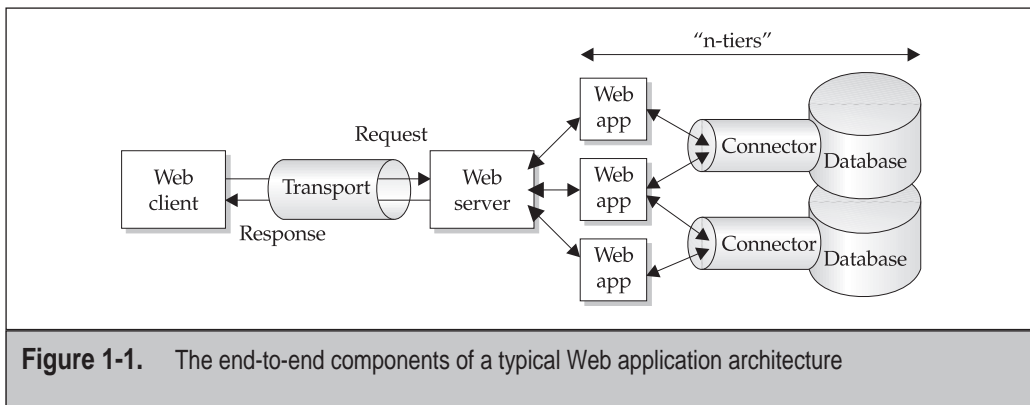
Let's begin our journey with a clarification of what a Web application is, and where it lies in the overall structure of the Internet.

## THE WEB APPLICATION ARCHITECTURE

Web application architectures most closely approximate the centralized model of computing, with many distributed "thin" clients that typically perform little more than data presentation connecting to a central "thick" server that does the bulk of the processing. What sets Web architectures apart from traditional centralized computing models (such as mainframe computing) is that they rely substantially on the technology popularized by the World Wide Web, the Hypertext Markup Language (HTML), and its primary transport medium, Hypertext Transfer Protocol (HTTP).

Although HTML and HTTP define a typical Web application architecture, there is a lot more to a Web app than these two technologies. We have outlined the basic components of a typical Web app in Figure 1-1.

In the upcoming section, we will discuss each of the components of Figure 1-1 in turn (don't worry if you're not immediately familiar with each and every component of Figure 1-1; we'll define them in the coming sections).



**Figure 1-1.** The end-to-end components of a typical Web application architecture

## A Brief Word about HTML

Although HTML is becoming a much less critical component of Web applications as we write this, it just wouldn't seem appropriate to omit mention of it completely since it was so critical to the early evolution of the Web. We'll give a very brief overview of the language here, since there are several voluminous primers available that cover its every aspect (the complete HTML specification can be found at the link listed in the "References and Further Reading" section at the end of this chapter). Our focus will be on the security implications of HTML.

As a *markup language*, HTML is defined by so-called *tags* that define the format or capabilities of document elements. Tags in HTML are delimited by angle brackets < and >, and can define a broad array of formats and functionalities as defined in the HTML specification. Here is a simple example of basic HTML document structure:

```
<HTML>
<H1>This is a First-Level Header</H1>
<p>This is the first paragraph.</p>
</HTML>
```

When displayed in a Web browser, the tags are interpreted and the document elements are given the format or functionality defined by the tags, as shown in the next illustration (we'll discuss Web browsers shortly).



As we can see in this example, the text enclosed by the <H1> </H1> brackets is formatted with a large, boldfaced font, while the <p> </p> text takes on a format appropriate for the body of the document. Thus, HTML primarily serves as the *data presentation engine* of a Web application (both server- and client-side).

As we've noted, a complete discussion of the numerous tags supported in the current HTML spec would be inappropriate here, but we will note that there are a few tags that can be used to deleterious effect by malicious hackers. Most commonly abused tags are related to taking user input (which is done using the <INPUT> tag, wouldn't you know). For

example, one of the most commonly abused input types is called “hidden,” which specifies a value that is not displayed in the browser, but nevertheless gets submitted with any other data input to the same form. Hidden input can be trivially altered in a client-side text editor and then posted back to the server—if a Web application specifies merchandise pricing in hidden fields, you can see where this might lead. Another popular point of attack is HTML forms for taking user input where variables (such as password length) are again set on the client side. For this reason, most savvy Web application designers don’t set critical variables in HTML very much anymore (although we still find them, as we’ll discuss throughout this book). In our upcoming overview of Web browsers in this chapter, we’ll also note a few tags that can be used to exploit client-side security issues.

Most of the power of HTML derives from its confluence with HTTP. When combined with HTTP’s ability to send and receive HTML documents, a vibrant protocol for communications is possible. Indeed, HTML over HTTP is considered the lingua franca of the Web today. Thus, we’ll spend more time talking about HTTP in this book than HTML by far.

Ironically, despite the elegance and early influence of HTML, it is being superseded by other technologies. This is primarily due to one of HTML’s most obvious drawbacks: it is a static format that cannot be altered on the fly to suit the constantly shifting needs of end users. Most Web sites today use scripting technologies to generate content on the fly (these will be discussed in the upcoming section “The Web Application”).

Finally, the ascendance of another markup language on the Internet has marked a decline in the use of HTML, and may eventually supersede it entirely. Although very similar to HTML in its use of tags to define document elements, the eXtensible Markup Language (XML) is becoming the universal format for structuring data on the Web due to its extensibility and flexibility to represent data of all types. XML is well on its way to becoming the new lingua franca of the Web, particularly in the arena of Web services, which we will cover briefly later in this chapter and at length in Chapter 10.

OK, enough about HTML. Let’s move on to the basic component of Web applications that’s probably not likely to change anytime soon, HTTP.

## Transport: HTTP

As we’ve mentioned, Web applications are largely defined by their use of HTTP as the medium of communication between client and server. HTTP version 1.0 is a relatively simple, stateless, ASCII-based protocol defined in RFC 1945 (version 1.1 is covered in RFC 2616). It typically operates over TCP port 80, but can exist on any unused port. Each of its characteristics—its simplicity, statelessness, text base, TCP 80 operation—is worth examining briefly since each is so central to the (in)security of the protocol. The discussion below is a very broad overview; we advise readers to consult the RFCs for more exacting detail.

HTTP’s simplicity derives from its limited set of basic capabilities, request and response. HTTP defines a mechanism to request a resource, and the server returns that resource if it is able. Resources are called *Uniform Resource Identifiers* (URIs) and they can range from static text pages to dynamic streaming video content. Here is a simple example of an HTTP GET request and a server’s HTTP 200 OK response, demonstrated using

the netcat tool. First, the client (in this case, netcat) connects to the server on TCP 80. Then, a simple request for the URI “/test.html” is made, followed by two carriage returns. The server responds with a code indicating the resource was successfully retrieved, and forwards the resource’s data to the client.

```
C:\>nc -vv www.test.com 80
www.test.com [10.124.72.30] 80 (http) open
GET /test.html HTTP/1.0

HTTP/1.1 200 OK
Date: Mon, 04 Feb 2002 01:33:20 GMT
Server: Apache/1.3.22 (Unix)
Connection: close
Content-Type: text/html

<HTML><HEAD><TITLE>TEST.COM</TITLE>etc.
```

HTTP is thus like a hacker’s dream—there is no need to understand cryptic syntax in order to generate requests, and likewise decipher the context of responses. Practically anyone can become a fairly proficient HTTP hacker with very little effort.

Furthermore, HTTP is stateless—no concept of session state is maintained by the protocol itself. That is, if you request a resource and receive a valid response, then request another, the server regards this as a wholly separate and unique request. It does not maintain anything like a session or otherwise attempt to maintain the integrity of a link with the client. This also comes in handy for hackers, as there is no need to plan multi-stage attacks to emulate intricate session maintenance mechanisms—a single request can bring a Web server or application to its knees.

HTTP is also an ASCII text-based protocol. This works in conjunction with its simplicity to make it approachable to anyone who can read. There is no need to understand complex binary encoding schemes or use translators—everything a hacker needs to know is available within each request and response, in cleartext.

Finally, HTTP operates over a well-known TCP port. Although it can be implemented on any other port, nearly all Web browsers automatically attempt to connect to TCP 80 first, so practically every Web server listens on that port as well (see our discussion of SSL/TLS in the next section for one big exception to this). This has great ramifications for the vast majority of networks that sit behind those magical devices called firewalls that are supposed to protect us from all of the evils of the outside world. *Firewalls and other network security devices are rendered practically defenseless against Web hacking when configured to allow TCP 80 through to one or more servers.* And what do you guess is the most common firewall configuration on the Internet today? Allowing TCP 80, of course—if you want a functional Web site, you’ve gotta make it accessible.

Of course, we’re oversimplifying things a great deal here. There are several exceptions and qualifications that one could make about the previous discussion of HTTP.

## SSL/TLS

One of the most obvious exceptions is that many Web applications today tunnel HTTP over another protocol called Secure Sockets Layer (SSL). SSL can provide for transport-layer encryption, so that an intermediary between client and server can't simply read cleartext HTTP right off the wire. Other than "wrapping" HTTP in a protective shell, however, SSL does not extend or substantially alter the basic HTTP request-response mechanism. *SSL does nothing for the overall security of a Web application other than to make it more difficult to eavesdrop on the traffic between client and server.* If an optional feature of the SSL protocol called *client-side certificates* is implemented, then the additional benefit of mutual authentication can be realized (the client's certificate must be signed by an authority trusted by the server). However, few if any sites on the Internet do this today.

The latest version of SSL is called Transport Layer Security (TLS). SSL/TLS typically operates via TCP port 443. That's all we're going to say about SSL/TLS for now, but it will definitely come up in further discussions throughout this book.

## State Management: Cookies

We've dwelt a bit on the fact that HTTP itself is stateless, but a number of mechanisms have been conceived to make it behave like a stateful protocol. The most widely used mechanism today uses data called *cookies* that can be exchanged as part of the HTTP request/response dialogue to make the client and application think they are actually connected via virtual circuit (this mechanism is described more fully in RFC 2965). Cookies are best thought of as tokens that servers can hand to a client allowing the client to access the Web site as long as they present the token for each request. They can be stored temporarily in memory or permanently written to disk. Cookies are not perfect (especially if implemented poorly) and there are issues relating to security and privacy associated with using them, but no other mechanism has become more widely accepted yet. That's all we're going to say about cookies for now, but it will definitely come up in further discussions throughout this book, especially in Chapter 7.

## Authentication

Close on the heels of statefulness comes the concept of authentication. What's the use of keeping track of state if you don't even know who's using your application? HTTP can embed several different types of authentication protocols. They include

- ▼ **Basic** Cleartext username/password, Base-64 encoded (trivially decoded).
- **Digest** Like Basic, but passwords are scrambled so that the cleartext version cannot be derived.
- **Form-based** A custom form is used to input username/password (or other credentials) and is processed using custom logic on the back end. Typically uses a cookie to maintain "logged on" state.
- **NTLM** Microsoft's proprietary authentication protocol, implemented within HTTP request/response headers.

- **Negotiate** A new protocol from Microsoft that allows any type of authentication specified above to be dynamically agreed upon by client and server, and additionally adds Kerberos for clients using Microsoft's Internet Explorer browser version 5 or greater.
- **Client-side Certificates** Although rarely used, SSL/TLS provides for an option that checks the authenticity of a digital certificate presented by the Web client, essentially making it an authentication token.
- ▲ **Microsoft Passport** A single-sign-in (SSI) service run by Microsoft Corporation that allows Web sites (called "Passport Partners") to authenticate users based on their membership in the Passport service. The mechanism uses a key shared between Microsoft and the Partner site to create a cookie that uniquely identifies the user.

These authentication protocols operate right over HTTP (or SSL/TLS), with credentials embedded right in the request/response traffic. We will discuss them and their security failings in more detail in Chapter 5.

**NOTE**

Clients authenticated to Microsoft's IIS Web server using Basic authentication are impersonated as if they were logged on interactively.

## Other Protocols

HTTP is deceptively simple—it's amazing how much mileage creative people have gotten out of its basic request/response mechanisms. However, it's not always the best solution to problems of application development, and thus still more creative people have wrapped the basic protocol in a diverse array of new dynamic functionality.

One simple example is what to do with non-ASCII-based content requested by a client. How does a server fulfill that request, since it only knows how to speak ASCII over HTTP? The venerable Multipart Internet Mail Extensions (MIME) format is used to transfer binary files over HTTP. MIME is outlined in RFC 2046. This enables a client to request almost any kind of resource with near assurance that the server will understand what it wants and return the object to the client.

Of course, Web applications can also call out to any of the other popular Internet protocols as well, such as e-mail (SMTP) and file transfer (FTP). Many Web applications rely on embedded e-mail links to communicate with clients.

Finally, work is always afoot to add new protocols to the HTTP suite. One of the most significant new additions is Web Distributed Authoring and Versioning (WebDAV). WebDAV is defined in RFC 2518, which describes several mechanisms for authoring and managing content on remote Web servers. Personally, we don't think this is a good idea, as protocol that involves writing data to a Web server is trouble in the making, a theme we'll see time and again in this book.

Nevertheless, WebDAV is backed by Microsoft and already exists in their widely deployed products, so a discussion of its security merits is probably moot at this point.

## The Web Client

The standard Web application client is the Web browser. It communicates via HTTP (among other protocols) and renders Hypertext Markup Language (HTML), among other markup languages. In combination, HTML and HTTP present the data processed by the Web server.

Like HTTP, the Web browser is also deceptively simple. Because of the extensibility of HTML and its variants, it is possible to embed a great deal of functionality within seemingly static Web content.

Some of those capabilities are based around active content technologies like Microsoft's ActiveX and Sun Microsystem's Java. Embedding an ActiveX object in HTML is this simple:

```
<object id="scr"
  classid="clsid:06290BD5-48AA-11D2-8432-06008C3FBFC">
</object>
```

Once again, in the world of the Web, everything is in ASCII. When rendered in a Web browser that understands what to do with ActiveX, the control specified by this object tag will either be downloaded from the remote Web site, or loaded directly from the local machine if it is already installed (many ActiveX controls come preinstalled with Windows and related products). Then it is checked for authenticity using Microsoft's Authenticode technology, and by default a message is displayed explaining who digitally signed the control and offering the user a chance to decline to run it. If the user says yes, the code executes. Some exceptions to this behavior are controls marked "safe for scripting," which run without any user intervention. We'll talk more about those in Chapter 12.

HTML is a capable language, but it's got its limitations. Over the years, new technologies like Dynamic HTML and Style Sheets have emerged to spice up the look and management of data presentation. And, as we've noted, more fundamental changes are afoot currently, as the eXtensible Markup Language (XML) slowly begins to replace HTML as the Web's language of choice.

Finally, the Web browser can speak in other protocols if it needs to. For example, it can talk to a Web server via SSL if that server uses a certificate that is signed by one of the many root authorities that ship certificates with popular commercial browsers. And it can request other resources such as FTP services. Truly, the Web browser is one of the greatest weapons available to attackers today.

Despite all of the frosting available with current Web browsers, it's still the raw HTTP/HTML functionality that is the hacker's best friend. In fact, throughout most of this book, we'll eschew using Web browsers, preferring instead to perform our tests with tools that make raw HTTP connections. A great deal of information slips by underneath the pretty presentation of a Web browser, and in some cases, they surreptitiously reformat some requests that might be used to test Web server security (for example, Microsoft's Internet Explorer strips out dot-dot-slashes before sending a request). Now, we can't have that happening during a serious security review, can we?

## The Web Server

The Web server is most simply described as an HTTP daemon (service) that receives client requests for resources, performs some basic parsing on the request to ensure the resource exists (among other things), and then hands it off to the Web application logic (see Figure 1-1) for processing. When the logic returns a response, the HTTP daemon returns it to the client.

There are many popular Web server software packages available today. In our consulting work, we see a large amount of Microsoft IIS, the Apache Software Foundation's Apache HTTP Server (commonly just called "Apache"), AOL/Netscape's Enterprise Server, and Sun's iPlanet. To get an idea of what the Web is running on its servers at any one time, check out the Netcraft survey at <http://www.netcraft.net>.

Although an HTTP server seems like such a simple thing, we once again must point out that numerous vulnerabilities in Web servers have been uncovered over the years. So many, in fact, that you could argue persuasively that Web server vulnerabilities drove hacking and security to international prominence during the 1990s.

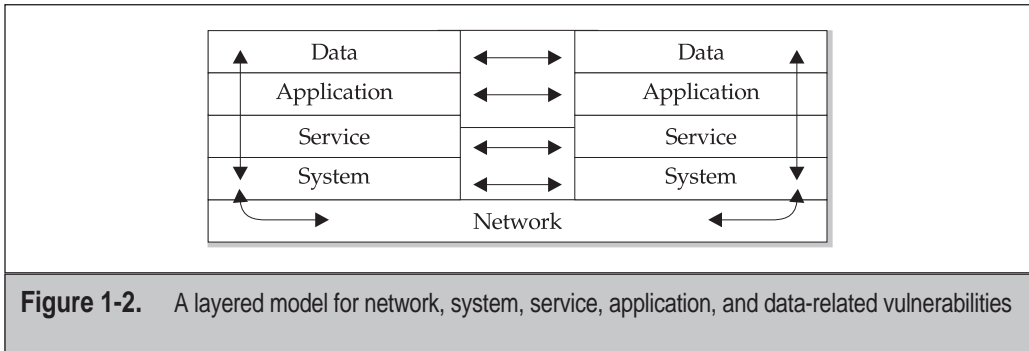
## Web Servers vs. Web Applications

Which brings up the oft-blurred distinction between Web servers and Web applications. In fact, many people don't distinguish between the Web server and the applications that run on it. This is a major oversight—we believe that vulnerabilities in either the server or elsewhere in the application are important, yet distinct, and will continue to make this distinction throughout this book.

While we're at it, let's also make sure everyone understands the distinction between two other classes of vulnerabilities, network- and system-level vulnerabilities. Network- and system-level vulnerabilities operate below the Web server and Web application. They are problems with the operating system of the Web server, or insecure services running on a system sitting on the same network as the Web server. In either case, exploitation of vulnerabilities at the network or system level can also lead to compromise of a Web server and the application running on it. This is why firewalls were invented—to block access to everything but the Web service so that you don't have to worry so much about intruders attacking these other points.

We bring these distinctions up so that readers learn to approach security holistically. Anywhere a vulnerability exists—be it in the network, system, Web server, or application—there is the potential for compromise. Although this book deals primarily with Web applications, and a little with Web servers, make sure you don't forget to close the other holes as well. The other books in the Hacking Exposed series cover network and system vulnerabilities in great detail.

Figure 1-2 diagrams the relationship among network, system, Web server, and Web application vulnerabilities to further clarify this point. Figure 1-2 is patterned roughly after the OSI networking model, and illustrates how each layer must be traversed in order to reach adjacent layers. For example, a typical attack must traverse the network, dealing with wire-level protocols such as Ethernet and TCP/IP, then pass the system layer with



**Figure 1-2.** A layered model for network, system, service, application, and data-related vulnerabilities

housekeeping issues such as packet reassembly, and on through what we call the services layer where servers like the HTTP daemon live, through to application logic, and finally to the actual data manipulated by the application. At any point during the path, a vulnerability existing in one of the layers could be exploited to cause system or network compromise.

However, like the OSI model, the abstraction provided by lower layers gives the appearance of communicating logically over one contiguous medium. For example, a properly implemented attack against an HTTP server would simply ride unobtrusively through the network and system layers, then arrive at the services layer to do its damage. The application and data layers are none the wiser, although a successful exploit of the HTTP server may lead to total system compromise, in which case the data is owned by the attacker anyway.

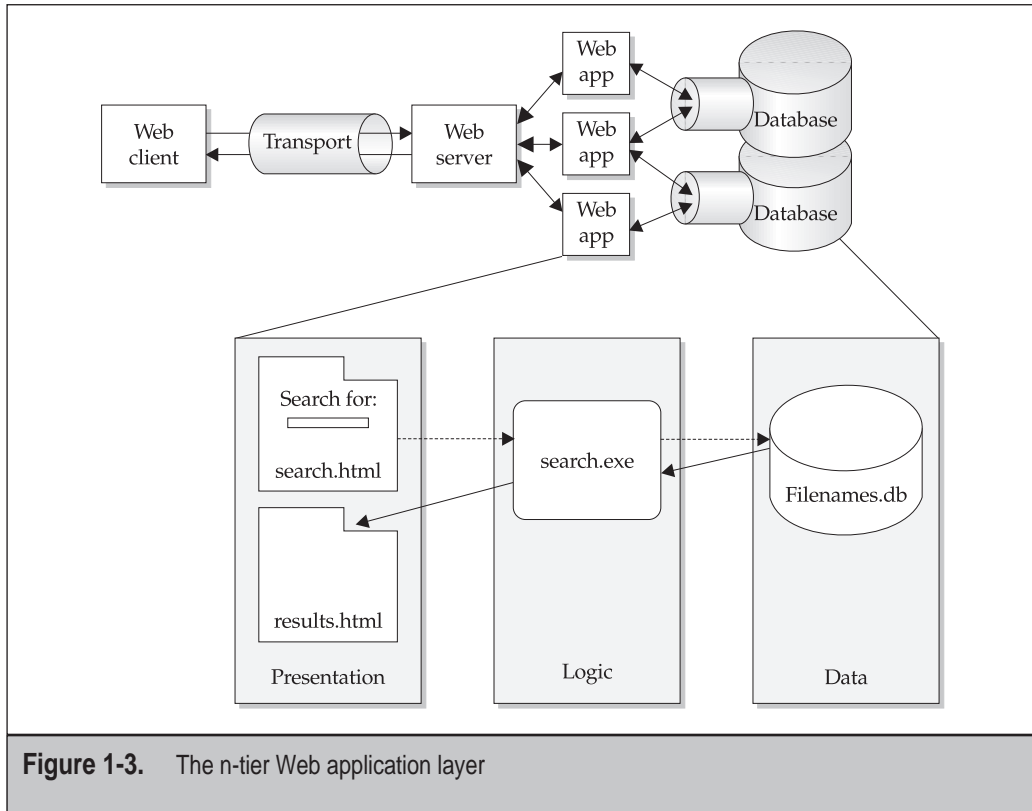
Once again, our focus throughout this book will primarily be on the application layer, with occasional coverage of services like HTTP. We hope this clarifies things a bit going forward.

## The Web Application

The core of a modern Web site is its server-side logic (although client-side logic embedded in the Web browser still does some heavy lifting). This so-called “n-tier” architecture extends what would normally be a pretty unsophisticated thing like a HTTP server and turns it into a dynamic engine of functionality that almost passes for a seamless, stateful application that users can interact with in real time.

The concept of “n-tier” is important to an understanding of a Web application. In contrast to the single layer presented in Figure 1-1, the Web app layer can itself be comprised of many distinct layers. The stereotypical representation is three-layered architecture, comprised of presentation, logic, and data, as shown in Figure 1-3. Let’s discuss each briefly.

The presentation layer provides a facility for taking input and displaying results. The logic layer takes the input from the presentation layer and performs some work on it (perhaps requiring the assistance of the data layer), and then hands the result back to



**Figure 1-3.** The n-tier Web application layer

presentation. Finally, the data layer provides nonvolatile storage of information that can be queried or updated by the logic layer, providing an abstraction so that data doesn't need to be hard-coded into the logic layer, and can be easily updated (we'll discuss the data layer by itself in an upcoming section).

To understand how this all works together, let's illustrate with an example. Consider a simple Web application that searches the local Web server hard drive for filenames containing text supplied by the user and displays the results. The presentation layer would consist of a form with a field to allow input of the search string. The logic layer might be an executable program that takes the input string, ensures that it doesn't contain any potentially malicious characters, and invokes the appropriate database connector to open a connection to the data layer, finally performing a query using the input string. The data layer might consist of a database that stores an index of all the filenames resident on the local machine, updated in real time. The database query returns a set of matching records, and spits them back to the logic layer executable. The logic layer parses out unnecessary data in the recordset, and then returns the matching records to the presentation layer, which embeds them in HTML so that they are formatted prettily for the end user on their trip back through the Web server to the client's browser.

Many of the technologies used to actually build applications integrate the functionality of one or more of these layers, so it's often hard to distinguish one from the other in a real-world app, but they're there. For example, Microsoft's Active Server Pages (ASP) allow you to embed server-side logic within Web pages in the presentation layer, so that there is no need to have a distinct executable to perform the database queries (although many sites use a distinct COM object to do the database access, and this architecture may be more secure in some cases; see Chapter 9).

There is a vast diversity of techniques and technologies used to create Web n-tier logic. Some of the most widely used (in our estimation) are categorized by vendor in Table 1-1.

Table 1-1 is a mere snippet of the vast number of objects and technologies that make up a typical Web application. Things like include files, ASA files, and so on all play a supporting role in keeping application logic humming (and also play a role in security vulnerabilities as well, of course).

The key thing to understand about all of these technologies is that they work more like *executables* rather than static, text-based HTML pages. For example, a request for a PHP script might look like this:

```
http://www.somesite.net/article.php?id=425&format=html
```

As you can see, the file `article.php` is run just like an executable, with the items to the left of the question mark treated like additional input, or arguments. If you envision `article.php`

Vendor	Technologies
Microsoft	Active Server Pages (ASP) ASP.NET ISAPI Common Object Model (COM) JavaScript
Sun Microsystems IBM Websphere BEA Weblogic	Java 2 Enterprise Edition (J2EE), including Java Servlets Java Server Pages (JSP) CORBA
Apache Software Foundation	PHP (Hypertext Preprocessor) Jakarta (server-side Java)
(none)	HTML CGI (including Perl)

**Table 1-1.** Selected Web Application Technologies and Vendors

as a Windows executable (call it `article.exe`) run from a command line, the previous example might look like this:

```
C:\>article.exe /id: 425 /format: html
```

Hackers the world over are probably still giving thanks for this crucial development in the Web's evolution, as it provides remote users the ability to *run code on the Web server with user-defined input*. This places an extremely large burden on Web application developers to design their scripts and executables correctly. Most fail to meet this rigorous standard, as we will see throughout this book.

There are also a whole host of vendors who package so-called Web application platforms that combine a Web server with an integrated development environment (IDE) for Web application logic. Some of the more popular players in this space include BEA Systems, Broadvision, and others.

Finally, as is evident from Figure 1-1, multiple applications can run on one Web server. This contributes to the complexity of the overall Web architecture, which in turn increases the risk of security exposures.

## The Database

Sometimes referred to as the “back end,” the data layer typically makes up the last tier in an n-tier architecture. Perhaps more than anything else, the database has been responsible for the evolution of the Web from a static, HTML-driven entity into a dynamic, fluid medium for information retrieval and e-commerce.

The vendors and platforms within the data layer are fairly uniform across the Web today: SQL (of the Microsoft and non-Microsoft variety) and Oracle are the dominant players here. Logic components typically invoke a particular database connector interface to talk directly with databases, make queries, update records, and so on. The most common connector used today is Open Database Connectivity, or ODBC.

## Complications and Intermediaries

Wouldn't the world be a wonderful place if things were as simple as portrayed in Figure 1-1? Of course, the world just isn't as neat and tidy. In order to make Web application architectures scale more readily to the demands of the Internet, a number of contrivances have been conceived.

### Proxies

One of the first usurpers of the clean one-client-to-one-server model was the Web proxy. Folks who administered large networks like America Online (AOL) decided one day that instead of allowing each of their umpteen million individual subscribers to connect to that newfangled Internet thing, they would implement a single gateway through which

all connections had to pass. This gateway would terminate the initial browser request, and then request the original resource on behalf of the client. This allowed the gateway to do things like cache commonly requested Internet content, thus saving bandwidth, increasing performance, and so on. A gateway that makes requests on behalf of a client system has traditionally been called a *proxy*. Proxies largely behave as advertised, sparing bandwidth and decreasing server load, but they have at least one ugly side effect: state management or security mechanisms based on client source IP address tend to get all fouled up when traversing a proxy, since the source address of the client is always the proxy. How do you tell one client's request from another? Even worse, when implemented in arrays as AOL does, one client request may come out of one proxy, and a second request may come out of another. Take home point: don't rely on client-side information when designing Web application state management or security measures.

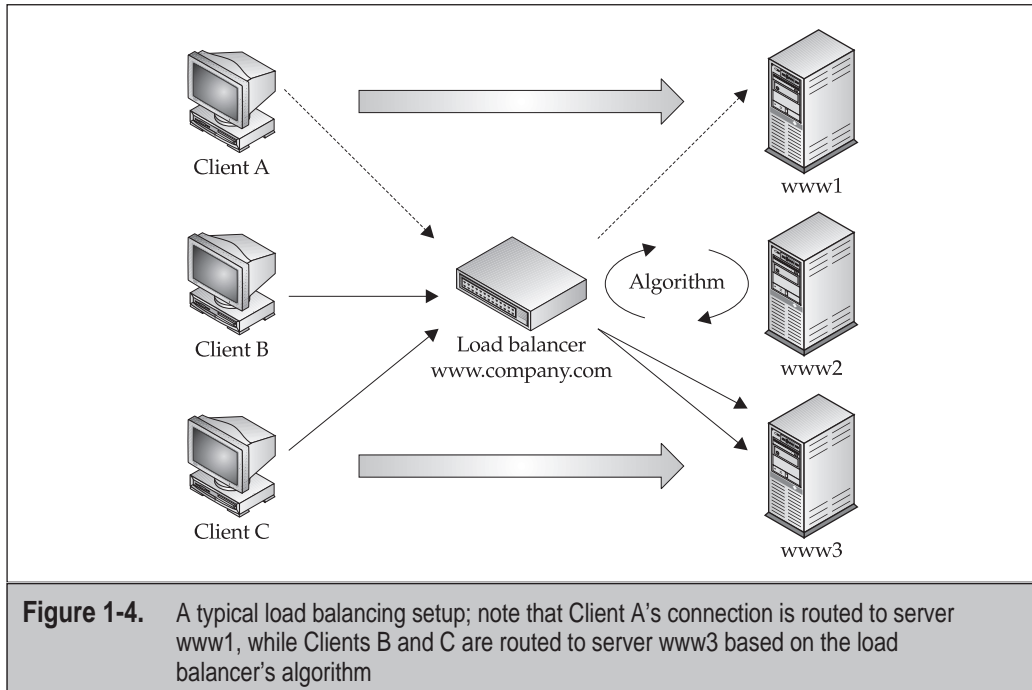
## Load Balancers

As you might imagine, someone soon came up with a similar idea for the server side of the Web equation. Load balancers perform somewhat like reverse proxies, managing the incoming load of client requests and distributing them across a farm of identically configured Web servers. The client neither knows nor cares if one server fulfills its request or another. This greatly improves the scalability of Web architectures, since a theoretically unlimited number of Web servers can be employed to respond to ever-increasing numbers of client requests.

Load balancing algorithms can be categorized into static (where requests are routed in a predetermined fashion such as round-robin) or dynamic (in which requests are shunted to servers based on some variable load factor like least connections or fastest link). The load balancer itself typically takes on a canonical name like `www.company.com`, and then routes requests to virtual servers, which may or may not have Internet-accessible addresses. Figure 1-4 illustrates a typical load balancing setup.

Load balancing implementations we commonly see in our work include Cisco Local Director and F5's Big-IP. Another interesting implementation is the Network Load Balancing (NLB) scheme from Microsoft. It is based on a physical layer broadcasting concept rather than request routing. In some ways, it's sort of like Ethernet's collision detection avoidance architecture. It works like this: An incoming request is broadcast to the entire farm of Web servers. Based on an internal algorithm, only one of the servers will respond. The rest of the client's requests are then routed to that server, like other load balancing schemes. Microsoft's Application Center product uses this approach, and we think it's elegant even though we haven't seen it deployed much. Scalability is greatly enhanced because the balancing device doesn't have to route packets; it only broadcasts them.

Whatever the technology employed, load balancers tend to make life harder for hackers. Because a given request doesn't always get sent to the same server, scanning techniques can yield unpredictable results. We'll discuss this in more detail in Chapter 2.



## The New Model: Web Services

As we've noted more than once in this chapter, the Web is constantly evolving. What's in store for Web application architectures in the near future? As we write this, the words on everybody's lips are *Web services*.

Looking at Figure 1-1 again, Web services are comparable to self-contained, modular Web applications. Web services are based on a set of much-hyped Internet standards-in-development. Those standards include the Web Services Definition Language (WSDL), an XML format for describing network services; the Universal Description, Discovery, and Integration (UDDI) specification, a set of XML protocols and an infrastructure for the description and discovery of Web services; and the Simple Object Access Protocol (SOAP), an XML-based protocol for messaging and RPC-style communication between Web services. (Is anyone not convinced XML will play an important role in the future of the Web?) Leveraging these three technologies, Web services can be mixed and matched to create innovative applications, processes, and value chains.

A quick review of this chapter will tell you why Web services are being held out as the Holy Grail for Web developers. As shown in Table 1-1, there are several competing standards for information interchange between Web applications today. Thus, integrating

two or more Web applications is generally an arduous task of coordinating standards to pass data, protocols, platforms, and so on.

Web services alleviate a lot of this work because they can describe their own functionality and search out and dynamically interact with other Web services via WSDL, UDDI, and SOAP. Web services thus provide a means for different organizations to connect their applications with one another to conduct dynamic e-business across a network, no matter what their application, design, or run-time environment (ASP, ISAPI, COM, J2EE, CORBA, and so on).

WDSL, UDDI, and SOAP grew out of collaborative efforts between Microsoft and various other vendors (including IBM, Ariba, DevelopMentor, and UserLand Software). Many of the other large technology movers like Sun and Oracle are also on board the Web service bandwagon, so even though the current standards may not look the same in six months, it's clear that Web services are here for the long haul. And of course, there will be a whole new crop of security woes as these new technologies move from crawling to walking. We'll look at what's in store security-wise in Chapter 10.

## POTENTIAL WEAK SPOTS

Now that we've described a typical Web application architecture, let's delve briefly into the topics that we will cover in more detail in the coming chapters. Namely, what are the commonly exploited weaknesses in the model we have just described?

Once again referring back to Figure 1-1, what components of our stereotypical Web application architecture would you guess are the most vulnerable to attack? If you guessed "all of them," then you are familiar with the concept of the trick question, and you are also correct. Here is a quick overview of the types of attacks that are typically made against each component of the architecture presented in Figure 1-1.

- ▼ **Web Client** Active content execution, client software vulnerability exploitation, cross-site scripting errors. Web client hacking is discussed in Chapter 12.
- **Transport** Eavesdropping on client-server communications, SSL redirection.
- **Web Server** Web server software vulnerabilities. See Chapter 3.
- **Web Application** Attacks against authentication, authorization, site structure, input validation, and application logic. Covered in the rest of this book.
- ▲ **Database** Running privileged commands via database queries, query manipulation to return excessive datasets. Tackled in Chapter 9.

Now that we've defined the target, let's discuss the approach we'll take for identifying and exploiting these vulnerabilities.

## THE METHODOLOGY OF WEB HACKING

The central goal of this book is to set forth a Web application security review methodology that is comprehensive, approachable, and repeatable by readers who wish to apply the wisdom we've gained over years of performing them professionally. The basic steps in the methodology are

- ▼ Profile the infrastructure
- Attack Web servers
- Survey the application
- Attack the authentication mechanism
- Attack the authorization schemes
- Perform a functional analysis
- Exploit the data connectivity
- Attack the management interfaces
- Attack the client
- ▲ Launch a denial-of-service attack

This book is structured around each of these steps—we've dedicated a chapter to each step so that by the end of your read, you should have a clear idea of how to find and fix the most severe security vulnerabilities in your own site. The following sections will offer a brief preview of what is to come.

### Profile the Infrastructure

The first step in the methodology is to glean a high-level understanding of the target Web infrastructure. Each component of Figure 1-1 should be reviewed: Is there a special client necessary to connect to the application? What transports does it use? Over which ports? How many servers are there? Is there a load balancer? What is the make and model of the Web server(s)? Are external sites relied on for some functionality? Chapter 2 will discuss the tools and techniques for answering these questions and much more.

### Attack Web Servers

The sheer number of Web server software vulnerabilities that have been published makes this one of the first and usually most fruitful areas of research for a Web hacker. If site administration is sloppy, you may hit the jackpot here—Chapter 3 will describe several attacks that yield remote superuser control over a Web server, all over TCP port 80.

### Survey the Application

If no serious vulnerabilities have been found yet, good for the application designers (or maybe they're just lucky). Now attention turns to a more granular examination of the

components of the application itself—what sort of content runs on the server? Surveying a Web application attempts to discern what application technologies are deployed (ASP, ISAPI, Java, CGI, others?), the directory structure and file composition of the site, any authenticated content and the types of authentication used, external linkage (if any), and the nature of back-end datastores (if any). This is probably one of the most important steps in the methodology, as oversights here can have significant effects on the overall accuracy and reliability of the entire application review. Surveying the application is covered in Chapter 4.

## Attack the Authentication Mechanism

If any authenticated content is discovered in the previous step, it should be thoroughly analyzed, as it most likely protects sensitive areas of a site. Techniques for assessing the strength of authentication features include automated password guessing attacks, spoofing tokens within a cookie, and so on. Chapter 5 looks at Web authentication hacking in greater detail.

## Attack the Authorization Schemes

Once a user is authenticated, the next step is to attack access to files and other objects. This can be accomplished in various ways—through directory traversal techniques, changing the user principle (for example, by altering form or cookie values), requesting hidden objects with guessable names, attempting canonicalization attacks, escalating privileges, and tunneling privileged commands to the SQL server. This portion of the methodology is discussed in Chapter 6.

We also discuss one of the most important aspects of authorization—maintaining state—in Chapter 7.

## Perform a Functional Analysis

Another critical step in the methodology is the actual analysis of each individual function of the application. The essence of functional analysis is identifying each component function of the application (for example, order input, confirmation, and order tracking) and attempting to inject faults into each input receptacle. This process of attempted fault injection is central to software security testing, and is sometimes referred to as *input validation attacks*, which is the title of Chapter 8.

## Exploit the Data Connectivity

Some of the most devastating attacks on Web applications actually relate to the back-end database. After all, that's usually where all of the juicy customer data is stored anyway, right? Because of the myriad of ways available to connect Web applications with databases, Web developers tend to focus on the most efficient way to make this connection, rather than the most secure. We'll cover some of the classic methods for extracting data—and even using SQL to take control of the operating system—in Chapter 9.

## Attack the Management Interfaces

Until now, we haven't discussed one of the other essential services that typically runs on or around Web applications: remote management. Web sites run 24/7, which means that it's not always feasible for the Webmaster to be sitting in the data center when something needs updating or fixing. Combined with the natural propensity of Web folk for remote telework (no dress code required), it's a good bet that any given Web application architecture has a port open somewhere to permit remote maintenance of servers, content, back-end databases, and so on.

In addition, just about every networking product (hardware or software) that has been produced since the mid-'90s likely shipped with a Web-based management interface running on an embedded Web server. We'll chat about some of these as well as plain ole' Web server management interfaces in Chapter 11.

## Attack the Client

In many years of professional Web application testing, we've seen darn few reviews take appropriate time to consider attacks against the client side of the Web application architecture. This is a gross oversight in our estimation, since there have been some devastating attacks against the Web user community over the years, including cross-site scripting ploys, like those published for eBay, E\*Trade, and Citigroup's Web sites, as well as Internet-born worms like Nimda that could easily be implemented within a rogue Web site and mailed out via URL to millions of people, or posted to a popular newsgroup, or forwarded via online chat. If you think this is bad, we've only scratched the surface of what we'll cover in Chapter 12.

## Launch a Denial-of-Service Attack

Assuming that an attacker hasn't gotten in at this point in the methodology, the last refuge of a defeated mind is denial of service (DoS), a sad but true component of today's Internet. As its name suggests, DoS describes the act of denying Web application functionality to legitimate users. It is typically carried out by issuing a flood of traffic to a site, drowning out legitimate requests. We'll cover DoS against Web servers in Chapter 3, and against Web applications in Chapter 8.

## SUMMARY

In this chapter, we've taken the 50,000-foot aerial view of a Web application architecture, its components, potential security weaknesses, and a methodology for finding and fixing those weaknesses. The rest of this book will zero in on the details of this methodology. Buckle your seatbelt, Dorothy, because Kansas is going bye-bye.

## REFERENCES AND FURTHER READING

Reference	Link
<i>General References</i>	
Microsoft IIS	<a href="http://www.microsoft.com/iis">http://www.microsoft.com/iis</a>
Microsoft ASP	<a href="http://msdn.microsoft.com/library/psdk/iisref/aspguide.htm">http://msdn.microsoft.com/library/psdk/iisref/aspguide.htm</a>
Microsoft ASP.NET	<a href="http://www.asp.net/">http://www.asp.net/</a>
Hypertext Preprocessor (PHP)	<a href="http://www.php.net/">http://www.php.net/</a>
Apache	<a href="http://www.apache.org/">http://www.apache.org/</a>
Netscape Enterprise Products	<a href="http://enterprise.netscape.com/index.html">http://enterprise.netscape.com/index.html</a>
Java	<a href="http://java.sun.com/">http://java.sun.com/</a>
Java Server Pages (JSP)	<a href="http://java.sun.com/products/jsp/">http://java.sun.com/products/jsp/</a>
IBM Websphere App. Server	<a href="http://www.ibm.com/software/webservers/appserv/">http://www.ibm.com/software/webservers/appserv/</a>
BEA Systems Weblogic App. Server	<a href="http://www.beasys.com/">http://www.beasys.com/</a>
Broadvision	<a href="http://www.broadvision.com/">http://www.broadvision.com/</a>
Cisco Local Director	<a href="http://www.cisco.com/warp/public/cc/pd/cxsr/400/index.shtml">http://www.cisco.com/warp/public/cc/pd/cxsr/400/index.shtml</a>
F5's Big-IP	<a href="http://www.f5.com/">http://www.f5.com/</a>
<i>Specifications</i>	
RFC Index Search Engine	<a href="http://www.rfc-editor.org/rfcsearch.html">http://www.rfc-editor.org/rfcsearch.html</a>
W3C HyperText Markup Language Home Page	<a href="http://www.w3.org/MarkUp/">http://www.w3.org/MarkUp/</a>
eXtensible Markup Language (XML)	<a href="http://www.w3.org/XML/">http://www.w3.org/XML/</a>
WSDL	<a href="http://www.w3.org/TR/wsdl">http://www.w3.org/TR/wsdl</a>
UDDI	<a href="http://www.uddi.org/">http://www.uddi.org/</a>
SOAP	<a href="http://www.w3.org/TR/SOAP/">http://www.w3.org/TR/SOAP/</a>

