CHAPTER 3

Web Application Technologies

Web applications employ a myriad of technologies to implement their functionality. This chapter is a short primer on the key technologies that you are likely to encounter when attacking web applications. We will examine the HTTP protocol, the technologies commonly employed on the server and client sides, and the encoding schemes used to represent data in different situations. These technologies are in general easy to understand, and a grasp of their relevant features is key to performing effective attacks against web applications.

If you are already familiar with the key technologies used in web applications, you can skim through this chapter to confirm that it offers you nothing new. If you are still learning how web applications work, you should read this chapter before continuing to the later chapters on specific vulnerabilities. For further reading on many of the areas covered, we recommend *HTTP*: *The Definitive Guide* by David Gourley and Brian Totty (O'Reilly, 2002), and also the website of the World Wide Web Consortium at `www.w3.org`.

The HTTP Protocol

Hypertext transfer protocol (HTTP) is the core communications protocol used to access the World Wide Web and is used by all of today's web applications. It is a simple protocol that was originally developed for retrieving static text-based resources. It has since been extended and leveraged in various ways to enable it to support the complex distributed applications that are now commonplace.

HTTP uses a message-based model in which a client sends a request message and the server returns a response message. The protocol is essentially connectionless: although HTTP uses the stateful TCP protocol as its transport mechanism, each exchange of request and response is an autonomous transaction and may use a different TCP connection.

HTTP Requests

All HTTP messages (requests and responses) consist of one or more headers, each on a separate line, followed by a mandatory blank line, followed by an optional message body. A typical HTTP request is as follows:

```
GET /auth/488/YourDetails.ashx?uid=129 HTTP/1.1
Accept: application/x-ms-application, image/jpeg, application/xaml+xml,
image/gif, image/pjpeg, application/x-ms-xbap, application/x-shockwave-
flash, */*
Referer: https://mdsec.net/auth/488/Home.ashx
Accept-Language: en-GB
User-Agent: Mozilla/4.0 (compatible; MSIE 8.0; Windows NT 6.1; WOW64;
Trident/4.0; SLCC2; .NET CLR 2.0.50727; .NET CLR 3.5.30729; .NET CLR
3.0.30729; .NET4.0C; InfoPath.3; .NET4.0E; FDM; .NET CLR 1.1.4322)
Accept-Encoding: gzip, deflate
Host: mdsec.net
Connection: Keep-Alive
Cookie: SessionId=5B70C71F3FD4968935CDB6682E545476
```

The first line of every HTTP request consists of three items, separated by spaces:

- A verb indicating the HTTP method. The most commonly used method is `GET`, whose function is to retrieve a resource from the web server. `GET` requests do not have a message body, so no further data follows the blank line after the message headers.
- The requested URL. The URL typically functions as a name for the resource being requested, together with an optional query string containing parameters that the client is passing to that resource. The query string is indicated by the `?` character in the URL. The example contains a single parameter with the name `uid` and the value `129`.
- The HTTP version being used. The only HTTP versions in common use on the Internet are 1.0 and 1.1, and most browsers use version 1.1 by default. There are a few differences between the specifications of these two versions; however, the only difference you are likely to encounter when attacking web applications is that in version 1.1 the `Host` request header is mandatory.

Here are some other points of interest in the sample request:

- The `Referer` header is used to indicate the URL from which the request originated (for example, because the user clicked a link on that page). Note that this header was misspelled in the original HTTP specification, and the misspelled version has been retained ever since.
- The `User-Agent` header is used to provide information about the browser or other client software that generated the request. Note that most browsers include the Mozilla prefix for historical reasons. This was the `User-Agent` string used by the originally dominant Netscape browser, and other browsers wanted to assert to websites that they were compatible with this standard. As with many quirks from computing history, it has become so established that it is still retained, even on the current version of Internet Explorer, which made the request shown in the example.
- The `Host` header specifies the hostname that appeared in the full URL being accessed. This is necessary when multiple websites are hosted on the same server, because the URL sent in the first line of the request usually does not contain a hostname. (See Chapter 17 for more information about virtually hosted websites.)
- The `Cookie` header is used to submit additional parameters that the server has issued to the client (described in more detail later in this chapter).

HTTP Responses

A typical HTTP response is as follows:

```
HTTP/1.1 200 OK
Date: Tue, 19 Apr 2011 09:23:32 GMT
Server: Microsoft-IIS/6.0
X-Powered-By: ASP.NET
Set-Cookie: tracking=tI8rk7joMx44S2Uu85nSWc
X-AspNet-Version: 2.0.50727
Cache-Control: no-cache
Pragma: no-cache
Expires: Thu, 01 Jan 1970 00:00:00 GMT
Content-Type: text/html; charset=utf-8
Content-Length: 1067

<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN" "http://
www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd"><html xmlns="http://
www.w3.org/1999/xhtml" ><head><title>Your details</title>
...
```

The first line of every HTTP response consists of three items, separated by spaces:

- The HTTP version being used.
- A numeric status code indicating the result of the request. 200 is the most common status code; it means that the request was successful and that the requested resource is being returned.
- A textual "reason phrase" further describing the status of the response. This can have any value and is not used for any purpose by current browsers.

Here are some other points of interest in the response:

- The `Server` header contains a banner indicating the web server software being used, and sometimes other details such as installed modules and the server operating system. The information contained may or may not be accurate.
- The `Set-Cookie` header issues the browser a further cookie; this is submitted back in the `Cookie` header of subsequent requests to this server.
- The `Pragma` header instructs the browser not to store the response in its cache. The `Expires` header indicates that the response content expired in the past and therefore should not be cached. These instructions are frequently issued when dynamic content is being returned to ensure that browsers obtain a fresh version of this content on subsequent occasions.
- Almost all HTTP responses contain a message body following the blank line after the headers. The `Content-Type` header indicates that the body of this message contains an HTML document.
- The `Content-Length` header indicates the length of the message body in bytes.

HTTP Methods

When you are attacking web applications, you will be dealing almost exclusively with the most commonly used methods: `GET` and `POST`. You need to be aware of some important differences between these methods, as they can affect an application's security if overlooked.

The `GET` method is designed to retrieve resources. It can be used to send parameters to the requested resource in the URL query string. This enables users to bookmark a URL for a dynamic resource that they can reuse. Or other users can retrieve the equivalent resource on a subsequent occasion (as in a bookmarked search query). URLs are displayed on-screen and are logged in various places, such as the browser history and the web server's access logs. They are also transmitted in the `Referer` header to other sites when external

links are followed. For these reasons, the query string should not be used to transmit any sensitive information.

The `POST` method is designed to perform actions. With this method, request parameters can be sent both in the URL query string and in the body of the message. Although the URL can still be bookmarked, any parameters sent in the message body will be excluded from the bookmark. These parameters will also be excluded from the various locations in which logs of URLs are maintained and from the `Referer` header. Because the `POST` method is designed for performing actions, if a user clicks the browser's Back button to return to a page that was accessed using this method, the browser does not automatically reissue the request. Instead, it warns the user of what it is about to do, as shown in Figure 3-1. This prevents users from unwittingly performing an action more than once. For this reason, `POST` requests should always be used when an action is being performed.

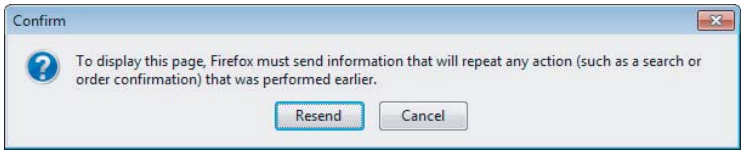

Figure 3-1: Browsers do not automatically reissue POST requests made by users, because these might cause an action to be performed more than once

In addition to the `GET` and `POST` methods, the HTTP protocol supports numerous other methods that have been created for specific purposes. Here are the other ones you are most likely to require knowledge of:

- `HEAD` functions in the same way as a `GET` request, except that the server should not return a message body in its response. The server should return the same headers that it would have returned to the corresponding `GET` request. Hence, this method can be used to check whether a resource is present before making a `GET` request for it.
- `TRACE` is designed for diagnostic purposes. The server should return in the response body the exact contents of the request message it received. This can be used to detect the effect of any proxy servers between the client and server that may manipulate the request.
- `OPTIONS` asks the server to report the HTTP methods that are available for a particular resource. The server typically returns a response containing an `Allow` header that lists the available methods.
- `PUT` attempts to upload the specified resource to the server, using the content contained in the body of the request. If this method is enabled, you may be able to leverage it to attack the application, such as by uploading an arbitrary script and executing it on the server.

Many other HTTP methods exist that are not directly relevant to attacking web applications. However, a web server may expose itself to attack if certain dangerous methods are available. See Chapter 18 for further details on these methods and examples of using them in an attack.

URLs

A uniform resource locator (URL) is a unique identifier for a web resource through which that resource can be retrieved. The format of most URLs is as follows:

```
protocol://hostname[:port]/[path/]file[?param=value]
```

Several components in this scheme are optional. The port number usually is included only if it differs from the default used by the relevant protocol. The URL used to generate the HTTP request shown earlier is as follows:

```
https://mdsec.net/auth/488/YourDetails.ashx?uid=129
```

In addition to this absolute form, URLs may be specified relative to a particular host, or relative to a particular path on that host. For example:

```
/auth/488/YourDetails.ashx?uid=129
YourDetails.ashx?uid=129
```

These relative forms are often used in web pages to describe navigation within the website or application itself.

NOTE **You may encounter the term *URI* (or uniform resource identifier) being used instead of URL, but it is really only used in formal specifications and by those who want to exhibit their pedantry.**

REST

Representational state transfer (REST) is a style of architecture for distributed systems in which requests and responses contain representations of the current state of the system's resources. The core technologies employed in the World Wide Web, including the HTTP protocol and the format of URLs, conform to the REST architectural style.

Although URLs containing parameters within the query string do themselves conform to REST constraints, the term "REST-style URL" is often used to signify a URL that contains its parameters within the URL file path, rather than the query string. For example, the following URL containing a query string:

```
http://wahh-app.com/search?make=ford&model=pinto
```

corresponds to the following URL containing "REST-style" parameters:

```
http://wahh-app.com/search/ford/pinto
```

Chapter 4 describes how you need to consider these different parameter styles when mapping an application's content and functionality and identifying its key attack surface.

HTTP Headers

HTTP supports a large number of headers, some of which are designed for specific unusual purposes. Some headers can be used for both requests and responses, and others are specific to one of these message types. The following sections describe the headers you are likely to encounter when attacking web applications.

General Headers

- `Connection` tells the other end of the communication whether it should close the TCP connection after the HTTP transmission has completed or keep it open for further messages.
- `Content-Encoding` specifies what kind of encoding is being used for the content contained in the message body, such as `gzip`, which is used by some applications to compress responses for faster transmission.
- `Content-Length` specifies the length of the message body, in bytes (except in the case of responses to `HEAD` requests, when it indicates the length of the body in the response to the corresponding `GET` request).
- `Content-Type` specifies the type of content contained in the message body, such as `text/html` for HTML documents.
- `Transfer-Encoding` specifies any encoding that was performed on the message body to facilitate its transfer over HTTP. It is normally used to specify chunked encoding when this is employed.

Request Headers

- `Accept` tells the server what kinds of content the client is willing to accept, such as image types, office document formats, and so on.
- `Accept-Encoding` tells the server what kinds of content encoding the client is willing to accept.
- `Authorization` submits credentials to the server for one of the built-in HTTP authentication types.
- `Cookie` submits cookies to the server that the server previously issued.
- `Host` specifies the hostname that appeared in the full URL being requested.

- `If-Modified-Since` specifies when the browser last received the requested resource. If the resource has not changed since that time, the server may instruct the client to use its cached copy, using a response with status code 304.
- `If-None-Match` specifies an *entity tag*, which is an identifier denoting the contents of the message body. The browser submits the entity tag that the server issued with the requested resource when it was last received. The server can use the entity tag to determine whether the browser may use its cached copy of the resource.
- `Origin` is used in cross-domain Ajax requests to indicate the domain from which the request originated (see Chapter 13).
- `Referer` specifies the URL from which the current request originated.
- `User-Agent` provides information about the browser or other client software that generated the request.

Response Headers

- `Access-Control-Allow-Origin` indicates whether the resource can be retrieved via cross-domain Ajax requests (see Chapter 13).
- `Cache-Control` passes caching directives to the browser (for example, `no-cache`).
- `ETag` specifies an entity tag. Clients can submit this identifier in future requests for the same resource in the `If-None-Match` header to notify the server which version of the resource the browser currently holds in its cache.
- `Expires` tells the browser for how long the contents of the message body are valid. The browser may use the cached copy of this resource until this time.
- `Location` is used in redirection responses (those that have a status code starting with 3) to specify the target of the redirect.
- `Pragma` passes caching directives to the browser (for example, `no-cache`).
- `Server` provides information about the web server software being used.
- `Set-Cookie` issues cookies to the browser that it will submit back to the server in subsequent requests.
- `WWW-Authenticate` is used in responses that have a 401 status code to provide details on the type(s) of authentication that the server supports.
- `X-Frame-Options` indicates whether and how the current response may be loaded within a browser frame (see Chapter 13).

Cookies

Cookies are a key part of the HTTP protocol that most web applications rely on. Frequently they can be used as a vehicle for exploiting vulnerabilities. The cookie mechanism enables the server to send items of data to the client, which the client stores and resubmits to the server. Unlike the other types of request parameters (those within the URL query string or the message body), cookies continue to be resubmitted in each subsequent request without any particular action required by the application or the user.

A server issues a cookie using the `Set-Cookie` response header, as you have seen:

```
Set-Cookie: tracking=tI8rk7joMx44S2Uu85nSWc
```

The user's browser then automatically adds the following header to subsequent requests back to the same server:

```
Cookie: tracking=tI8rk7joMx44S2Uu85nSWc
```

Cookies normally consist of a name/value pair, as shown, but they may consist of any string that does not contain a space. Multiple cookies can be issued by using multiple `Set-Cookie` headers in the server's response. These are submitted back to the server in the same `Cookie` header, with a semicolon separating different individual cookies.

In addition to the cookie's actual value, the `Set-Cookie` header can include any of the following optional attributes, which can be used to control how the browser handles the cookie:

- `expires` sets a date until which the cookie is valid. This causes the browser to save the cookie to persistent storage, and it is reused in subsequent browser sessions until the expiration date is reached. If this attribute is not set, the cookie is used only in the current browser session.
- `domain` specifies the domain for which the cookie is valid. This must be the same or a parent of the domain from which the cookie is received.
- `path` specifies the URL path for which the cookie is valid.
- `secure` — If this attribute is set, the cookie will be submitted only in HTTPS requests.
- `HttpOnly` — If this attribute is set, the cookie cannot be directly accessed via client-side JavaScript.

Each of these cookie attributes can impact the application's security. The primary impact is on the attacker's ability to directly target other users of the application. See Chapters 12 and 13 for more details.

Status Codes

Each HTTP response message must contain a status code in its first line, indicating the result of the request. The status codes fall into five groups, according to the code's first digit:

- **1*xx*** — Informational.
- **2*xx*** — The request was successful.
- **3*xx*** — The client is redirected to a different resource.
- **4*xx*** — The request contains an error of some kind.
- **5*xx*** — The server encountered an error fulfilling the request.

There are numerous specific status codes, many of which are used only in specialized circumstances. Here are the status codes you are most likely to encounter when attacking a web application, along with the usual reason phrase associated with them:

- `100 Continue` is sent in some circumstances when a client submits a request containing a body. The response indicates that the request headers were received and that the client should continue sending the body. The server returns a second response when the request has been completed.
- `200 OK` indicates that the request was successful and that the response body contains the result of the request.
- `201 Created` is returned in response to a `PUT` request to indicate that the request was successful.
- `301 Moved Permanently` redirects the browser permanently to a different URL, which is specified in the `Location` header. The client should use the new URL in the future rather than the original.
- `302 Found` redirects the browser temporarily to a different URL, which is specified in the `Location` header. The client should revert to the original URL in subsequent requests.
- `304 Not Modified` instructs the browser to use its cached copy of the requested resource. The server uses the `If-Modified-Since` and `If-None-Match` request headers to determine whether the client has the latest version of the resource.
- `400 Bad Request` indicates that the client submitted an invalid HTTP request. You will probably encounter this when you have modified a request in certain invalid ways, such as by placing a space character into the URL.
- `401 Unauthorized` indicates that the server requires HTTP authentication before the request will be granted. The `WWW-Authenticate` header contains details on the type(s) of authentication supported.

- `403 Forbidden` indicates that no one is allowed to access the requested resource, regardless of authentication.
- `404 Not Found` indicates that the requested resource does not exist.
- `405 Method Not Allowed` indicates that the method used in the request is not supported for the specified URL. For example, you may receive this status code if you attempt to use the `PUT` method where it is not supported.
- `413 Request Entity Too Large` — If you are probing for buffer overflow vulnerabilities in native code, and therefore are submitting long strings of data, this indicates that the body of your request is too large for the server to handle.
- `414 Request URI Too Long` is similar to the 413 response. It indicates that the URL used in the request is too large for the server to handle.
- `500 Internal Server Error` indicates that the server encountered an error fulfilling the request. This normally occurs when you have submitted unexpected input that caused an unhandled error somewhere within the application's processing. You should closely review the full contents of the server's response for any details indicating the nature of the error.
- `503 Service Unavailable` normally indicates that, although the web server itself is functioning and can respond to requests, the application accessed via the server is not responding. You should verify whether this is the result of any action you have performed.

HTTPS

The HTTP protocol uses plain TCP as its transport mechanism, which is unencrypted and therefore can be intercepted by an attacker who is suitably positioned on the network. HTTPS is essentially the same application-layer protocol as HTTP but is tunneled over the secure transport mechanism, Secure Sockets Layer (SSL). This protects the privacy and integrity of data passing over the network, reducing the possibilities for noninvasive interception attacks. HTTP requests and responses function in exactly the same way regardless of whether SSL is used for transport.

NOTE **SSL has strictly been superseded by transport layer security (TLS), but the latter usually still is referred to using the older name.**

HTTP Proxies

An HTTP proxy is a server that mediates access between the client browser and the destination web server. When a browser has been configured to use a proxy

server, it makes all its requests to that server. The proxy relays the requests to the relevant web servers and forwards their responses back to the browser. Most proxies also provide additional services, including caching, authentication, and access control.

You should be aware of two differences in how HTTP works when a proxy server is being used:

- When a browser issues an unencrypted HTTP request to a proxy server, it places the full URL into the request, including the protocol prefix `http://`, the server's hostname, and the port number if this is nonstandard. The proxy server extracts the hostname and port and uses these to direct the request to the correct destination web server.
- When HTTPS is being used, the browser cannot perform the SSL handshake with the proxy server, because this would break the secure tunnel and leave the communications vulnerable to interception attacks. Hence, the browser must use the proxy as a pure TCP-level relay, which passes all network data in both directions between the browser and the destination web server, with which the browser performs an SSL handshake as normal. To establish this relay, the browser makes an HTTP request to the proxy server using the `CONNECT` method and specifying the destination hostname and port number as the URL. If the proxy allows the request, it returns an HTTP response with a 200 status, keeps the TCP connection open, and from that point onward acts as a pure TCP-level relay to the destination web server.

By some measure, the most useful item in your toolkit when attacking web applications is a specialized kind of proxy server that sits between your browser and the target website and allows you to intercept and modify all requests and responses, even those using HTTPS. We will begin examining how you can use this kind of tool in the next chapter.

HTTP Authentication

The HTTP protocol includes its own mechanisms for authenticating users using various authentication schemes, including the following:

- **Basic** is a simple authentication mechanism that sends user credentials as a Base64-encoded string in a request header with each message.
- **NTLM** is a challenge-response mechanism and uses a version of the Windows NTLM protocol.
- **Digest** is a challenge-response mechanism and uses MD5 checksums of a nonce with the user's credentials.

It is relatively rare to encounter these authentication protocols being used by web applications deployed on the Internet. They are more commonly used within organizations to access intranet-based services.

COMMON MYTH

"Basic authentication is insecure."

Because basic authentication places credentials in unencrypted form within the HTTP request, it is frequently stated that the protocol is insecure and should not be used. But forms-based authentication, as used by numerous banks, also places credentials in unencrypted form within the HTTP request.

Any HTTP message can be protected from eavesdropping attacks by using HTTPS as a transport mechanism, which should be done by every security-conscious application. In relation to eavesdropping, at least, basic authentication in itself is no worse than the methods used by the majority of today's web applications.

Web Functionality

In addition to the core communications protocol used to send messages between client and server, web applications employ numerous technologies to deliver their functionality. Any reasonably functional application may employ dozens of distinct technologies within its server and client components. Before you can mount a serious attack against a web application, you need a basic understanding of how its functionality is implemented, how the technologies used are designed to behave, and where their weak points are likely to lie.

Server-Side Functionality

The early World Wide Web contained entirely static content. Websites consisted of various resources such as HTML pages and images, which were simply loaded onto a web server and delivered to any user who requested them. Each time a particular resource was requested, the server responded with the same content.

Today's web applications still typically employ a fair number of static resources. However, a large amount of the content that they present to users is generated dynamically. When a user requests a dynamic resource, the server's response is created on the fly, and each user may receive content that is uniquely customized for him or her.

Dynamic content is generated by scripts or other code executing on the server. These scripts are akin to computer programs in their own right. They have various inputs, perform processing on these, and return their outputs to the user.

When a user's browser requests a dynamic resource, normally it does not simply ask for a copy of that resource. In general, it also submits various parameters along with its request. It is these parameters that enable the server-side application to generate content that is tailored to the individual user. HTTP requests can be used to send parameters to the application in three main ways:

- In the URL query string
- In the file path of REST-style URLs
- In HTTP cookies
- In the body of requests using the `POST` method

In addition to these primary sources of input, the server-side application may in principle use any part of the HTTP request as an input to its processing. For example, an application may process the `User-Agent` header to generate content that is optimized for the type of browser being used.

Like computer software in general, web applications employ a wide range of technologies on the server side to deliver their functionality:

- Scripting languages such as PHP, VBScript, and Perl
- Web application platforms such as ASP.NET and Java
- Web servers such as Apache, IIS, and Netscape Enterprise
- Databases such as MS-SQL, Oracle, and MySQL
- Other back-end components such as filesystems, SOAP-based web services, and directory services

All these technologies and the types of vulnerabilities that can arise in relation to them are examined in detail throughout this book. Some of the most common web application platforms and technologies you are likely to encounter are described in the following sections.

COMMON MYTH

"Our applications need only cursory security review, because they employ a well-used framework."

Use of a well-used framework is often a cause for complacency in web application development, on the assumption that common vulnerabilities such as SQL injection are automatically avoided. This assumption is mistaken for two reasons.

First, a large number of web application vulnerabilities arise in an application's design, not its implementation, and are independent of the development framework or language chosen.

Second, because a framework typically employs plug-ins and packages from the cutting edge of the latest repositories, it is likely that these packages have not undergone security review. Interestingly, if a vulnerability is later found in the application, the same proponents of the myth will readily swap sides and blame their framework or third-party package!

The Java Platform

For many years, the Java Platform, Enterprise Edition (formerly known as J2EE) was a de facto standard for large-scale enterprise applications. Originally developed by Sun Microsystems and now owned by Oracle, it lends itself to multitiered and load-balanced architectures and is well suited to modular development and code reuse. Because of its long history and widespread adoption, many high-quality development tools, application servers, and frameworks are available to assist developers. The Java Platform can be run on several underlying operating systems, including Windows, Linux, and Solaris.

Descriptions of Java-based web applications often employ a number of potentially confusing terms that you may need to be aware of:

- An **Enterprise Java Bean** (EJB) is a relatively heavyweight software component that encapsulates the logic of a specific business function within the application. EJBs are intended to take care of various technical challenges that application developers must address, such as transactional integrity.
- A **Plain Old Java Object** (POJO) is an ordinary Java object, as distinct from a special object such as an EJB. A POJO normally is used to denote objects that are user-defined and are much simpler and more lightweight than EJBs and those used in other frameworks.
- A **Java Servlet** is an object that resides on an application server and receives HTTP requests from clients and returns HTTP responses. Servlet implementations can use numerous interfaces to facilitate the development of useful applications.
- A Java **web container** is a platform or engine that provides a runtime environment for Java-based web applications. Examples of Java web containers are Apache Tomcat, BEA WebLogic, and JBoss.

Many Java web applications employ third-party and open source components alongside custom-built code. This is an attractive option because it reduces development effort, and Java is well suited to this modular approach. Here are some examples of components commonly used for key application functions:

- **Authentication** — JAAS, ACEGI
- **Presentation layer** — SiteMesh, Tapestry

- **Database object relational mapping** — Hibernate
- **Logging** — Log4J

If you can determine which open source packages are used in the application you are attacking, you can download these and perform a code review or install them to experiment on. A vulnerability in any of these may be exploitable to compromise the wider application.

ASP.NET

ASP.NET is Microsoft's web application framework and is a direct competitor to the Java Platform. ASP.NET is several years younger than its counterpart but has made significant inroads into Java's territory.

ASP.NET uses Microsoft's .NET Framework, which provides a virtual machine (the Common Language Runtime) and a set of powerful APIs. Hence, ASP.NET applications can be written in any .NET language, such as C# or VB.NET.

ASP.NET lends itself to the event-driven programming paradigm that is normally used in conventional desktop software, rather than the script-based approach used in most earlier web application frameworks. This, together with the powerful development tools provided with Visual Studio, makes developing a functional web application extremely easy for anyone with minimal programming skills.

The ASP.NET framework helps protect against some common web application vulnerabilities such as cross-site scripting, without requiring any effort from the developer. However, one practical downside of its apparent simplicity is that many small-scale ASP.NET applications are actually created by beginners who lack any awareness of the core security problems faced by web applications.

PHP

The PHP language emerged from a hobby project (the acronym originally stood for "personal home page"). It has since evolved almost unrecognizably into a highly powerful and rich framework for developing web applications. It is often used in conjunction with other free technologies in what is known as the LAMP stack (composed of Linux as the operating system, Apache as the web server, MySQL as the database server, and PHP as the programming language for the web application).

Numerous open source applications and components have been developed using PHP. Many of these provide off-the-shelf solutions for common application functions, which are often incorporated into wider custom-built applications:

- **Bulletin boards** — PHPBB, PHP-Nuke
- **Administrative front ends** — PHPMyAdmin

- **Web mail** — SquirrelMail, IlohaMail
- **Photo galleries** — Gallery
- **Shopping carts** — osCommerce, ECW-Shop
- **Wikis** — MediaWiki, WakkaWikki

Because PHP is free and easy to use, it has often been the language of choice for many beginners writing web applications. Furthermore, the design and default configuration of the PHP framework has historically made it easy for programmers to unwittingly introduce security bugs into their code. These factors have meant that applications written in PHP have suffered from a disproportionate number of security vulnerabilities. In addition, several defects have existed within the PHP platform itself that often could be exploited via applications running on it. See Chapter 19 for details on common defects arising in PHP applications.

Ruby on Rails

Rails 1.0 was released in 2005, with strong emphasis on Model-View-Controller architecture. A key strength of Rails is the breakneck speed with which fully fledged data-driven applications can be created. If a developer follows the Rails coding style and naming conventions, Rails can autogenerate a model for database content, controller actions for modifying it, and default views for the application user. As with any highly functional new technology, several vulnerabilities have been found in Ruby on Rails, including the ability to bypass a "safe mode," analogous to that found in PHP.

More details on recent vulnerabilities can be found here:

```
www.ruby-lang.org/en/security/
```

SQL

Structured Query Language (SQL) is used to access data in relational databases, such as Oracle, MS-SQL server and MySQL. The vast majority of today's web applications employ SQL-based databases as their back-end data store, and nearly all application functions involve interaction with these data stores in some way.

Relational databases store data in tables, each of which contains a number of rows and columns. Each column represents a data field, such as "name" or "e-mail address," and each row represents an item with values assigned to some or all of these fields.

SQL uses queries to perform common tasks such as reading, adding, updating, and deleting data. For example, to retrieve a user's e-mail address with a specified name, an application might perform the following query:

```
select email from users where name = 'daf'
```

To implement the functionality they need, web applications may incorporate user-supplied input into SQL queries that are executed by the back-end database. If this process is not carried out safely, attackers may be able to submit malicious input to interfere with the database and potentially read and write sensitive data. These attacks are described in Chapter 9, along with detailed explanations of the SQL language and how it can be used.

XML

Extensible Markup Language (XML) is a specification for encoding data in a machine-readable form. Like any markup language, the XML format separates a document into content (which is data) and markup (which annotates the data).

Markup is primarily represented using tags, which may be start tags, end tags, or empty-element tags:

```
<tagname>
</tagname>
<tagname />
```

Start and end tags are paired into elements and may encapsulate document content or child elements:

```
<pet>ginger</pet>
<pets><dog>spot</dog><cat>paws</cat></pets>
```

Tags may include attributes, which are name/value pairs:

```
<data version="2.1"><pets>...</pets></data>
```

XML is extensible in that it allows arbitrary tag and attribute names. XML documents often include a Document Type Definition (DTD), which defines the tags and attributes used in the documents and the ways in which they can be combined.

XML and technologies derived from it are used extensively in web applications, on both the server and client side, as described in later sections of this chapter.

Web Services

Although this book covers web application hacking, many of the vulnerabilities described are equally applicable to web services. In fact, many applications are essentially a GUI front-end to a set of back-end web services.

Web services use Simple Object Access Protocol (SOAP) to exchange data. SOAP typically uses the HTTP protocol to transmit messages and represents data using the XML format.

A typical SOAP request is as follows:

```
POST /doTransfer.asp HTTP/1.0
Host: mdsec-mgr.int.mdsec.net
Content-Type: application/soap+xml; charset=utf-8
Content-Length: 891
<?xml version="1.0"?>
<soap:Envelope xmlns:soap="http://www.w3.org/2001/12/soap-envelope">
  <soap:Body>
      <pre:Add xmlns:pre=http://target/lists soap:encodingStyle=
"http://www.w3.org/2001/12/soap-encoding">
      <Account>
        <FromAccount>18281008</FromAccount>
        <Amount>1430</Amount>
        <ClearedFunds>False</ClearedFunds>
        <ToAccount>08447656</ToAccount>
      </Account>
    </pre:Add>
  </soap:Body>
</soap:Envelope>
```

In the context of web applications accessed using a browser, you are most likely to encounter SOAP being used by the server-side application to communicate with various back-end systems. If user-supplied data is incorporated directly into back-end SOAP messages, similar vulnerabilities can arise as for SQL. These issues are described in detail in Chapter 10.

If a web application also exposes web services directly, these are also worthy of examination. Even if the front-end application is simply written on top of the web service, differences may exist in input handling and in the functionality exposed by the services themselves. The server normally publishes the available services and parameters using the Web Services Description Language (WSDL) format. Tools such as soapUI can be used to create sample requests based on a published WSDL file to call the authentication web service, gain an authentication token, and make any subsequent web service requests.

Client-Side Functionality

For the server-side application to receive user input and actions and present the results to the user, it needs to provide a client-side user interface. Because all web applications are accessed via a web browser, these interfaces all share a

common core of technologies. However, these have been built upon in various, diverse ways, and the ways in which applications leverage client-side technology has continued to evolve rapidly in recent years.

HTML

The core technology used to build web interfaces is hypertext markup language (HTML). Like XML, HTML is a tag-based language that is used to describe the structure of documents that are rendered within the browser. From its simple beginnings as a means of providing basic formatting for text documents, HTML has developed into a rich and powerful language that can be used to create highly complex and functional user interfaces.

XHTML is a development of HTML that is based on XML and that has a stricter specification than older versions of HTML. Part of the motivation for XHTML was the need to move toward a more rigid standard for HTML markup to avoid the various compromises and security issues that can arise when browsers are obligated to tolerate less-strict forms of HTML.

More details about HTML and related technologies appear in the following sections.

Hyperlinks

A large amount of communication from client to server is driven by the user's clicking on hyperlinks. In web applications, hyperlinks frequently contain preset request parameters. These are items of data that the user never enters; they are submitted because the server places them into the target URL of the hyperlink that the user clicks. For example, a web application might present a series of links to news stories, each having the following form:

```
<a href="?redir=/updates/update29.html">What's happening?</a>
```

When a user clicks this link, the browser makes the following request:

```
GET /news/8/?redir=/updates/update29.html HTTP/1.1
Host: mdsec.net
...
```

The server receives the `redir` parameter in the query string and uses its value to determine what content should be presented to the user.

Forms

Although hyperlink-based navigation is responsible for a large amount of client-to-server communications, most web applications need more flexible ways to gather input and receive actions from users. HTML forms are the usual

mechanism for allowing users to enter arbitrary input via their browser. A typical form is as follows:

```
<form action="/secure/login.php?app=quotations" method="post">
username: <input type="text" name="username"><br>
password: <input type="password" name="password">
<input type="hidden" name="redir" value="/secure/home.php">
<input type="submit" name="submit" value="log in">
</form>
```

When the user enters values into the form and clicks the Submit button, the browser makes a request like the following:

```
POST /secure/login.php?app=quotations HTTP/1.1
Host: wahh-app.com
Content-Type: application/x-www-form-urlencoded
Content-Length: 39
Cookie: SESS=GTnrpx2ss2tSWSnhXJGyG0LJ47MXRsjcFM6Bd

username=daf&password=foo&redir=/secure/home.php&submit=log+in
```

In this request, several points of interest reflect how different aspects of the request are used to control server-side processing:

- Because the HTML `form` tag contains an attribute specifying the `POST` method, the browser uses this method to submit the form and places the data from the form into the body of the request message.
- In addition to the two items of data that the user enters, the form contains a hidden parameter (`redir`) and a submit parameter (`submit`). Both of these are submitted in the request and may be used by the server-side application to control its logic.
- The target URL for the form submission contains a preset parameter (`app`), as in the hyperlink example shown previously. This parameter may be used to control the server-side processing.
- The request contains a cookie parameter (`SESS`), which was issued to the browser in an earlier response from the server. This parameter may be used to control the server-side processing.

The preceding request contains a header specifying that the type of content in the message body is `x-www-form-urlencoded`. This means that parameters are represented in the message body as name/value pairs in the same way as they are in the URL query string. The other content type you are likely to encounter when form data is submitted is `multipart/form-data`. An application can request that browsers use multipart encoding by specifying this in an `enctype` attribute in the form tag. With this form of encoding, the `Content-Type` header in the request also specifies a random string that is used as a separator for the

parameters contained in the request body. For example, if the form specified multipart encoding, the resulting request would look like the following:

```
POST /secure/login.php?app=quotations HTTP/1.1
Host: wahh-app.com
Content-Type: multipart/form-data; boundary=------------7d71385d0a1a
Content-Length: 369
Cookie: SESS=GTnrpx2ss2tSWSnhXJGyG0LJ47MXRsjcFM6Bd

------------7d71385d0a1a
Content-Disposition: form-data; name="username"

daf
------------7d71385d0a1a
Content-Disposition: form-data; name="password"

foo
------------7d71385d0a1a
Content-Disposition: form-data; name="redir"

/secure/home.php
------------7d71385d0a1a
Content-Disposition: form-data; name="submit"

log in
------------7d71385d0a1a--
```

CSS

Cascading Style Sheets (CSS) is a language used to describe the presentation of a document written in a markup language. Within web applications, it is used to specify how HTML content should be rendered on-screen (and in other media, such as the printed page).

Modern web standards aim to separate as much as possible the content of a document from its presentation. This separation has numerous benefits, including simpler and smaller HTML pages, easier updating of formatting across a website, and improved accessibility.

CSS is based on formatting rules that can be defined with different levels of specificity. Where multiple rules match an individual document element, different attributes defined in those rules can "cascade" through these rules so that the appropriate combination of style attributes is applied to the element.

CSS syntax uses selectors to define a class of markup elements to which a given set of attributes should be applied. For example, the following CSS rule defines the foreground color for headings that are marked up using `<h2>` tags:

```
h2 { color: red; }
```

In the earliest days of web application security, CSS was largely overlooked and was considered to have no security implications. Today, CSS is increasingly relevant both as a source of security vulnerabilities in its own right and as a means of delivering effective exploits for other categories of vulnerabilities (see Chapters 12 and 13 for more information).

JavaScript

Hyperlinks and forms can be used to create a rich user interface that can easily gather most kinds of input that web applications require. However, most applications employ a more distributed model, in which the client side is used not simply to submit user data and actions but also to perform actual processing of data. This is done for two primary reasons:

- It can improve the application's performance, because certain tasks can be carried out entirely on the client component, without needing to make a round trip of request and response to the server.
- It can enhance usability, because parts of the user interface can be dynamically updated in response to user actions, without needing to load an entirely new HTML page delivered by the server.

JavaScript is a relatively simple but powerful programming language that can be easily used to extend web interfaces in ways that are not possible using HTML alone. It is commonly used to perform the following tasks:

- Validating user-entered data before it is submitted to the server to avoid unnecessary requests if the data contains errors
- Dynamically modifying the user interface in response to user actions — for example, to implement drop-down menus and other controls familiar from non-web interfaces
- Querying and updating the document object model (DOM) within the browser to control the browser's behavior (the browser DOM is described in a moment)

VBScript

VBScript is an alternative to JavaScript that is supported only in the Internet Explorer browser. It is modeled on Visual Basic and allows interaction with the browser DOM. But in general it is somewhat less powerful and developed than JavaScript.

Due to its browser-specific nature, VBScript is scarcely used in today's web applications. Its main interest from a security perspective is as a means of delivering exploits for vulnerabilities such as cross-site scripting in occasional situations where an exploit using JavaScript is not feasible (see Chapter 12).

Document Object Model

The Document Object Model (DOM) is an abstract representation of an HTML document that can be queried and manipulated through its API.

The DOM allows client-side scripts to access individual HTML elements by their `id` and to traverse the structure of elements programmatically. Data such as the current URL and cookies can also be read and updated. The DOM also includes an event model, allowing code to hook events such as form submission, navigation via links, and keystrokes.

Manipulation of the browser DOM is a key technique used in Ajax-based applications, as described in the following section.

Ajax

Ajax is a collection of programming techniques used on the client side to create user interfaces that aim to mimic the smooth interaction and dynamic behavior of traditional desktop applications.

The name originally was an acronym for "Asynchronous JavaScript and XML," although in today's web Ajax requests need not be asynchronous and need not employ XML.

The earliest web applications were based on complete pages. Each user action, such as clicking a link or submitting a form, initiated a window-level navigation event, causing a new page to be loaded from the server. This approach resulted in a disjointed user experience, with noticeable delays while large responses were received from the server and the whole page was rerendered.

With Ajax, some user actions are handled within client-side script code and do not cause a full reload of the page. Instead, the script performs a request "in the background" and typically receives a much smaller response that is used to dynamically update only part of the user interface. For example, in an Ajax-based shopping application, clicking an Add to Cart button may cause a background request that updates the server-side record of the user's shopping cart and a lightweight response that updates the number of cart items showing on the user's screen. Virtually the entire existing page remains unmodified within the browser, providing a much faster and more satisfying experience for the user.

The core technology used in Ajax is `XMLHttpRequest`. After a certain consolidation of standards, this is now a native JavaScript object that client-side scripts can use to make "background" requests without requiring a window-level navigation event. Despite its name, `XMLHttpRequest` allows arbitrary content to be sent in requests and received in responses. Although many Ajax applications do use XML to format message data, an increasing number have opted to exchange data using other methods of representation. (See the next section for one example.)

Note that although most Ajax applications do use asynchronous communications with the server, this is not essential. In some situations, it may actually make

more sense to prevent user interaction with the application while a particular action is carried out. In these situations, Ajax is still beneficial in providing a more seamless experience by avoiding the need to reload an entire page.

Historically, the use of Ajax has introduced some new types of vulnerabilities into web applications. More broadly, it also increases the attack surface of a typical application by introducing more potential targets for attack on both the server and client side. Ajax techniques are also available for use by attackers when they are devising more effective exploits for other vulnerabilities. See Chapters 12 and 13 for more details.

JSON

JavaScript Object Notation (JSON) is a simple data transfer format that can be used to serialize arbitrary data. It can be processed directly by JavaScript interpreters. It is commonly employed in Ajax applications as an alternative to the XML format originally used for data transmission. In a typical situation, when a user performs an action, client-side JavaScript uses `XMLHttpRequest` to communicate the action to the server. The server returns a lightweight response containing data in JSON format. The client-side script then processes this data and updates the user interface accordingly.

For example, an Ajax-based web mail application may contain a feature to show the details of a selected contact. When a user clicks a contact, the browser uses `XMLHttpRequest` to retrieve the details of the selected contact, which are returned using JSON:

```
{
    "name": "Mike Kemp",
    "id": "8041148671",
    "email": "fkwitt@layerone.com"
}
```

The client-side script uses the JavaScript interpreter to consume the JSON response and updates the relevant part of the user interface based on its contents.

A further location where you may encounter JSON data in today's applications is as a means of encapsulating data within conventional request parameters. For example, when the user updates the details of a contact, the new information might be communicated to the server using the following request:

```
POST /contacts HTTP/1.0
Content-Type: application/x-www-form-urlencoded
Content-Length: 89

Contact={"name":"Mike Kemp","id":"8041148671","email":"pikey@
clappymonkey.com"}
&submit=update
```

Same-Origin Policy

The same-origin policy is a key mechanism implemented within browsers that is designed to keep content that came from different origins from interfering with each other. Basically, content received from one website is allowed to read and modify other content received from the same site but is not allowed to access content received from other sites.

If the same-origin policy did not exist, and an unwitting user browsed to a malicious website, script code running on that site could access the data and functionality of any other website also visited by the user. This may enable the malicious site to perform funds transfers from the user's online bank, read his or her web mail, or capture credit card details when the user shops online. For this reason, browsers implement restrictions to allow this type of interaction only with content that has been received from the same origin.

In practice, applying this concept to the details of different web features and technologies leads to various complications and compromises. Here are some key features of the same-origin policy that you need to be aware of:

- A page residing on one domain can cause an arbitrary request to be made to another domain (for example, by submitting a form or loading an image). But it cannot itself process the data returned from that request.
- A page residing on one domain can load a script from another domain and execute this within its own context. This is because scripts are assumed to contain code, rather than data, so cross-domain access should not lead to disclosure of any sensitive information.
- A page residing on one domain cannot read or modify the cookies or other DOM data belonging to another domain.

These features can lead to various cross-domain attacks, such as inducing user actions and capturing data. Further complications arise with browser extension technologies, which implement same-origin restrictions in different ways. These issues are discussed in detail in Chapter 13.

HTML5

HTML5 is a major update to the HTML standard. HTML5 currently is still under development and is only partially implemented within browsers.

From a security perspective, HTML5 is primarily of interest for the following reasons:

- It introduces various new tags, attributes, and APIs that can be leveraged to deliver cross-site scripting and other attacks, as described in Chapter 12.

- It modifies the core Ajax technology, `XMLHttpRequest`, to enable two-way cross-domain interaction in certain situations. This can lead to new cross-domain attacks, as described in Chapter 13.
- It introduces new mechanisms for client-side data storage, which can lead to user privacy issues, and new categories of attack such as client-side SQL injection, as described in Chapter 13.

"Web 2.0"

This buzzword has become fashionable in recent years as a rather loose and nebulous name for a range of related trends in web applications, including the following:

- Heavy use of Ajax for performing asynchronous, behind-the-scenes requests
- Increased cross-domain integration using various techniques
- Use of new technologies on the client side, including XML, JSON, and Flex
- More prominent functionality supporting user-generated content, information sharing, and interaction

As with all changes in technology, these trends present new opportunities for security vulnerabilities to arise. However, they do not define a clear subset of web application security issues in general. The vulnerabilities that occur in these contexts are largely the same as, or closely derived from, types of vulnerabilities that preceded these trends. In general, talking about "Web 2.0 Security" usually represents a category mistake that does not facilitate clear thinking about the issues that matter.

Browser Extension Technologies

Going beyond the capabilities of JavaScript, some web applications employ browser extension technologies that use custom code to extend the browser's built-in capabilities in arbitrary ways. These components may be deployed as bytecode that is executed by a suitable browser plug-in or may involve installing native executables onto the client computer itself. The thick-client technologies you are likely to encounter when attacking web applications are

- Java applets
- ActiveX controls
- Flash objects
- Silverlight objects

These technologies are described in detail in Chapter 5.

State and Sessions

The technologies described so far enable the server and client components of a web application to exchange and process data in numerous ways. To implement most kinds of useful functionality, however, applications need to track the state of each user's interaction with the application across multiple requests. For example, a shopping application may allow users to browse a product catalog, add items to a cart, view and update the cart contents, proceed to checkout, and provide personal and payment details.

To make this kind of functionality possible, the application must maintain a set of stateful data generated by the user's actions across several requests. This data normally is held within a server-side structure called a session. When a user performs an action, such as adding an item to her shopping cart, the server-side application updates the relevant details within the user's session. When the user later views the contents of her cart, data from the session is used to return the correct information to the user.

In some applications, state information is stored on the client component rather than the server. The current set of data is passed to the client in each server response and is sent back to the server in each client request. Of course, because the user may modify any data transmitted via the client component, applications need to protect themselves from attackers who may change this state information in an attempt to interfere with the application's logic. The ASP.NET platform makes use of a hidden form field called `ViewState` to store state information about the user's web interface and thereby reduce overhead on the server. By default, the contents of the `ViewState` include a keyed hash to prevent tampering.

Because the HTTP protocol is itself stateless, most applications need a way to reidentify individual users across multiple requests for the correct set of state data to be used to process each request. Normally this is achieved by issuing each user a token that uniquely identifies that user's session. These tokens may be transmitted using any type of request parameter, but most applications use HTTP cookies. Several kinds of vulnerabilities arise in relation to session handling, as described in detail in Chapter 7.

Encoding Schemes

Web applications employ several different encoding schemes for their data. Both the HTTP protocol and the HTML language are historically text-based, and different encoding schemes have been devised to ensure that these mechanisms can safely handle unusual characters and binary data. When you are attacking a web application, you will frequently need to encode data using a relevant

scheme to ensure that it is handled in the way you intend. Furthermore, in many cases you may be able to manipulate the encoding schemes an application uses to cause behavior that its designers did not intend.

URL Encoding

URLs are permitted to contain only the printable characters in the US-ASCII character set — that is, those whose ASCII code is in the range 0x20 to 0x7e, inclusive. Furthermore, several characters within this range are restricted because they have special meaning within the URL scheme itself or within the HTTP protocol.

The URL-encoding scheme is used to encode any problematic characters within the extended ASCII character set so that they can be safely transported over HTTP. The URL-encoded form of any character is the % prefix followed by the character's two-digit ASCII code expressed in hexadecimal. Here are some characters that are commonly URL-encoded:

- `%3d` — =
- `%25` — %
- `%20` — Space
- `%0a` — New line
- `%00` — Null byte

A further encoding to be aware of is the `+` character, which represents a URL-encoded space (in addition to the `%20` representation of a space).

NOTE **For the purpose of attacking web applications, you should URL-encode any of the following characters when you insert them *as data* into an HTTP request:**

```
space % ? & = ; + #
```

(Of course, you will often need to use these characters with their special meaning when modifying a request — for example, to add a request parameter to the query string. In this case, they should be used in their literal form.)

Unicode Encoding

Unicode is a character encoding standard that is designed to support all of the world's writing systems. It employs various encoding schemes, some of which can be used to represent unusual characters in web applications.

16-bit Unicode encoding works in a similar way to URL encoding. For transmission over HTTP, the 16-bit Unicode-encoded form of a character is

the `%u` prefix followed by the character's Unicode code point expressed in hexadecimal:

- `%u2215` — ∕
- `%u00e9` — é

UTF-8 is a variable-length encoding standard that employs one or more bytes to express each character. For transmission over HTTP, the UTF-8-encoded form of a multibyte character simply uses each byte expressed in hexadecimal and preceded by the `%` prefix:

- `%c2%a9` — ©
- `%e2%89%a0` — ≠

For the purpose of attacking web applications, Unicode encoding is primarily of interest because it can sometimes be used to defeat input validation mechanisms. If an input filter blocks certain malicious expressions, but the component that subsequently processes the input understands Unicode encoding, it may be possible to bypass the filter using various standard and malformed Unicode encodings.

HTML Encoding

HTML encoding is used to represent problematic characters so that they can be safely incorporated into an HTML document. Various characters have special meaning as metacharacters within HTML and are used to define a document's structure rather than its content. To use these characters safely as part of the document's content, it is necessary to HTML-encode them.

HTML encoding defines numerous HTML entities to represent specific literal characters:

- `&quot;` — "
- `&apos;` — '
- `&` — &
- `<` — <
- `>` — >

In addition, any character can be HTML-encoded using its ASCII code in decimal form:

- `&#34;` — "
- `&#39;` — '

or by using its ASCII code in hexadecimal form (prefixed by an `x`):

- `&#x22;` — "
- `&#x27;` — '

When you are attacking a web application, your main interest in HTML encoding is likely to be when probing for cross-site scripting vulnerabilities. If an application returns user input unmodified within its responses, it is probably vulnerable, whereas if dangerous characters are HTML-encoded, it may be safe. See Chapter 12 for more details on these vulnerabilities.

Base64 Encoding

Base64 encoding allows any binary data to be safely represented using only printable ASCII characters. It is commonly used to encode e-mail attachments for safe transmission over SMTP. It is also used to encode user credentials in basic HTTP authentication.

Base64 encoding processes input data in blocks of three bytes. Each of these blocks is divided into four chunks of six bits each. Six bits of data allows for 64 different possible permutations, so each chunk can be represented using a set of 64 characters. Base64 encoding employs the following character set, which contains only printable ASCII characters:

```
ABCDEFGHIJKLMNOPQRSTUVWXYZabcdefghijklmnopqrstuvwxyz0123456789+/
```

If the final block of input data results in fewer than three chunks of output data, the output is padded with one or two = characters.

For example, here is the Base64-encoded form of *The Web Application Hacker's Handbook*:

```
VGhlIFdlYiBBcHBsaWNhdGlvbiBIYWNrZXIncyBIYW5kYm9vaw==
```

Many web applications use Base64 encoding to transmit binary data within cookies and other parameters, and even to obfuscate (that is, to hide) sensitive data to prevent trivial modification. You should always look out for, and decode, any Base64 data that is issued to the client. Base64-encoded strings can often be easily recognized by their specific character set and the presence of padding characters at the end of the string.

Hex Encoding

Many applications use straightforward hexadecimal encoding when transmitting binary data, using ASCII characters to represent the hexadecimal block. For example, hex-encoding the username "daf" within a cookie would result in this:

```
646166
```

As with Base64, hex-encoded data is usually easy to spot. You should always attempt to decode any such data that the server sends to the client to understand its function.

Remoting and Serialization Frameworks

In recent years, various frameworks have evolved for creating user interfaces in which client-side code can remotely access various programmatic APIs implemented on the server side. This allows developers to partly abstract away from the distributed nature of web applications and write code in a manner that is closer to the paradigm of a conventional desktop application. These frameworks typically provide stub APIs for use on the client side. They also automatically handle both the remoting of these API calls to the relevant server-side functions and the serialization of any data that is passed to those functions.

Examples of these kinds of remoting and serialization frameworks include the following:

- Flex and AMF
- Silverlight and WCF
- Java serialized objects

We will discuss techniques for working with these frameworks, and the kinds of security issues that can arise, in Chapters 4 and 5.

Next Steps

So far, we have described the current state of web application (in)security, examined the core mechanisms by which web applications can defend themselves, and taken a brief look at the key technologies employed in today's applications. With this groundwork in place, we are now in a position to start looking at the actual practicalities of attacking web applications.

In any attack, your first task is to map the target application's content and functionality to establish how it functions, how it attempts to defend itself, and what technologies it uses. The next chapter examines this mapping process in detail and shows how you can use it to obtain a deep understanding of an application's attack surface. This knowledge will prove vital when it comes to finding and exploiting security flaws within your target.

Questions

Answers can be found at `http://mdsec.net/wahh`.

1. What is the `OPTIONS` method used for?
2. What are the `If-Modified-Since` and `If-None-Match` headers used for? Why might you be interested in these when attacking an application?
3. What is the significance of the `secure` flag when a server sets a cookie?
4. What is the difference between the common status codes 301 and 302?
5. How does a browser interoperate with a web proxy when SSL is being used?