CHAPTER 3

Web Application Technologies

Web applications employ a myriad of different technologies to implement their functionality. This chapter contains a short primer on the key technologies that you are likely to encounter when attacking web applications. We shall 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 quickly skim through this chapter to confirm that there is nothing new in here for you. If you are still learning how web applications work, you should read this primer before continuing to the later chapters on specific vulnerabilities. For further reading on any of the areas covered, we recommended *HTTP: The Definitive Guide* by David Gourley and Brian Totty (O'Reilly, 2002).

The HTTP Protocol

The 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, and 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 /books/search.asp?q=wahh HTTP/1.1
Accept: image/gif, image/xxbitmap, image/jpeg, image/pjpeg,
application/xshockwaveflash, application/vnd.msexcel,
application/vnd.mspowerpoint, application/msword, */*
Referer: http://wahh-app.com/books/default.asp
Accept-Language: en-gb,en-us;q=0.5
Accept-Encoding: gzip, deflate
User-Agent: Mozilla/4.0 (compatible; MSIE 7.0; Windows NT 5.1)
Host: wahh-app.com
Cookie: lang=en; JSESSIONID=0000tI8rk7joMx44S2Uu85nSWc_:vsnlc502
```

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 there is no further data following the blank line after the message headers.
- The requested URL. The URL 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, and in the example there is a single parameter with the name `q` and the value `wahh`.
- 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.

Some other points of interest in the example request are:

- 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 the Mozilla prefix is included by most browsers for historical reasons — this was the `User-Agent` string used by the originally dominant Netscape browser, and other browsers wished to assert to web sites 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 is used to specify the hostname that appeared in the full URL being accessed. This is necessary when multiple web sites are hosted on the same server, because the URL sent in the first line of the request does not normally contain a hostname. (See Chapter 16 for more information about virtually hosted web sites.)
- 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: Sat, 19 May 2007 13:49:37 GMT
Server: IBM_HTTP_SERVER/1.3.26.2  Apache/1.3.26 (Unix)
Set-Cookie: tracking=tI8rk7joMx44S2Uu85nSWc
Pragma: no-cache
Expires: Thu, 01 Jan 1970 00:00:00 GMT
Content-Type: text/html;charset=ISO-8859-1
Content-Language: en-US
Content-Length: 24246

<!DOCTYPE html PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN">

<html lang="en">
<head>
    <meta http-equiv="Content-Type" content="text/html;
charset=iso-8859-1">
...
```

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 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.

Some other points of interest in the previous response are:

- 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 is issuing the browser a further cookie; this will be submitted back in the `Cookie` header of subsequent requests to this server.
- The `Pragma` header is instructing the browser not to store the response in its cache, and the `Expires` header also indicates that the response content expired in the past and so 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, and 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`. There are some important differences between these methods which you need to be aware of, and which can affect an application's security if overlooked.

The `GET` method is designed for retrieval of 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 can be reused by themselves or

other users to 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 for performing 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 Back button of the browser to return to a page that was accessed using this method, the browser will not automatically reissue the request but will warn 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 result in an action being 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. The other methods you are most likely to require knowledge of are:

- **HEAD —** This 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 for checking whether a resource is present before making a `GET` request for it.
- **TRACE —** This method is designed for diagnostic purposes. The server should return in the response body the exact contents of the request message that it received. This can be used to detect the effect of any proxy servers between the client and server that may manipulate the

request. It can also sometimes be used as part of an attack against other application users (see Chapter 12).

- **OPTIONS —** This method asks the server to report the HTTP methods that are available for a particular resource. The server will typically return a response containing an `Allow` header that lists the available methods.
- **PUT —** This method attempts to upload the specified resource to the server, using the content contained in the body of the request. If this method is enabled, then you may be able to leverage it to attack the application; for example, by uploading an arbitrary script and executing this 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 17 for further details on these and examples of using them in an attack.

URLs

A uniform resource locator (URL) is a unique identifier for a web resource, via 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, and the port number is normally only included if it diverges from the default used by the relevant protocol. The URL used to generate the HTTP request shown earlier is:

```
http://wahh-app.comm/books/search.asp?q=wahh
```

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:

```
/books/search.asp?q=wahh
search.asp?q=wahh
```

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

NOTE **The correct technical term for a URL is actually *URI* (or uniform resource identifier), but this term is really only used in formal specifications and by those who wish to exhibit their pedantry.**

HTTP Headers

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

General Headers

- **Connection —** This is used to inform 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** — This is used to specify 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** — This is used to specify 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** — This is used to specify the type of content contained in the message body; for example, `text/html` for HTML documents.
- **Transfer-Encoding** — This is used to specify 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 —** This is used to tell the server what kinds of content the client is willing to accept, such as image types, office document formats, and so on.
- **Accept-Encoding** — This is used to tell the server what kinds of content encoding the client is willing to accept.
- **Authorization —** This is used to submit credentials to the server for one of the built-in HTTP authentication types.
- **Cookie —** This is used to submit cookies to the server which were previously issued by it.

- **Host —** This is used to specify the hostname that appeared in the full URL being requested.
- **If-Modified-Since** — This is used to specify the time at which 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** — This is used to specify 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.
- **Referer —** This is used to specify the URL from which the current request originated.
- **User-Agent** — This is used to provide information about the browser or other client software that generated the request.

Response Headers

- **Cache-Control** — This is used to pass caching directives to the browser (for example, `no-cache`).
- **ETag —** This is used to specify 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 —** This is used to instruct the browser how long the contents of the message body are valid for. The browser may use the cached copy of this resource until this time.
- **Location —** This is used in redirection responses (those with a status code starting with 3) to specify the target of the redirect.
- **Pragma —** This is used to pass caching directives to the browser (for example, `no-cache`).
- **Server —** This is used to provide information about the web server software being used.
- **Set-Cookie** — This is used to issue cookies to the browser that it will submit back to the server in subsequent requests.
- **WWW-Authenticate** — This is used in responses with a 401 status code to provide details of the type(s) of authentication supported by the server.

Cookies

Cookies are a key part of the HTTP protocol which most web applications rely upon, and which can frequently 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 back 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 already observed:

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

The user's browser will then automatically add 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 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, and are all 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 also include any of the following optional attributes, which can be used to control how the browser handles the cookie:

- **expires —** Used to set a date until which the cookie is valid. This will cause the browser to save the cookie to persistent storage, and it will be 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 —** Used to specify 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 —** Used to specify the URL path for which the cookie is valid.
- **secure** – If this attribute is set, then the cookie will only ever be submitted in HTTPS requests.
- **HttpOnly —** If this attribute is set, then the cookie cannot be directly accessed via client-side JavaScript, although not all browsers support this restriction.

Each of these cookie attributes can impact the security of the application, and the primary impact is on the ability of an attacker to directly target other users of the application. See Chapter 12 for further 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 first digit of the code:

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

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

- **100 Continue** — This response 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 will then return a second response when the request has been completed.
- **200 Ok —** This indicates that the request was successful and the response body contains the result of the request.
- **201 Created —** This is returned in response to a `PUT` request to indicate that the request was successful.
- **301 Moved Permanently —** This 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 —** This 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 —** This 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 —** This indicates that the client submitted an invalid HTTP request. You will probably encounter this when you have modified a request in certain invalid ways, for example by placing a space character into the URL.
- **401 Unauthorized —** The server requires HTTP authentication before the request will be granted. The `WWW-Authenticate` header contains details of the type(s) of authentication supported.
- **403 Forbidden —** This indicates that no one is allowed to access the requested resource, regardless of authentication.
- **404 Not Found —** This indicates that the requested resource does not exist.
- **405 Method Not Allowed —** This 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 so 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 —** Similar to the previous response, this indicates that the URL used in the request is too large for the server to handle.
- **500 Internal Server Error —** This 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 review the full contents of the server's response closely for any details indicating the nature of the error.
- **503 Service Unavailable —** This normally indicates that, although the web server itself is functioning and able to respond to requests, the application accessed via the server is not responding. You should verify whether this is the result of any action that you have performed.

HTTPS

The HTTP protocol uses plain TCP as its transport mechanism, which is unencrypted and so can be intercepted by an attacker who is suitably positioned on the network. HTTPS is essentially the same application-layer protocol as

HTTP, but this is tunneled over the secure transport mechanism, Secure Sockets Layer (SSL). This protects the privacy and integrity of all data passing over the network, considerably 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 now strictly been superseded by transport layer security (TLS), but the latter is still normally referred to using the older name.

HTTP Proxies

An HTTP proxy server 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 of its requests to that server, and 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.

There are two differences in the way HTTP works when a proxy server is being used, which you should be aware of:

- When a browser issues an HTTP request to a proxy server, it places the full URL into the request, including the protocol prefix `http://` and the hostname of the server. The proxy server extracts the hostname and uses this 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, as 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 onwards 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 web site 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:

- **Basic —** This is a very simple authentication mechanism that sends user credentials as a Base64-encoded string in a request header with each message.
- **NTLM —** This is a challenge-response mechanism and uses a version of the Windows NTLM protocol.
- **Digest —** This 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, although they are more commonly used within organizations to access intranet-based services.

COMMON MYTH **"Basic authentication is insecure."**

Basic authentication places credentials in unencrypted form within the HTTP request, and so 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 is in itself 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 different 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.