9

HTTP over SSL

9.1 Introduction

HTTP (*Hypertext Transfer Protocol*) provides a natural example of a protocol secured with SSL. It was the first protocol to use SSL and is still by far the most important secured protocol. The standard approach—found in nearly every Web browser and server—uses a separate ports strategy, whereas the IETF has standardized an upward negotiation technique for upgrading to TLS in HTTP.

This chapter starts with a discussion of the problem of Web security at a high level, including an introduction to the basic Web technologies. Then we discuss the traditional approach to HTTP with SSL (HTTPS, described in RFC 2818) and how it interacts with these Web technologies. This leads into a discussion of the newer HTTP Upgrade technique (described in RFC 2817), and how it compares to HTTPS. Finally, we discuss some programming issues commonly encountered when using HTTP with SSL.

9.2 Securing the Web

The prototypical Web security application is credit card submission to a Web server. The user browses some Web site and places a number of items in his virtual shopping cart. In general, this step is done without any security. It's assumed that this information isn't sensitive—although it might be if you were buying something embarrassing.

The requirement for security kicks in when the customer is ready to check out. In order to do so, the customer needs to provide his credit card number to the server. Because anyone who has the credit card number can pose as the user and initiate charges, it's obvious that we need to provide confidentiality for the credit card. Moreover, the user needs to be sure that he's submitting his credit card to the right server lest a fake server steal his credit card.

At checkout time, the server hands the user off to a secure page, where he can type in his credit card number and expiration date. After this information is submitted securely, the user needs to get a confirmation from the server that his order was received. This also needs to be securely delivered to prevent attackers from silently blocking orders.

For this application, our security requirements are very simple. We need to provide confidentiality for the data passing between client and server and the user needs to be sure that his client is connected to the right server. Other applications may have more advanced security needs, but any Web security protocol needs to at least satisfy these requirements.

Basic Technologies

Before we even discuss security, it's important to understand the basic technologies that make up the World Wide Web infrastructure. The three technologies that we need to consider are *HTTP*, *HTML*, and *URLs*.

HTTP

HTTP (*HyperText Transfer Protocol*) is the basic transport protocol of the Web. The Web is a client/server system and some mechanism needs to exist to move data between servers and clients. HTTP provides that mechanism. Most Web browsers speak other protocols, such as FTP, but the vast majority of Web traffic is moved by HTTP.

HTML

HTML *(HyperText Markup Language*) is the basic document format of the Web. HTML is basically an enhanced version of ASCII text. The two most important features it offers are the ability to structure the document to indicate paragraphs, line breaks, etc. and the ability to provide *links*. Links allow the user to click from one document to another.

URLs

URLs (*Uniform Resource Locators*) provide the references used in links. Every HTML link has an associated URL which tells the browser what to do when the user clicks on the link. In theory URLs can describe data fetched over any protocol but in practice they mostly refer to data to be fetched over HTTP.

Practical Considerations

There are several features of the real world Web environment that are not obvious from the simple client/server model that we've suggested so far. Three of these features turn out to have particular relevance when we consider the Web security problem: *connection behavior*, *proxies*, and *virtual hosts*.

Connection Behavior

Most Web pages contain a number of embedded ("inline") images on the page. These images must be fetched individually with their own HTTP request. As a performance improvement, most clients perform these fetches in parallel. Thus, fetching any page actually involves an extensive sequence of requests. To ensure reasonable performance, we need to ensure that per-request overhead is kept to a minimum.

Proxies

A very large number of Web transactions occur from large intranet environments. These intranets are often separated from the Internet by firewalls, and access to Internet resources is allowed only through proxies. A successful Web security solution must be able to pass through proxies.

Virtual Hosts

It's quite common for a single server (such as an ISP server) to host Web sites for a number of organizations. For instance, it's quite common for ISPs to offer combined Web hosting and credit card clearing for merchants too small to have their own merchant accounts. Naturally, each merchant would like to appear to have his own Web server even though multiple servers are in fact running on the same server as other merchants. A technique called *virtual hosts* makes this possible, but it can interact badly with security.

Security Considerations

Once we understand the sorts of transactions we'd like to perform and the Web protocols and environments in which they need to work, we can consider how to provide the appropriate security services. In particular, we need to figure out how we'll approach the important problems discussed in Chapter 7: *protocol selection*, *client authentication*, *reference integrity*, and *connection semantics*.

9.3 HTTP

This section provides a very brief overview of HTTP, described in [Fielding1999]. Our intent is not to provide a full description; that purpose is amply filled by a number of other sources, including [Stevens1994]. Rather, our intent is to describe enough of HTTP so that we can adequately talk about what it means to secure HTTP with SSL. Thus, we will focus on the details that are most relevant to security and SSL.

Conceptually, HTTP is a simple protocol. The basic unit of HTTP interaction is the request/response pair. The client opens a TCP connection to the server and writes the request. The server writes back the response. The server indicates the end of the response either with a length header or simply by closing the connection.

Requests

An HTTP request consists of three parts, the *request line*, *header*, and an optional *body*. The request line is simply a single line. The header is a series of colon-separated key-value pairs, and the body is arbitrary data. The header and the body are separated by a single blank line. Figure 9.1 shows a sample request.

```
GET / HTTP/1.0
Connection: Keep-Alive
User-Agent: Mozilla/4.7 [en] (X11; U; FreeBSD 3.4-STABLE i386)
Host: www.rtfm.com
Accept: image/gif, image/x-xbitmap, image/jpeg, image/pjpeg, image/png, */*
Accept-Encoding: gzip
Accept-Language: en
Accept-Charset: iso-8859-1,*,utf-8
(blank line)
```

Figure 9.1 An HTTP request

The request shown in Figure 9.1 consists simply of a request line and the header. The first line (starting with `GET`) is the request line. The rest of the request is header.

The format of an HTTP request line is

Method Request-URI HTTP-Version

The HTTP/1.1 RFC defines seven request methods: `OPTIONS`, `GET`, `HEAD`, `POST`, `PUT`, `DELETE`, `TRACE`. A number of other methods are defined by WEBDAV [Goland1999]. We don't need to be much concerned with the difference between various request methods. The two most common methods are `GET` and `POST`. The only relevant difference between them is that `POST`s may have a message body whereas `GET`s do not.

The Request-URI should be thought of as the name of the resource that we're trying to access. In general, it looks like a UNIX path name, for instance `/foo/bar/baz`. It may also have a series of arguments on the end. Finally, the HTTP-Version will generally be either `HTTP/1.0` or `HTTP/1.1`. HTTP/1.1 is the version that the IETF is standardizing on but we can mostly ignore the differences between the two versions.

A large number of header fields are possible but most of these don't have any interaction with SSL. The request in Figure 9.1 does have one header line that is relevant, however. The `Connection` header indicates that the client would like the server to keep the connection open after sending the response. Thus, keep-alive interacts with the SSL closure process. We'll discuss that interaction further in Section 9.15.

All the information that the client transmits to the server is in the client request. Thus, if we are to secure that data, we must ensure that the client's request is encrypted. Note that this necessarily includes securing the request line because the identity of the

resource that the client is fetching may itself be sensitive. Obviously, providing this service requires that the client know it is talking to the correct server.

Responses

HTTP responses have a very similar format to requests. The only difference is that the request line is replaced with a status line which indicates how the server processed the request. The format of a status line is

HTTP-Version Status-Code Reason-Phrase

The HTTP-Version is the same as with the request. The Status-Code is a numeric code indicating the action the server took. In general this will be 200 for a successful request or some other number for failure (300 series codes are used to indicate some other non-error conditions in which the request isn't serviced.) We'll see a few more status codes when we discuss Upgrade in Section 9.21. The Reason-Phrase is simply a textual description of what happened. For a successful transaction this is usually `OK`. Figure 9.2 shows a successful HTTP response—the response to the request in Figure 9.1.

The response shown in Figure 9.2 illustrates a number of important points. First note the `HTTP/1.1 200 OK` status line, indicating that the request succeeded. Also, note the following three header fields:

```
Content-Length: 1650
Keep-Alive: timeout=15, max=100
Connection: Keep-Alive
```

The `Keep-Alive` and `Connection: Keep-Alive` header fields indicate that the server will not be closing the connection after sending the response. This means that the client will have to determine when the response ends from the contents of the response. The `Content-Length` header tells the clients how many bytes are in the body of the response. The `Content-Type` line tells us that the body of the message is of type `text/html`, indicating that the body is an HTML document.

Obviously, if securing HTTP transactions is to have any meaning at all, we must provide security for the response. This not only means ensuring that it is safe from viewing by attackers but also that no attacker can pose as the server in order to send data to the client or tamper with that data in transit.

9.4 HTML

HTML (*HyperText Markup Language*) is simply ASCII text decorated with a series of markers (called *tags*) that add structure to the document. For instance the `<P>` tag says to begin a new paragraph. A Web browser contains an *HTML parser* which takes the HTML as input and produces a formatted page as output. Note that because the markup

```
HTTP/1.1 200 OK
Date: Sat, 15 Jan 2000 05:15:54 GMT
Server: Apache/1.3.1 (UNIX)
Last-Modified: Tue, 22 Jun 1999 19:25:14 GMT
ETag: "2a99d-672-376fe31a"
Accept-Ranges: bytes
Content-Length: 1650
Keep-Alive: timeout=15, max=100
Connection: Keep-Alive
Content-Type: text/html

<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 3.2 Final//EN">
<HTML>
 <HEAD>
  <TITLE>RTFM</TITLE>
 </HEAD>
<!-- Background white, links blue (unvisited), navy (visited), red (active) -->
 <BODY
  BGCOLOR="#FFFFFF"
  TEXT="#000000"
  LINK="#0000FF"
  VLINK="#000080"
  ALINK="#FF0000"
 >
<CENTER>
<A HREF="contact.html">
<IMG SRC="rtfm.gif" BORDER=0></A>
</CENTER>
Deleted text
RTFM in cooperation with Claymore Systems is releasing a free
Java SSLv3/TLS implementation: <A HREF="/puretls>PureTLS"</A>.
<P>
 
<CENTER>

<A HREF="contact.html">Contact Us</A>
</BODY>
</HTML>
```

Figure 9.2 An HTTP Response

mostly describes structure and not layout, there is more than one way to format a given page.

The only aspect of HTML that we need to be concerned with is the links that it contains. A link is merely a reference in the page to another piece of content. That content can in turn be fetched by the Web browser, usually using HTTP but sometimes via some other protocol.

Anchors

The most familiar type of link is what's referred to as an *anchor*. This is simply a section of the HTML (typically a region of text) that is tagged in such a way that it corresponds to a given reference. Clicking on the section of screen that corresponds to that section of HTML causes the browser to fetch the indicated content. When you're clicking the various highlighted and underlined links in your Web browser, you're using anchors. The HTML page in Figure 9.2 contains an anchor:

```
<A HREF="/puretls>PureTLS"</A>
```

What appears on the screen is the underlined text PureTLS. Clicking on the underlined section causes the browser to dereference the link by fetching the document pointed to by `/puretls` (on the Web server from which it fetched the original document: www.rtfm.com). The `HREF="/puretls"` is what specifies the target of the link. The target of the link (in this case the string `/puretls`) is called a *Uniform Resource Locator* (URL). We'll discuss URL syntax in Section 9.5. The link itself is defined by the region bracketed by `<A HREF="/puretls">` (the *start tag*) and `</A>` (the *end tag*).

Inline Images

Another type of link is what's called an inline image. Unlike an anchor, which just sits there until the user clicks on it, the browser automatically fetches inline images and puts them into the Web page at the location in the HTML where the image tag appears. (Because the layout of the HTML page is determined in part by the browser, the whole idea of location is a rather fuzzy one.)

Whenever you see a picture in some Web page, that's an inline image of some sort. Even animated pictures are done with a variant of inline images. The HTML page in Figure 9.2 contains a reference to an inline image:

```
<IMG SRC="rtfm.gif" BORDER=0>
```

The `IMG` tag is basically like the `A` tag except that it uses the field `SRC` rather than `HREF` to carry the URL pointing to the content it's referencing. Thus, the picture contained in the file `rtfm.gif` (RTFM's logo) will be displayed in this page. This tag also contains a `BORDER=0` attribute, indicating that the image should be presented without any border.

Note that the security properties of an inline image may not be the same as the security properties of the page that references it. Because images are often used as integral parts of the meaning of the page, the browser should clearly indicate if the security properties are different from that of the page in which they are embedded.

Forms

The final sort of link that we'll consider is what's generally referred to as a Web form. Although the implementation of a Web form is complicated, the idea is very simple and familiar. The Web page shows a number of user interface widgets such as text fields and pulldown menus. There's also a special button called the *submit button*. Sometimes it's even labeled "submit."

The user interacts with the widgets in some way, say by selecting which kind of credit card he has and filling in the credit card number and expiration of the card that he wants to pay with. When he's done he clicks the submit button. At this point some magic happens: the browser takes the current values of all the widgets and builds a string that carries their values. Then it makes a request to the server (to the URL associated with the form) and passes the values of the widgets with the request.

Forms (and indeed all links) can be labeled with a method type which represents the HTTP method type to use when dereferencing the link. The relevant two methods for forms are `POST` and `GET`. The important difference between them is simple: In a `GET` the values of the fields in the form are attached to the URI in the request line; in a `POST` it's sent in the message body.

Dynamic Content

Web pages can also contain a variety of types of dynamic content: code that is executed by the Web client. This code can either be directly in the HTML or can be referenced via a link similar to an inline image. This code can be in a variety of languages, including Java, JavaScript and VBScript. It is also possible to have a link that references a binary program which is then loaded into the memory space of the Web browser. This is called a *plug-in*.

9.5 URLs

URLs are the basic form of addressing in the World Wide Web. The idea behind a URL is to provide a single short string that identifies any network-accessible resource. URLs provide a unified interface for a wide variety of different access methods. A URL might reference a document accessible via HTTP, FTP, Gopher, or even instruct the browser to create a mail message. This saves the user from having to be familiar with the different conventions previously required to use any particular access method (for instance, anonymous FTP).

URLs can be extremely complicated, and they vary substantially between different access methods. (See [Berners-Lee1998] for a full description.) However, all the URLs that we'll be concerned with have a common form:

<scheme>://<host>[:<port>]/<path>[?<query>]

The scheme corresponds to the protocol being used to access the resource. Thus, for HTTP, it would be `http`, for FTP `ftp`, and so on. The host and port fields specify the server to connect to. The [ ] around the port field means that it's optional. Protocols usually have a default port defined, but access may be permitted on any port. The default port for HTTP is 80.

The path section of the URL provides the name (location) of the resource on the indicated server. Paths typically look like UNIX filenames (`/foo/bar/baz`). In theory, the fact that the path looks like a filename does not mean that the various parts correspond to directories, files, etc., but in practice this is often the case. In such cases, the server will serve the file that corresponds to the resource.

Finally, a URL might have a query at the end, set off by a ?. The query is intended to be resource specific and is "interpreted by the resource." Queries are most frequently used when the resource isn't a file but is rather a program that dynamically generates the resource in question. In such cases, the query part is used to provide the input to the program. For instance, an `ldap:` URL corresponds to an entry in an LDAP (*Lightweight Directory Access Protocol*) directory server. The query information can be used to identify the attributes that the client is interested in, the scope of the query, as well as other query parameters.

An Example

A simple URL might look something like Figure 9.3.

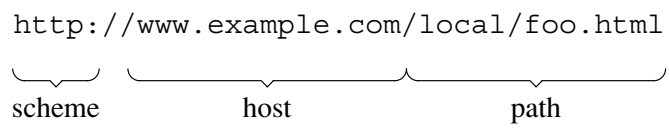

Figure 9.3 A simple URL

The resource described by this URL might be described as follows: Using HTTP, connect to the machine `www.example.com` on port 80. Request the resource `/local/foo.html`.

It's also possible to have a *relative* URL, such as `/local/foo.html`. In this case, the scheme and the host are assumed to be the same as those used to fetch the document containing the URL.

URIs versus URLs

The HTTP request line contains a URI. We've just seen what URLs are, but what's a URI? URI stands for *Uniform Resource Identifier*. URIs are a superset of URLs. A URI is simply a short string that refers to a given resource. A URL is a URI that contains explicit instructions about how to fetch the resource. It's possible to have URIs that uniquely identify a resource but don't provide instructions on how to fetch it. One such class of URIs is called a *Uniform Resource Name* (URN).

In general, any URI you're likely to see is a URL, because they describe how to use HTTP, FTP, etc., to fetch the resource. However, it's conceivable that the HTTP request line could contain a non-URL URI and therefore RFC 2616 uses the term URI. For the rest of this chapter we'll use the term URI when talking about the token that appears in the HTTP request and URL when talking about the token appearing in a link. See [W3C2000] for more information on this topic.

9.6 HTTP Connection Behavior

Consider a typical Web page. Such a page likely has between five and ten inline images. As we discussed in the previous section, each of these images requires a separate HTTP request/response pair. Because most older Web browsers and servers closed the connection to indicate the end of the response, it could potentially require as many as 11 separate TCP connections to fetch a single page.

Really old browsers such as Mosaic make all of these TCP connections sequentially, but the performance consequences of this procedure were terrible and Netscape soon introduced the technique of opening a number of simultaneous HTTP connections, one for each image up to a configurable upper limit.

In 1994 Spero [Spero1994] showed that the performance of using multiple connections (whether sequential or parallel) suffers badly from its interaction with TCP. First, each connection requires a three-way handshake to initiate (costing 1.5 round trips). Worse yet, TCP slow start (described in [Jacobsen1988]) can cause serious delays when the client's requests are longer than the server's maximum segment size.

In order to work around these problems, [Padmanabhan1995] and [Mogul1995] suggested keeping the TCP connection open after serving pages. This approach eventually emerged as the `Connection: Keep-Alive` header we saw in Section 9.1. As we saw, the end of data in that case is signalled with the `Content-Length` header. Although persistent connections reduce the number of connections made by browsers, many browsers still initiate a number of connections when they load a page with a large number of images to allow them to load all the images in parallel.

The use of a large number of parallel connections was one of the prime motivators for the session resumption feature of SSL. Because even simple pages cause a large number of connections within seconds of each other, SSL performance can be dramatically improved with session resumption.

9.7 Proxies

An HTTP proxy is a program that sits between a client and a server. It accepts the client request and then initiates its own request to the server. There are two primary reasons why proxies are useful: caching for multiple clients and as a pass-through for firewalls. The primary issue we face is to ensure that HTTP with SSL can operate even in proxied environments. This section provides some background on how proxies work.

The protocol traffic between a browser and a proxy is almost identical to the protocol traffic between a browser and a server. It is not, however, completely identical. The primary difference that we need to be concerned with is the difference in the request line. The problem is that the standard request line does not contain the identity of the server, which contains the resource but only the address of the resource on the server. Thus, when talking to a proxy, the client uses a fully qualified URL, containing the hostname as well as the path of the resource. For the request shown in Figure 9.1, the request line would be:

```
GET http://www.rtfm.com/ HTTP/1.0
```

When talking to the server, the proxy would remove the hostname and send the request as the client would have sent it to the server. There are a number of other subtleties in the proxy interaction, mainly having to do with the meaning of various header fields. Some header fields are intended for the proxy and the proxy should process (and remove) them when sending the request to the server. Others are intended for the server and the proxy should ignore them. We'll see an example of this distinction when we address Upgrade in Section 9.21.

Caching Proxies

The idea behind a caching proxy is simple. Many Web pages are both static and frequently fetched. If those pages are cached on the client machine, then server load and performance can be greatly improved. Most browsers now cache files on disk or in memory but this is useful only when a client repeatedly fetches the same data. Even so, this optimization is surprisingly useful: consider the case in which a single image such as a menu bar or a logo is used repeatedly throughout a site.

In an environment where many users are likely to fetch the same page, a more aggressive strategy can pay off. If caching is done at the proxy, then all users can benefit once a single user has fetched a page. Naturally, browsers must be configured to make connections to the cache rather than directly to the server. This configuration can be done manually but many browsers also provide auto-configuration features which allow IS administrators to configure proxies for all internal clients.

The primary problem that complicates caching proxy design and implementation is maintaining cache freshness. It is necessary to detect content that is dynamic and therefore cannot be cached as well as to periodically check whether a document has been updated on the server. HTTP contains complicated mechanisms for addressing this problem but they are largely irrelevant to SSL because the traffic from client to server is encrypted and therefore caching is impossible anyway.

Firewall Proxies

The purpose of a firewall is to restrict traffic between the inner (protected) network and the Internet as a whole. Although some firewalls allow arbitrary TCP connections between the inner network and the Internet, many do not. Such firewalls typically rely on application-layer proxies to enable the necessary (typically restricted) set of services.

The primary design goal for firewall proxies is of course security. To that end, the proxies are typically very simple and designed to be error free. Although a firewall HTTP proxy can also be a caching proxy, usually such proxies do not contain a cache. The primary issue for SSL/proxy interaction is to ensure that our HTTP over SSL traffic can pass through proxies safely without being damaged by the proxy.

9.8 Virtual Hosts

Consider the situation where a customer's Web site is being hosted by an *Internet Service Provider* (ISP). Even though an ISP may have many customers, it wants to have only a few server machines. This necessitates the ability to appear to have many Web servers on the same server machine.

Naturally, each customer wants to be able to have a separate Web address. Even if they are actually hosted on the server `www.provider.com`, they would like their clients to be able to access their home page at `http://www.customer.com/`. Because there is only one Web server, it will be receiving requests for multiple *virtual servers*. The problem now becomes how to distinguish which virtual server a given request is intended for. The Request-URI specifies only the path of the resource within the server, so other means are necessary.

Modern Web clients use the `Host` header to provide the hostname of the Web server that they believe they are connecting to. This permits the Web server to disambiguate multiple virtual servers on the same physical server. Obviously, a good design for HTTP over SSL should ensure that it doesn't break virtual hosts.

9.9 Protocol Selection

At the time that the first HTTP over SSL solutions were being designed, the upward negotiation strategy for SSL hadn't been invented. Also, an important design consideration for HTTP servers was that they were supposed to be *stateless*—each request/response pair was intended to be independent. Thus, a protocol model that required multiple round trips through the HTTP engine (as we'll see is required by Upgrade) was difficult to accept. The idea behind SSL, after all, was that you could simply replace your sockets calls with SSL calls and not change the rest of your code at all. If minimizing impact on the rest of the code is the overriding design goal (which it often is) a separate ports approach is the only workable strategy.

9.10 Client Authentication

Certificate-based client authentication is completely unnecessary for secure forms submission. In practice, most Web sites choose not to authenticate their users at all. The ones that do typically wish to use some external authenticator (such as a credit card number). As a consequence, certificate-based client authentication is not a high priority for Web applications.

Although client authentication in SSLv2 wasn't widely supported, most SSLv3 and TLS implementations do have support for client authentication. In certain restricted (intranet) environments, certificate-based authentication is very useful and practical. Thus, a Web security solution should also support certificates for clients.

It's possible to support certificates without any special protocol support from HTTP (as opposed to from SSL) because the server can simply request client authentication from the client in the SSL handshake. However, because the SSL handshake happens before the server knows which resource the client is attempting to access, this tends to be a rather all-or-nothing proposition. As we'll see in Section 9.18, it might be desirable for servers to be able to signal in the reference that they expect client authentication, thus allowing the client to offer it in the handshake.

9.11 Reference Integrity

The Web has a very clear reference model: resources are identified by URL. Thus, the obvious design choice for HTTP over SSL was simply to try to match the URL reference to the server's identity. This required that CAs issue certificates containing hostnames. This is the approach that the HTTPS designers chose.

A more sophisticated approach would be to have the references themselves contain an additional indicator of the expected server identity. This could be placed in the anchor (as was done in Secure HTTP [Rescorla1999a]; see Chapter 11) or it could be somewhere in the URL. The primary disadvantage of this approach is that it would make the common user activity of typing in URLs much harder. The primary advantage is that it would allow vastly more flexibility in terms of what sort of certificates servers could have. We'll see a case where that sort of flexibility would have been valuable when we discuss virtual servers in Section 9.17.

Connection Semantics

HTTP over SSL is essentially constrained to use HTTP's connection semantics. This was obviously problematic because SSL handshakes were far more expensive than TCP handshakes. Reducing this load was one of the primary motivators for the fast resumption mechanism. Unfortunately, resumption broke the stateless model of HTTP. In fact, as we'll see in Section 9.24, it placed quite a burden on the server because implementing resumption required interprocess communication of session state.

If we think of SSL as being designed without reference to HTTP and then applied to HTTP, this burden seems like an unfortunate outcome of the interaction of a general security protocol with HTTP. However, in reality SSL was largely designed for HTTP and modifications to reduce the state burden would have been possible, had the designers thought of them. We'll discuss a few such tricks when we discuss Secure HTTP in Chapter 11. Nevertheless, the stateless interaction model was already on its way out when SSL was designed, due to the TCP performance issues discussed in Section 9.6 and its unacceptable performance consequences on UNIX systems.

9.12 HTTPS

Now that we've covered the problem of Web security from a design perspective, we're prepared to understand HTTPS, the dominant approach used for HTTP with SSL. We're also prepared to evaluate how good a job it actually does at meeting the challenges we've discussed.

We begin by examining a simple request made with HTTPS to illustrate its basic features. Although the basic idea behind HTTPS is very simple, there are some technical points that need to be carefully defined: *connection closure* behavior and the use of URLs to provide *reference integrity*. With a clear definition of HTTPS in hand, we then consider its interaction with a number of confounding features faced in the real network: *proxies* and *virtual hosts*. We also discuss the problem of determining when to require certificate-based *client authentication* in a real Web environment.

Although HTTPS is basically secure, a number of attacks and limitations have been discovered. We describe these attacks and some techniques for avoiding them in Sections 9.19 and 9.20. Finally, we consider HTTP Upgrade, a newer alternative to HTTPS. HTTP Upgrade is intended to fix some known problems with HTTPS, but we'll see that it introduces a number of problems of its own.

9.13 HTTPS Overview

HTTP was the first application-layer protocol to be secured with SSL. The first public implementation of HTTP over SSL was in Netscape Navigator 2 in 1995. At the time, the separate ports strategy was the only one available and Netscape used it for HTTP as well as for NNTP and SMTP. The URLs for pages fetched via HTTP over SSL began with `https://` to disambiguate them from HTTP URLs, and this approach quickly became known as HTTPS.

> HTTPS stands for "HTTP Secure." This construction seems a little stilted but Netscape was prevented from using `shttp://` because Secure HTTP (see Chapter 11) already used that protocol name for its URL scheme.

Despite the wide deployment of SSL-enabled HTTP implementations, neither Netscape nor anyone else published a standard for HTTP over SSL until the IETF

started considering the matter in 1998. Instead, it was widely assumed that there was only one obvious way to do things and the (somewhat nontrivial) details were passed around as lore. HTTPS was finally documented in RFC 2818 [Rescorla2000].

The HTTPS approach is very simple: The client makes a connection to the server, negotiates an SSL connection, and then transmits its HTTP data over the SSL application data channel. This description makes it sound very easy and in principle this is true. The only other piece of information that you need to have in order to create an interoperable (though not necessarily secure) HTTPS implementation is the port number. Because we're using a separate ports strategy, HTTPS connections need their own port. IANA assigned HTTPS port 443, so by default HTTPS connections happen on that port. Of course, it's still possible to specify a different port number in the URL.

Figure 9.4 shows an example HTTPS request in action between a Netscape 4.7 client and an Apache 1.3.9 server running mod_ssl 2.4.10. All the initial traffic (records 1–9) is the SSL handshake. This is a rather ordinary SSLv3 handshake except that the client is connecting using the SSLv2 backward-compatibility handshake. Even with modern Web browsers, this is still by far the most common case. Also, note that the client offered only SSLv3 not TLS. As of this writing, TLS support is still not available in Netscape.

```
New TCP connection: romeo(4577) <-> romeo(443)
1 948676151.6444 (0.0005) C>S SSLv2 compatible client hello
  Version 3.0
  cipher suites
      TLS_RSA_WITH_RC4_128_MD5
      value unknown: 0xffe0   Proprietary Netscape cipher suite
      TLS_RSA_WITH_3DES_EDE_CBC_SHA
      value unknown: 0xffe1   Proprietary Netscape cipher suite
      TLS_RSA_WITH_DES_CBC_SHA
      TLS_RSA_EXPORT1024_WITH_RC4_56_SHA
      TLS_RSA_EXPORT1024_WITH_DES_CBC_SHA
      TLS_RSA_EXPORT_WITH_RC4_40_MD5
      TLS_RSA_EXPORT_WITH_RC2_CBC_40_MD5
2 948676151.6495 (0.0051) S>C   Handshake
      ServerHello
        session_id[32]=
          15 07 d3 46 a9 40 bc bc 6f 54 f9 60
          40 d0 bf 2f 08 3e 1e 4e f4 1d 7c 52
          31 46 14 20 ad 95 5b 04
        cipherSuite           TLS_RSA_WITH_RC4_128_MD5
        compressionMethod           NULL
3 948676151.6495 (0.0000) S>C   Handshake
      Certificate
4 948676151.6495 (0.0000) S>C   Handshake
      ServerHelloDone
```

(continued)

Figure 9.4 *(continued)*

```
5  948676151.6637 (0.0141) C>S   Handshake
       ClientKeyExchange
6  948676151.6900 (0.0262) C>S   ChangeCipherSpec
7  948676151.6900 (0.0000) C>S   Handshake
       Finished
8  948676151.6921 (0.0020) S>C   ChangeCipherSpec
9  948676151.6921 (0.0000) S>C   Handshake
       Finished
10 948676151.6933 (0.0012) C>S   application_data
    data: 284 bytes
    ---------------------------------
    GET /tmp.html HTTP/1.0
    Connection: Keep-Alive
    User-Agent: Mozilla/4.7 [en] (X11; U; FreeBSD 3.4-STABLE i386)
    Host: romeo
    Accept: image/gif, image/x-xbitmap, image/jpeg, image/pjpeg, ↵
    image/png, */*
    Accept-Encoding: gzip
    Accept-Language: en
    Accept-Charset: iso-8859-1,*,utf-8

    ---------------------------------
11 948676151.7013 (0.0079) S>C   application_data
    data: 395 bytes
    ---------------------------------
    HTTP/1.1 200 OK
    Date: Mon, 24 Jan 2000 01:09:11 GMT
    Server: Apache/1.3.9 (Unix) mod_ssl/2.4.10 OpenSSL/0.9.4
    Last-Modified: Sun, 23 Jan 2000 23:08:11 GMT
    ETag: "58820-79-388b89db"
    Accept-Ranges: bytes
    Content-Length: 121
    Connection: close
    Content-Type: text/html

    <HTML>
     <HEAD>
      <TITLE>Test</TITLE>
    </HEAD>
    <BODY>
    <H1>
    Test Page
    </H1>
```

```
    This page is just a sample.
    <P>
    </BODY>
    </HTML>
    ------------------------------------
12 948676151.7063 (0.0050) S>C   Alert
        level       warning
        value       close_notify
Server FIN
13 948676151.8052 (0.0989) C>S   Alert
        level       warning
        value       close_notify
Client FIN
```

Figure 9.4 A Request with HTTPS

Record 10 contains the HTTP request. This request is essentially identical to the request we showed in Figure 9.1 except that it's for a different URL. Note that absolutely no client data is sent until the SSL connection has been negotiated. Shortly, we'll see how this can cause problems in transitioning HTTP systems to HTTPS.

Record 11 contains the HTTP response. Note that the HTTP response fits entirely in one record. Of course, if the Web page were longer, this might require spanning multiple records. This response is basically similar to the response we showed in Figure 9.2, except that the page itself is different.

The only significant difference is that instead of `Connection: keep-alive`, the server has sent a `Connection: close` header, indicating that it will close the connection after serving this page. We've configured the server not to use retained connections so that we can show the closure behavior.

As we'd expect, the server sends its close_notify followed by a TCP FIN. The client responds with its close_notify and a FIN. Technically, the server could withhold its FIN until after it had received the client's close_notify. The SSL spec is silent on this issue. However, in practice, early versions of Netscape would not respond to the close_notify until they had received the FIN, so sending it immediately is good practice.

Also in this chapter:

- HTTPS Reference Integrity
- Proxies
- Virtual hosts
- Client Authentication
- Coverage of the new HTTP Upgrade standard