```
43                  case SSL_ERROR_ZERO_RETURN:
44                    /* End of data */
45                    if(!shutdown_wait)
46                      SSL_shutdown(ssl);
47                    goto end;
48                    break;
49                  case SSL_ERROR_WANT_READ:
50                    break;
51                  default:
52                    berr_exit("SSL read problem");
53                }
54              } while (SSL_pending(ssl));
55            }

56            /* Check for input on the console*/
57            if(FD_ISSET(fileno(stdin),&readfds)){
58              c2sl=read(fileno(stdin),c2s,BUFSIZZ);
59              if(c2sl==0){
60                shutdown_wait=1;
61                if(SSL_shutdown(ssl))
62                  return;
63              }
64              c2s_offset=0;
65            }

66            /* If we've got data to write then try to write it*/
67            if(c2sl && FD_ISSET(sock,&writefds)){
68              r=SSL_write(ssl,c2s+c2s_offset,c2sl);

69              switch(SSL_get_error(ssl,r)){
70                /* We wrote something*/
71                case SSL_ERROR_NONE:
72                  c2sl-=r;
73                  c2s_offset+=r;
74                  break;

75                /* We would have blocked */
76                case SSL_ERROR_WANT_WRITE:
77                  break;

78                /* Some other error */
79                default:
80                  berr_exit("SSL write problem");
81              }
82            }
83
```

(continued)

Figure 8.23 *(continued)*

```
84          }
85        end:
86          SSL_free(sal);
87          close(sock);
88          return;
89        }
90
```

read_write.c

Figure 8.23 `read_write()` function

Set nonblocking

17–21 The first thing we do is make the socket nonblocking. This is standard UNIX code.

Set up `select()` ***masks***

24–34 We always want to be ready to read from the server, but we want to do flow control for data we write to the server. To that end, we always look for the server socket being ready to read, but we switch between looking for data on the keyboard and looking for being ready to write to the server. If `c2sl` is nonzero, then we have data to write to the server and so we tell `select()` to look for the server socket to be writeable. Otherwise, we wait for the keyboard to be readable.

Note that if we wanted to we could always wait for data to be available on the keyboard. However, this might mean that data came in on the keyboard while we still had data to write to the server. Then we would have to append this new data to the `c2s` buffer and would thus complicate the code. We definitely want to look for the server socket to be writeable only when we have data to write. Most of the time there is no data to write or read and so the server socket is writeable most of the time. Thus, if we listened for the server socket to be writeable we'd constantly be spinning through the `select()` loop, which is not the behavior we want. Rather, we want our program to sleep until it has something to do.

We've seen the rest of the function before, but one inefficiency is worth noting: once we've read data from the keyboard we have to reenter the `select()` before we even attempt to write it to the server. In most cases the server is ready to write, so it would be slightly faster to simply attempt the write immediately. However, this would complicate the code and be only a trivial optimization.

8.10 Closure

Now that we've seen the code for reading and writing data, we're ready to examine closure. We need to be concerned with two primary issues. First, we need to ensure that the program both sends and receives a close_notify—and that an error is thrown if a close_notify is not received. Second, we need to ensure that we handle an incomplete close correctly. First, let's examine the OpenSSL-based closure code in Figure 8.23.

OpenSSL Closure

We've already seen the closure behavior on the server, so now let's examine it on the client. There are two situations in which the client might need to shut down the connection. First, the client might receive an end of data via the keyboard. Second, the client might receive a close_notify from the server. (If the client receives a TCP FIN without a close_notify it just exits with an error at the end of `SSL_read()`.) The only way that our server can terminate is if it receives a user end of data.

User end of data

59–63 If we receive an end of data from the keyboard, then we need to initiate the closure process. OpenSSL only has one function for doing this, `SSL_shutdown()`. It returns 1 if it was able to complete the closure process and 0 otherwise. The only condition in which it's likely to return 0 is if it hasn't received a close_notify yet. Thus, if we get a 0 return value, we have to wait for one. We set `shutdown_wait` to 1 and return to the select loop.

Wait for close

43–47 This section of code handles both the case where we receive an unsolicited close_notify from the server and the case where we receive one in delayed response to our close_notify. If the server socket is readable but we get a 0 return value we know that we've received a close_notify (because a bare TCP FIN would generate an error). If this is an unsolicited close_notify, then `shutdown_wait` will be zero and we need to use `SSL_shutdown()` to send our close_notify. On the other hand, if we have already sent our close_notify, then we simply jump out of the loop.

In the case of an unsolicited close_notify it's possible that the other side has completely closed the connection. In this case, our close_notify might trigger a TCP RST, which manifests itself on the sending end by raising a `SIGPIPE` signal. Because the default response to a `SIGPIPE` is to shut down the process, we have to arrange to ignore it. The call to `signal()` in Line 52 in `common.c` (Figure 8.4) takes care of this by setting up a dummy handler. Another approach would be to set the signal to be ignored. This approach allows us to catch the signal and potentially do something with it—like close the connection.

Clean up objects

84–88 Cleaning up consists of freeing the `SSL` object and closing the socket afterwards.

PureTLS closure

Unlike OpenSSL, PureTLS provides three methods for handling the SSL closure process. `close()` acts much like `SSL_shutdown()`, but `sendClose()` and `recvClose()` allow a finer control over what's going on. The sample code uses these rather than `close()`.

Recall that we omitted the implementation of `ReadWrite.onEOD()` from our discussion in Section 8.8. We present it here in Figure 8.24.

ReadWrite

```
56          protected void onEOD(){
57            try {
58              s.sendClose();
59            } catch (IOException e){
60              ; // Ignore broken pipe
61            }
62          }
```

ReadWrite

Figure 8.24 `ReadWrite.onEOD()`

`onEOD()` is called at the end of `ReadWrite.run()`. Typically, this occurs when the thread has received an end of data from the keyboard. In that case, the thread sends a close_notify using `sendClose()` and exits. However, `ReadWrite.onEOD()` can also be called when an unsolicited close_notify is received from the server. This process is initiated by the class `ReadWriteWithCancel`, as shown in Figure 8.25.

ReadWriteWithCancel.java

```
11      public class ReadWriteWithCancel extends ReadWrite {
12          protected ReadWrite cancel;

13          public ReadWriteWithCancel(SSLSocket s,InputStream in,OutputStream out,
14            ReadWrite cancel){
15            super(s,in,out);
16            this.cancel=cancel;
17          }

18          protected void onEOD(){
19            if(cancel.isAlive()){
20              cancel.interrupt();
21              try {
22                cancel.join();
23              } catch (InterruptedException e){
24                throw new InternalError(e.toString());
25              }
26            }
27          }
```

ReadWriteWithCancel.java

Figure 8.25 `ReadWriteWithCancel` class

`ReadWriteWithCancel` differs from `ReadWrite` in that it overrides `onEOD()`. Recall from Section 8.8 that the thread listening to the server is an instance of `ReadWriteWithCancel`. When that thread sees an end of data from the server it knows that it's received a close_notify. (Bare TCP FINs cause an `IOException` to be thrown.) It responds to this by triggering the thread writing to the server to send its own close_notify. This ensures that we don't have two threads trying to write to the same socket at the same time, which could happen if `ReadWriteWithCancel` tried to call `sendClose()` itself.

The signalling is done by calling `cancel.interrupt()`. `ReadWrite` tests for the interrupt using the `interrupted()` call. If `interrupted()` returns `true` then the read/write loop exits and `onEOD()` is called. Meanwhile, the server-to-client thread waits for the client-to-server thread to exit and then exits itself.

Once again, if the other side sent an unsolicited close_notify we need to be concerned with getting an RST when we send our close_notify. This is why the call to `sendClose()` is wrapped in a `try/catch`. The RST triggers an `IOException` which is simply ignored.

8.11 Session Resumption

The last topic we'll treat in this chapter is session resumption. As we discussed in Section 4.10, when the client has the server's domain name, it's possible to do automatic session resumption. PureTLS does this. Figure 8.26 shows a PureTLS example class that does session resumption on the client side. The PureTLS server we've been using all along automatically loops and so needs no special support to demonstrate session resumption.

Java Session Resumption

RClient.java

```
6   public class RClient extends Client {
7       public static void main(String []args)
8         throws IOException {
9         SSLContext ctx=createSSLContext(keyfile,password);
10        /* Connect and close*/
11        SSLSocket s=connect(ctx,host,port);
12        s.close();
13        /* Now reconnect: resumption happens automatically*/
14        s=connect(ctx,host,port);
15        s.close();
16      }
17  }
```

RClient.java

Figure 8.26 Resuming sessions with PureTLS

Note that we haven't done anything special to enable session resumption. The `SClient` class that we were using simply didn't do it. This class does.

C Session Resumption

OpenSSL requires some additional support in order to do session resumption. We've seen the initialization code required on the server previously and we don't need to add any new code there. However, we do need to do some additional work on the client side. Figure 8.27 shows a client that does session resumption. The key code here is at line 27 and line 34. We use `SSL_get_session()` to get the session data from the first `SSL` object and then use `SSL_set_session()` to assign the data to the second `SSL` object. From there, we can simply call `SSL_connect()` again.

Of course, this code isn't suitable for a production application. You can only resume sessions with the same server that you created them with, and this code makes no attempt to discriminate between various servers—because the server name is hardwired. In a real application, you would want to have some sort of lookup table that maps hostname/port pairs to `SESSION` objects.

Finally, note that even without having the hostname and port passed into the SSL toolkit, it is still possible to automatically resume sessions. The toolkit can call `getpeername()` on the socket in order to get the remote host and port. However, because OpenSSL's `BIO` abstraction separates the SSL code from the socket, this isn't possible with OpenSSL. TLSGold fixes this problem by having a `getpeername()` function as part of its I/O abstraction. However, OpenSSL does not do this and so the application needs to handle resumption explicitly.

8.12 What's Missing?

As long as this chapter is, there are still some aspects of SSL programming that we've omitted. This section quickly summarizes them.

Better Certificate Checking

As we discussed in Chapter 7, a more sophisticated approach to checking server certificates against the server hostname is to use the X.509 subjectAltName extension. In order to make this check, you would need to extract this extension from the certificate and then check it against the hostname. This isn't extremely difficult but it is intricate and tightly bound to the certificate processing API of the SSL toolkit, which is why we've omitted it here. Additionally, it would be nice to be able to check host names against wildcarded names in certificates. We don't show how to do this either.

rclient.c

```
6   int main(argc,argv)
7     int argc;
8     char **argv;
9     {
10      SSL_CTX *ctx;
11      SSL *ssl;
12      BIO *sbio;
13      SSL_SESSION *sess;
14      int sock;
15      /* Build our SSL context*/
16      ctx=initialize_ctx(KEYFILE,PASSWORD);
17      /* Connect the TCP socket*/
18      sock=tcp_connect();
19      /* Connect the SSL socket */
20      ssl=SSL_new(ctx);
21      sbio=BIO_new_socket(sock,BIO_NOCLOSE);
22      SSL_set_bio(ssl,sbio,sbio);
23      if(SSL_connect(ssl)<=0)
24        berr_exit("SSL connect error (first connect)");
25      check_cert_chain(ssl,HOST);
26      /* Now hang up and reconnect */
27      sess=SSL_get_session(ssl); /*Collect the session*/
28      SSL_shutdown(ssl);
29      close(sock);
30      sock=tcp_connect();
31      ssl=SSL_new(ctx);
32      sbio=BIO_new_socket(sock,BIO_NOCLOSE);
33      SSL_set_bio(ssl,sbio,sbio);
34      SSL_set_session(ssl,sess); /*And resume it*/
35      if(SSL_connect(ssl)<=0)
36        berr_exit("SSL connect error (second connect)");
37      check_cert_chain(ssl,HOST);
38      /*Now close everything down again*/
39      SSL_shutdown(ssl);
40      close(sock);
41      destroy_ctx(ctx);
42    }
```

rclient.c

Figure 8.27 Resuming sessions with OpenSSL

/dev/random

An increasing number of UNIX systems support a device called /dev/random. In essence, /dev/random acts like a continuous source of random bytes. You simply open the device and read from it. /dev/random operates by examining a number of system variables, timers, and interrupts and mixing them into a pseudo-random number generator to produce high-quality randomness as output. On systems with /dev/random, it is a useful source of randomness, either in place of or in addition to the OpenSSL or PureTLS randomness file. Windows 2000 also has a similar facility. We haven't shown how to access these facilities for portability reasons, but they do deserve mention.

Multiprocess Operation

The most significant omission from these examples is a method for servers to support more than one client. As we mentioned, our PureTLS-based server starts a new thread to actually serve clients but the SSL accepts all happen in the main thread. This is undesirable in a high-traffic server. Our OpenSSL-based server is completely incapable of handling more than one client at a time. Handling this case cleanly comes up quite frequently in Web applications—where multiple simultaneous connections at low cost are very important—and so we'll take it up in the next chapter when we discuss HTTP over TLS.

Better Error Handling

Note that these applications handle errors simply by exiting with an error. A real application would, of course, be able to recognize errors and signal them to the user or some audit log rather than just exiting.

8.13 Summary

This chapter has discussed techniques for programming applications that use SSL. We've provided two sets of example programs, one written in C using OpenSSL and another written in Java using PureTLS. These examples illustrate a number of common idioms seen in applications that use SSL.

Programming language matters. As we saw, PureTLS and OpenSSL mimic the networking APIs for the languages they were written in. Moreover, the sort of I/O discipline that is most appropriate depends on the capabilities of your language.

Context objects allow one-time initialization. A number of initialization procedures such as keyfile loading and randomness generation are time consuming. In order to avoid performing them repeatedly for every SSL connection, we initialize a context object and use that to create our new connections.

Single-source I/O is easy. In simple server cases, we often need to simply read from the SSL connection, perform some service, and respond. In such cases programming the appropriate reads and writes is straightforward—nearly identical to programming with sockets.

Multiplexed I/O is hard. When we need to multiplex between several sources of input, we use either threads or `select()` to arrange that we service all I/O needs without starvation. This requires careful attention to correct record framing and closure behavior.

SSL record framing causes problems. SSL data must be read from and written to the network one record at a time. Because applications often want to read and write data in sizes other than a record, `select()` often gives incorrect answers about the amount of data available. In such cases, programmers must use toolkit-specific mechanisms to determine the state of the SSL read and write buffers.

Closure must be handled carefully. It's important to be able to handle both locally initiated and unsolicited closes. With a locally initiated close, we must not simply close the connection but rather wait until a close_notify is received. With an unsolicited close, the programmer must arrange to send a close_notify but be prepared to recover if it's answered with an RST.

Session resumption APIs vary widely. PureTLS automatically resumes sessions where appropriate. OpenSSL requires user intervention. Some other C-based toolkits do resumption automatically as well.

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.

Ex.1024
Askeladden, L.L.C. / Page 361 of 788

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.

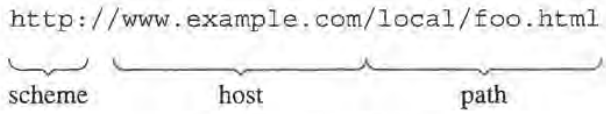


Figure 9.3 A simple URL

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.

Ex.1024
Askeladden, L.L.C. / Page 369 of 788

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.

9.14 URLs and Reference Integrity

The vast majority of HTTP connections are initiated by dereferencing some URL. Because HTTPS uses the separate ports strategy, the URL must indicate that the protocol to be used is HTTPS rather than HTTP. This is done simply by using the `https` protocol identifier rather than the `http` protocol identifier:

```
https://www.example.com/example.html
```

Downgrade Attacks

The use of the `https` protocol identifier protects against downgrade attacks. The client expects to initiate an SSL connection to the server—indeed, there is no way for the server (or an attacker) to indicate that it *won't* speak SSL. The attacker can only cause an error to be generated. Of course, as we discussed earlier, users can sometimes be tricked into retrying with an HTTP connection if the HTTPS connection fails.

However, the `https` identifier provides only one bit of information: The client should expect security. This doesn't provide any indication of what security properties to expect. With SSLv2, this presented a serious problem because SSLv2 had no protection against an attacker downgrading the connection to a weaker algorithm. However, SSLv3 and TLS do provide algorithm downgrade protection. Because SSLv3 and TLS also provide protection against downgrading to SSLv2, as long as the server speaks SSLv3 at all, the negotiation cannot be downgraded to a weaker algorithm.

As long as the server is correctly authenticated, then the handshake can be conducted securely. Thus, provided the client doesn't accept anonymous DH and adequately checks the server's identity as described in the next section, HTTPS (with SSLv3) is secure against downgrade attacks. However, if the server isn't authenticated correctly, the attacker can mount a man-in-the-middle attack.

End-point Authentication

The HTTPS spec provides a somewhat complicated algorithm for checking the server's identity. The intent is to accommodate a variety of special cases. In most cases the server's expected hostname will be available from the URL or some similar source. If the client has the server's expected hostname, it must check it against the server's certificate.

Alternatively, the client might have specific information about the server's certificate rather than the hostname, in which case it's required to check that. Finally, although it is explicitly allowed to perform no checks at all, this should really never be done because it leaves the connection open to an active attack.

Figure 9.5 shows a flowchart for the hostname checking procedure. If a subjectAltName extension is present with type dNSName, that must be checked. Currently, no major CA issues certificates of this type but it is hoped that in the future they will do so, because this approach is far cleaner than using the Common Name—the Common Name approach requires an implicit agreement between the CA and users about the semantics of the Common Name, whereas the semantics of dNSName are clear.

If the dNSName is not present, however, the Common Name must be checked. Because a DN may have two RDNs which both specify Common Name values, the specification explicitly requires that the most local RDN be checked. No explicit rule is provided for the case in which a single RDN contains two Common Name values. The obvious choice is to accept the certificate if any of the names match.

RFC 2818 also allows wildcarded DNs. Part of the name is replaced by a "*". The "*" can match any string. Thus, if the Common Name is `*.example.com` this certificate would work for `foo.example.com`, `bar.example.com`, etc. Wildcarding only really works well if you have a number of virtual servers run by the same

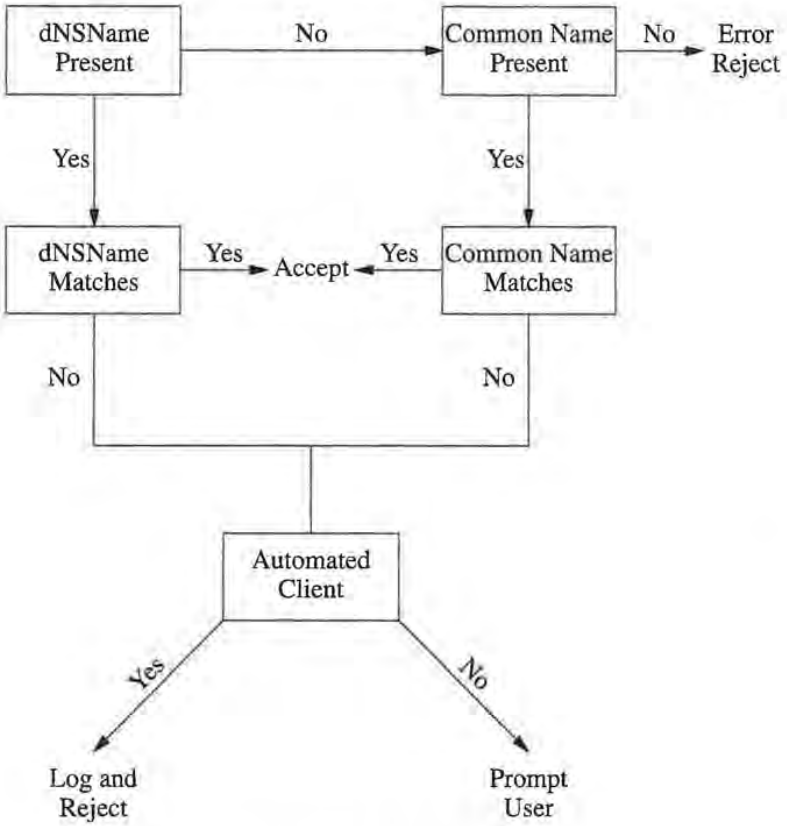


Figure 9.5 Certificate checking algorithm

organization. You can't use it to represent (say) `www.domain.com` and `www.anotherdomain.com` because any wildcard that matched both would also match other domains not owned by the certificate-holder.

Failure Behavior

The obvious response to a mismatch between the certificate and the server's expected identity is simply to terminate the connection. Unfortunately, this behavior has been found to be very irritating to users. Although such a mismatch may in principle reflect an active attack, in practice it more likely reflects a misconfiguration of the server.

Netscape Navigator has traditionally responded to mismatches (and to unverifiable certificates) by prompting the user with a dialog box identifying the error and offering to shut down the connection or continue. Older versions of Internet Explorer simply shut down the connection and refused to continue. However, newer versions of IE allow the user to continue.

The HTTPS specification provides a number of alternatives for handling a mismatch. A client such as a browser must at minimum notify the user of the mismatch. It may offer a choice to continue or simply terminate the connection. An automated client must log the error and should terminate the connection.

Although continuing in the face of a certificate mismatch is known to be insecure against active attack, it's important to remember that users will often attempt to refetch the page using HTTP if the HTTPS fetch fails. In cases in which the user will attempt an insecure fetch, then it is still better to accept the wrong certificate, because there is some chance that the problem is simply a configuration error. For example, the server administrator might have changed the DNS name of the server without getting a new certificate. Figure 9.6 shows the possible cases. As you can see, accepting the wrong certificate is never worse (and sometimes better) than having the user retry with HTTP.

	Client Behavior		
Situation	Failure	Accept Certificate	Retry with HTTP
Configuration Error	Failure to communicate	Communication OK	Data transmitted in clear
Attack	Attack prevented	Successful attack	Data transmitted in clear

Figure 9.6 Results of various error handling strategies

Determining how to behave when certificate processing fails requires examining your security policy and the attitudes of your users. It's never appropriate to simply accept a certificate that the implementation knows to be wrong. However, in some circumstances it might be superior to allow the user to accept the certificate anyway. As Figure 9.6 shows, this may be superior to simply refusing to connect altogether. When certificate validation fails, both Netscape and Internet Explorer pop up a dialog to allow the user to continue with the connection.

User Override

As discussed above, Netscape and later versions of IE allow the user to continue even if the certificate does not match. From the user's perspective, they behave essentially identically. However, from a bits-on-the-wire perspective they behave rather differently.

Figure 9.7 shows a Netscape client connecting to a server with an unacceptable certificate. As you might expect, the client waits for the server to send the Certificate message and then holds the connection open while it prompts the user to accept the

certificate or cancel the connection. In the trace shown in Figure 9.7, the user has chosen to accept the certificate, so the handshake continues.

Note that depending on what exactly is wrong with the certificate (bad hostname, unknown CA, expired, or maybe all three), the user may have to click through a number of dialogs to accept the certificate and continue. This can take a while and the server must therefore set its timeouts long enough that it doesn't time out during this process.

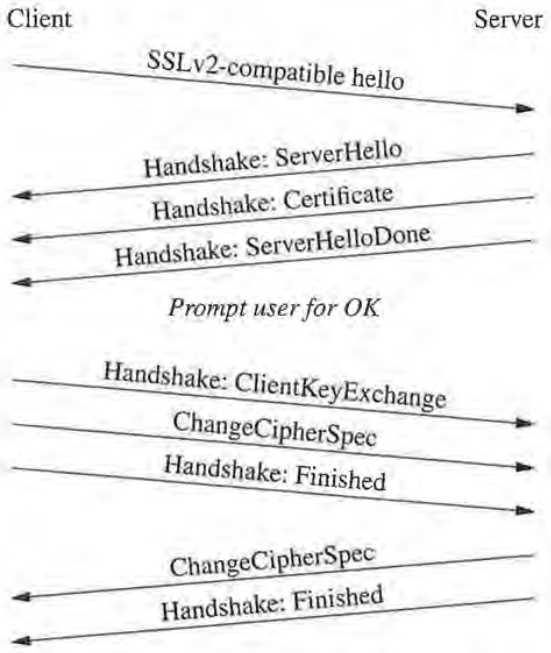


Figure 9.7 Netscape: user accepts bad certificate

Figure 9.8 shows Netscape's behavior when the user chooses to cancel the handshake. After the user presses the cancel button in the dialog, the client sends a bad_certificate alert to the server and terminates the connection.

IE's behavior is somewhat less obvious. When IE receives a bad certificate, it completes the handshake and then closes the connection, as shown in Figure 9.9. Once the connection is closed, IE prompts the user to accept or reject the certificate. If the certificate is rejected, the connection just stays closed. If the certificate is accepted, IE initiates a new connection to the server. Note that IE resumes the SSL session, thus avoiding the cost of a full new handshake.

Netscape's behavior is what you would expect from reading the spec. However, IE's behavior, though surprising, is not actually forbidden. The primary advantage of IE's behavior is that it does not consume server resources while the user is deciding whether or not to accept the certificate. Because server sockets are a somewhat limited resource,

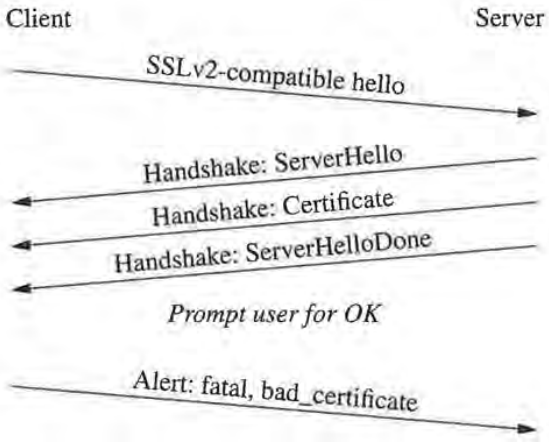


Figure 9.8 Netscape: user rejects bad certificate

this can be a valuable optimization for heavily loaded servers. However, since a server in heavy use should have a certificate from a well-known CA, this case isn't actually that important.

IE's approach has two disadvantages: First, the server doesn't get any feedback that the client didn't like its certificate. If the server administrator actually reads the log files, a bad_certificate alert would warn him that there was a problem. Second, if the server isn't using session caching or times out the client's session before the reconnect, a new full handshake will be required, which is expensive for both the client and the server.

Since responding to the accept/reject dialogs is irritating to the user, both Netscape and IE cache the user's acceptance of the certificate. Netscape also allows the user to specify that this certificate should be accepted permanently so that the user will never be bothered again. Deciding whether this is a bug or a feature is left up to the reader.

Reference Sources

It's important to note that reference integrity depends on the reference being safely delivered to the client. If the reference is delivered in such a way that it can itself be trusted, then the previously mentioned properties hold. For instance, the reference might itself be delivered over an HTTPS link. However, in cases where the reference is delivered over an unsecured link, then it's possible to mount active attacks by changing the page containing the reference. We'll see such an attack in Section 9.20. No possible reference integrity checking can prevent this form of attack.

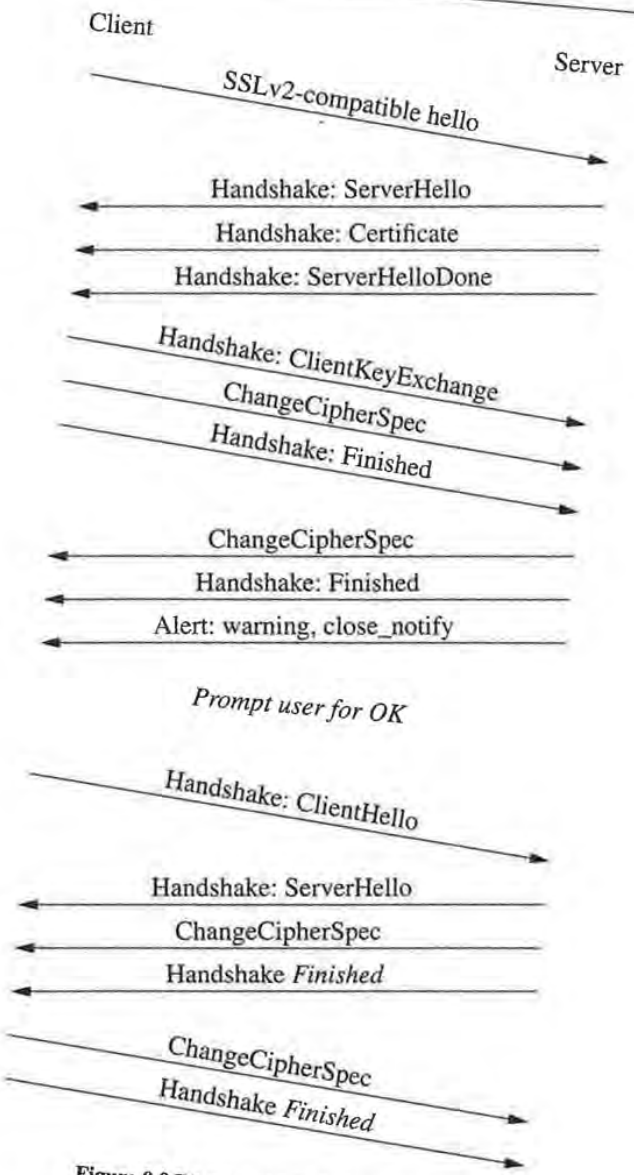


Figure 9.9 IE: user accepts bad certificate

Client Identity

In some non-Web environments the "client" is actually another host. If the server knows that this is the case, then the server should perform the same sorts of checks that are described above.

9.15 Connection Closure

Finally, we must consider the issue of closure. The correct closure behavior is easily summarized: Whenever either side wishes to close the connection, it should send a close_notify and optionally wait for a close_notify. This is required both by the TLS and HTTPS RFCs. The issue that we need to be concerned with is how to behave when a TCP FIN is received without a close_notify. In some cases, this situation represents an obvious possible attack. In others, it may simply be a programming error. We'd like to be able to distinguish such cases in order to avoid recording a large number of spurious errors.

Session Resumption

Recall from our discussion in Chapter 7 that the rules about session resumption are clear. In no case should an SSL implementation resume a session when it has received a premature close. To do so would be to explicitly violate the SSL specification. However, it is permissible to send one's own close_notify and then shut down the connection without waiting for the other side's close_notify. In such a case, an SSL implementation may resume the session.

Error Handling

Primarily we're concerned with client behavior. First, HTTP semantics imply that errors are more serious for clients than servers. A server that receives an end of data during the request, whether via proper SSL closure or via a TCP FIN will behave in essentially the same way: abandon processing the request. Second, when servers encounter errors, they usually simply log them to audit logs. Thus, a server author can safely log anything out of the ordinary.

By contrast, clients often signal errors directly to users via dialog boxes and the like. It's therefore important to avoid both failure to report legitimate errors (because this hides important information from the user) and reporting non-errors (because this overwhelms the user and increases the chance that the user will ignore legitimate errors).

As a general rule we want to be able to distinguish between two classes of error:

Likely programming error. The HTTP traffic indicators about connection closure don't match the SSL behavior. We want to be forgiving when we receive a message that is broken but doesn't indicate a security problem.

Truncation attack. The client received a response that might be incomplete.

Likely Programming Error

Consider the case where the client expects the connection to close but then receives a premature close. For instance, the client receives a response from the server with the `Connection: close` header. The client reads the entire amount of data specified by the `Content-Length` header. Thus, the client expects the connection to close immediately. Now, the client receives a bare FIN without ever receiving a close_notify. It's possible that an attacker forged the FIN but it's more likely that the server has made an error and closed the connection incorrectly. We know that the client has received all the data that the server is supposed to send, so we don't need to worry that the page is damaged, and the client can display it without reporting an error.

A slightly more difficult class of cases involves those in which the client doesn't expect any traffic from the server but it also does not expect the server to close the connection. Imagine that the client receives a response with a `Content-Length` header and it has read all of that data. However, the client believes that it's using persistent connections and so does not know if the server will close the connection. (HTTP permits the server to close the connection without notifying the client.) Thus, if the client receives a premature close after reading the whole response, it's possible that an attacker is mounting a denial-of-service attack, but again it's more likely that the server is broken. In any case, the client knows that the response that it actually did receive is complete and can display it.

Finally, consider the case where a proper close_notify occurs before it is expected, for instance before the number of bytes specified by the `Content-Length` header has been read. This is an obvious programming error on the server because an attacker could neither forge a close_notify nor undetectably change the `Content-Length` header.

Truncation Attack

Recall that older versions of HTTP use connection closure to signal end of data. Thus, if a client expects to receive more data from the server but instead receives a premature close, it must signal an error. First, consider the case where no `Content-Length` header is present. Since the client has no indicator of when the response is finished other than the connection close, receipt of a close_notify is essential to know that the entire response was received.

Similarly, if the `Content-Length` header is present but a premature close is found before the full response has been read, this must be treated as an error. It's of course possible that the server has merely miscomputed the length, but there is no way to distinguish this case from an attack. Note that we handle this case differently from the case where the full length has been read, in which case we know that we've received the full response and any possible attack merely prevented us from receiving the next response.

In any case where there has been a possible truncation attack and a response might have been shortened, the client should warn the user, usually by printing an error message or popping up a dialog indicating that the connection closed unexpectedly and the data might be truncated. Particularly paranoid clients might refuse to display any of the response at all.

9.16 Proxies

HTTPS interacts uncleanly with proxies. The general problem is that HTTPS semantics imply that the client negotiated an SSL session with the target server. Once the client has done so, all the data that passes between the client and server are encrypted. The result is that caching proxies won't work. It's reasonable to regard this as a feature because the data we're attempting to transmit between client and server are supposed to be confidential anyway. However, even in cases where no confidentiality is employed (i.e., cipher suites with only message integrity) no caching is possible.

CONNECT

A more serious problem is that the usual HTTP proxy semantics do not work with HTTPS. Recall that the proxy examines the client's request in order to determine which server to connect to. HTTPS requires that the client pass its request over the encrypted channel, so this approach is obviously no longer possible. This is a more serious issue than the inability to cache, because proxies are the only method available for traversing certain firewalls. Thus, without special support, HTTPS is incompatible with those firewalls.

The special support comes in the form of a new proxy method: `CONNECT`, documented in RFC 2817 [Khare2000]. The `CONNECT` method instructs the proxy to initiate a TCP connection to the specified remote server and then pass data between client and server without examining it or changing it. The client then transmits the SSL data to the proxy as if the proxy were the server. Figure 9.10 shows the beginning of such a connection up through the transmission of the ClientHello. Note that the client supplies the host and port of the specified server to the proxy in the Request-URI field of the request.

From the perspective of the HTTPS user or programmer, this approach appears to work well. However, from the perspective of the firewall administrator, the results are less positive. Firewall administrators are often concerned with controlling the sorts of traffic that traverse the firewall. For instance, they might wish to screen out particular types of Web content. Because traffic transmitted with `CONNECT` is supposed to be unexamined—and is in theory unreadable because it's encrypted—this type of screening is now impractical.

The protocols that can cross firewalls using `CONNECT` are not even limited to SSL over HTTP. Although in principle the firewall could attempt to determine that the traffic was SSL, in practice this is too difficult. Thus `CONNECT` method permits a client to punch a hole in the firewall and pass arbitrary traffic through that hole. An administrator can attempt to limit the hosts and port numbers to which a client connects, but this is a fairly weak sort of enforcement.

Man-in-the-Middle Proxies

As we've seen, in general proxies cannot view the traffic between client and server. However, it's possible to arrange matters so that the proxy can do so by having the proxy mount a man-in-the-middle attack on the connection. The proxy accepts the `CONNECT` request but instead of proxying the data it negotiates a pair of SSL

```
New TCP connection: romeo(2577) <-> romeo(80)
1 949442170.3636 (0.0002) C>S
data: 37 bytes
-----------------------------------
CONNECT www.rtfm.com:443 HTTP/1.0

-----------------------------------

2 949442170.3686 (0.0052) S>C
data: 102 bytes
-----------------------------------
HTTP/1.0 200 Connection established
Proxy-agent: Apache/1.3.9 (Unix) mod_ssl/2.4.10 OpenSSL/0.9.4

-----------------------------------

3 949442170.4403 (0.0769) C>S   Handshake
      ClientHello
        Version 3.1
        cipher suites
                TLS_DHE_DSS_WITH_RC4_128_SHA
                TLS_DHE_DSS_WITH_RC2_56_CBC_SHA
                TLS_RSA_EXPORT1024_WITH_RC4_56_SHA
                TLS_DHE_DSS_EXPORT1024_WITH_DES_CBC_SHA
                TLS_RSA_EXPORT1024_WITH_DES_CBC_SHA
                TLS_RSA_EXPORT1024_WITH_RC2_CBC_56_MD5
                TLS_RSA_EXPORT1024_WITH_RC4_56_MD5
                TLS_DHE_RSA_WITH_3DES_EDE_CBC_SHA
                TLS_DHE_DSS_WITH_3DES_EDE_CBC_SHA
                TLS_RSA_WITH_3DES_EDE_CBC_SHA
                TLS_RSA_WITH_IDEA_CBC_SHA
                TLS_RSA_WITH_RC4_128_SHA
                TLS_RSA_WITH_RC4_128_MD5
                TLS_DHE_RSA_WITH_DES_CBC_SHA
                TLS_DHE_DSS_WITH_DES_CBC_SHA
                TLS_RSA_WITH_DES_CBC_SHA
                TLS_DHE_RSA_EXPORT_WITH_DES40_CBC_SHA
                TLS_DHE_DSS_EXPORT_WITH_DES40_CBC_SHA
                TLS_RSA_EXPORT_WITH_DES40_CBC_SHA
                TLS_RSA_EXPORT_WITH_RC2_CBC_40_MD5
                TLS_RSA_EXPORT_WITH_RC4_40_MD5
        compression methods
                NULL
```

Figure 9.10 Using CONNECT with a proxy

connections, one to the client and one to the server. It then passes data between client and server but the data is available in cleartext form to the proxy. Figure 9.11 shows this process.

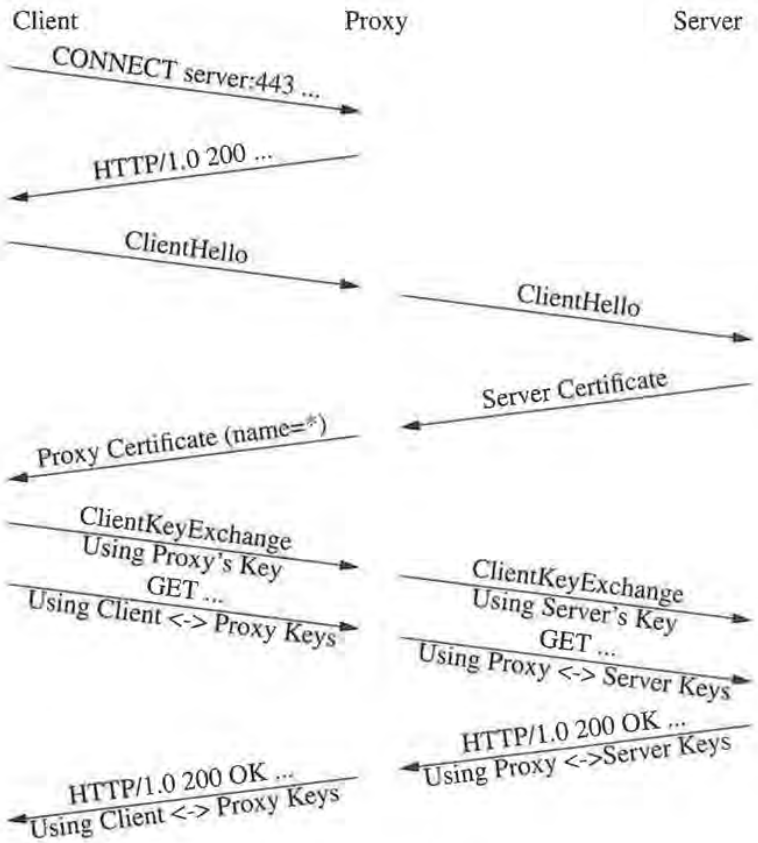


Figure 9.11 A man-in-the-middle proxy in action

But how is this possible, since SSL is supposed to be resistant to man-in-the-middle attacks? It requires the cooperation of the client. The proxy has a very special certificate that has "*" as its Common Name. Because the hostname match is wildcarded, the client will accept this certificate no matter which host it wants to connect to. Obviously, no legitimate CA will issue such a certificate because it would let the holder impersonate any server. To work around this, the administrator needs to run his own CA and

install it into the browser. Because browsers that aren't under the control of the administrator won't have this CA in their list of trusted CAs, they can't be fooled by this dangerous certificate.

> Note that RFC 2818 forbids such certificates from being issued, since "*" can only match a single domain name component. However, as we'll see in Section 9.17, Netscape allows a "*" to match anything. In order to make this trick work with RFC 2818-compliant browsers, you would need to have a certificate containing a series of patterns: "*", "*.*", "*.*.*", ...

Note that this sort of proxy has serious drawbacks. If the proxy is compromised, the traffic of all clients is compromised. Because the proxy is outside (or part of) the firewall, it makes a good target for attack. Also, because the client does not connect directly to the server, SSL client authentication no longer works.

Cipher Suite Translation

Due to the old U.S. export regulations, a curious asymmetry exists between clients and servers. The two most popular browsers, Netscape and Internet Explorer, are both made by U.S. companies and thus were subject to strict U.S. export regulations. Because downloading the more secure "domestic" version of the browsers was difficult, many Americans have "export" browsers. As a consequence, there are a lot of browsers available with only weak cryptography.

By far the most popular Web server is Apache. Although Apache was written in the U.S., there are several easily available SSL/TLS patches for Apache written outside the U.S. Also, because Netscape's servers weren't free, getting the "domestic" Netscape server wasn't much harder than getting the "export" server. Thus, a fairly high percentage of the Netscape servers sold in the U.S. had strong cryptography. In general, most servers suppport strong cryptography.

C2 Net took advantage of this situation to offer a man-in-the-middle proxy called SafePassage. SafePassage runs on the same computer as the client. It allows "export" clients to connect to it with weak cryptography, but then itself uses strong cryptography to connect to the server. Because SafePassage was manufactured outside the U.S., foreigners could get a copy and upgrade their browsers. Thus, the weak cryptography is only used to encrypt data that never leaves the machine anyway.

The need for SafePassage dwindled with the introduction of Fortify, a patch to Netscape that upgraded "export" versions to support strong cipher suites, and finally vanished when the U.S. government liberalized its export regulations in January 2000. Nevertheless, SafePassage was the first example of a man-in-the-middle proxy and thus an innovative product. SPYRUS has used a similar approach to translate ordinary RSA-based cipher suites to cipher suites using the U.S. government's FORTEZZA card.

9.17 Virtual Hosts

HTTPS also interacts poorly with virtual hosts. Once again, the problem is that the SSL connection is established before any HTTP data is transmitted. Recall that ordinarily the server uses the `Host` header in the request to determine which virtual host is being accessed. Because the request is transmitted after the handshake occurs, the server must negotiate the SSL connection without the guidance of the `Host` header.

Consider the issue of which server certificate to present to the client. If the servers were in fact on separate machines, we would expect them to have different certificates and under normal circumstances we would expect each virtual server to have its own certificate. However, knowing which certificate to present requires knowing which virtual server the client is attempting to connect to. Similar considerations apply if the virtual servers have different security policies and want to negotiate different cipher suites.

However, there is another method of implementing virtual servers that is consistent with SSL. It is possible for a machine to be configured so that a single network interface has multiple IP addresses (called aliases). Each virtual server is assigned its own alias. Thus, when the server accepts the connection, it looks at the IP address that it accepted the connection on and uses that to determine which virtual server the client is attempting to access.

As we implied in Section 9.11, another approach would be to allow the anchor to specify the server's certificate, perhaps somewhere in the URL. Then a server could use the same certificate for all its virtual servers while still preserving reference integrity.

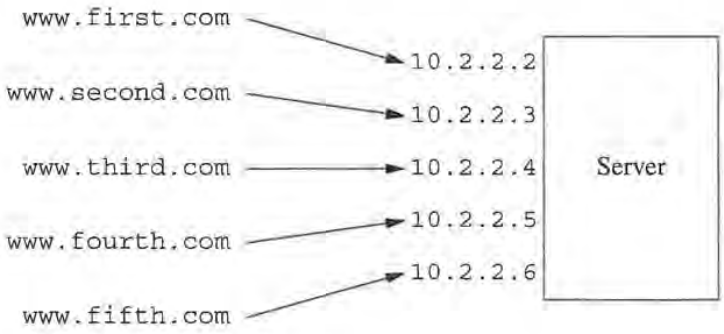


Figure 9.12 Virtual hosts using IP aliases

Figure 9.12 shows a sample virtual server configuration with five virtual servers. The DNS name `www.first.com` is assigned to 10.2.2.2, the name `www.second.com` is assigned to 10.2.2.3, etc. Note that this technique is distinctly different from using CNAME records. Although multiple Web addresses point to the same server, they don't

point to the same DNS A record but rather to different A records with different IP addresses.

This situation can be considered a design flaw in SSL. It would be fairly straightforward for the client to signal the DNS name of the server it was trying to connect to in the ClientHello, thus allowing the server to behave appropriately. This change would not compromise confidentiality because it's fairly easy for an attacker to deduce from other public network data (the IP header and the rest of the handshake) which server the client expects to connect to. This is one of several proposed changes for the next version of TLS—if there ever is one.

Note that this change to SSL would not relieve the client of the obligation of checking the server's certificate against the expected identity. Failure to make that check *would* introduce a vulnerability. The only purpose of this change would be as a pre-handshake hint to the server.

Multiple Names

Another slightly less tasteful approach to the virtual hosts problem is to allow a certificate to serve multiple hosts. RFC 2818 specifies how to do this using wildcarding. See Section 9.14 for more details.

Netscape supports wildcarding. In fact, Netscape's wildcarding approach is substantiallly more flexible (and complicated) than that described in RFC 2818. It supports

*—anything at all

?—any single character

—the end of a string

[abc]—one occurrence of a, b, or c

[A-Z]—any character in the range A to Z

[^ab]—any character other than a or b

~—negates the pattern that follows

(pattern1|pattern2)—either pattern1 or pattern2

In order to allow special characters to appear in patterns, Netscape also uses \ as an escape character. However, since none of the special characters can appear in domain names in any case, this isn't really necessary. Netscape's approach to wildcarding is described in [Netscape1995a].

In addition to wildcarding, Internet Explorer supports another approach. Each certificate is allowed to have multiple Common Names. Thus, the certificate could contain names for `foo.example.com` and `bar.example.com`. This approach has the advantage that a certificate can be issued for multiple unrelated domains. However, it has the disadvantage that if even a single new virtual host is added for a certificate, it must be totally reissued. By contrast, wildcarding allows the certificate to match any number of virtual hosts as long as they match the pattern.

9.18 Client Authentication

Finally, consider the decision of whether the server should request client authentication using certificates. Some servers require client authentication for all pages that are fetched via HTTPS but much more common is for only certain pages to require client authentication. This policy is difficult to support cleanly with HTTPS.

Yet again, the problem is that the SSL handshake must be completed before the HTTP request is transmitted to the server. Thus, the server cannot know what resource is being requested (and therefore whether to request client authentication) until after the time at which it would have liked to request client authentication.

One approach would be for the server to always request client authentication but to allow the client to connect without it. Then, if the request turns out to require client authentication and the client didn't provide it, to throw an error at that point. Although technically workable, this has unfortunate user interface consequences: Users are almost always prompted for which certificate to use when authenticating. Thus, always requesting client authentication inconveniences all users—even those who do not need to authenticate. This inconvenience is usually considered enough to disqualify this approach.

The commonly used approach is for the server to negotiate an ordinary SSL connection for all clients. Then, once the request has been received, the server determines whether client authentication is required. If it isn't required, the server simply services the request. If it is required, the server requests a rehandshake using HelloRequest. In this second handshake, the server requests client authentication. Figure 9.13 shows a connection where this has happened.

```
New TCP connection: romeo(4569) <-> romeo(443)
1 948675468.0084 (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 948675468.0139 (0.0054) S>C   Handshake
      ServerHello
        session_id[32]=
          22 0f e0 2c 1e 63 a3 b2 cd 68 6a af
          6c f5 bc ff 0d 77 7d d4 dc bd e5 3f
          80 1e da 33 3c 79 f4 0d
        cipherSuite             TLS_RSA_WITH_RC4_128_MD5
        compressionMethod               NULL
```

```
3 948675468.0139 (0.0000) S>C   Handshake
      Certificate
4 948675468.0139 (0.0000) S>C   Handshake
      ServerHelloDone
5 948675468.0281 (0.0142) C>S   Handshake
      ClientKeyExchange
6 948675468.0796 (0.0515) C>S   ChangeCipherSpec
7 948675468.0796 (0.0000) C>S   Handshake
      Finished
8 948675468.0818 (0.0021) S>C   ChangeCipherSpec
9 948675468.0818 (0.0000) S>C   Handshake
      Finished
10 948675468.0830 (0.0012) C>S   application_data
    data: 291 bytes
    ---------------------------------
    GET /secure/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 948675468.0892 (0.0062) S>C   Handshake
      HelloRequest
12 948675468.0912 (0.0019) C>S   Handshake
      ClientHello
        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
        compression methods
                  NULL
```

(continued)

Figure 9.13 *(continued)*

```
13 948675468.0928 (0.0016) S>C   Handshake
      ServerHello
        session_id[32]=
          20 60 4a f0 84 cb b4 7c 7e 9b af d0
          3c fe 70 4c 47 58 96 18 2e 02 89 39
          94 50 4b e2 77 b0 2c a8
        cipherSuite           TLS_RSA_WITH_RC4_128_MD5
        compressionMethod           NULL
14 948675468.0928 (0.0000) S>C   Handshake
      Certificate
15 948675468.1005 (0.0077) S>C   Handshake
      CertificateRequest
              certificate_types           rsa_sign
              certificate_types           dss_sign
        certificate_authority
          30 49 31 0b 30 09 06 03 55 04 06 13
          02 55 53 31 13 30 11 06 03 55 04 0a
          13 0a 52 54 46 4d 2c 20 49 6e 63 2e
          31 13 30 11 06 03 55 04 0b 13 0a 43
          6f 6e 73 75 6c 74 69 6e 67 31 10 30
          0e 06 03 55 04 03 13 07 54 65 73 74
          20 43 41
16 948675468.1005 (0.0000) S>C   Handshake
      ServerHelloDone
17 948675471.2692 (3.1687) C>S   Handshake
      Certificate
      ClientKeyExchange
        EncryptedPreMasterSecret[128]=
      CertificateVerify
        Signature[128]=
          77 db fe 35 67 0d fa 1d 7d ea 2e 70
          ae 8f b4 a8 6f 26 91 df 81 1b 8b c6
          e1 7a 94 67 ed 9c ad be be 1a 71 74
          6e 1b b1 ae c1 9e 26 81 d7 6c 30 ae
          67 54 b8 12 f5 cf 0a e2 71 81 ae 0e
          8a 14 ee 76 de 39 44 33 9b 6e fd e7
          19 51 73 43 67 28 5d bf d3 74 a2 b8
          4a f1 32 0a 02 c8 27 b7 bd eb 79 38
          ca 3f 91 c7 95 46 b0 c3 32 ff 07 0d
          7b 54 28 30 c3 f4 67 f1 f1 58 e9 7c
          61 4c a7 28 51 c5 ad 92
18 948675471.2997 (0.0304) C>S   ChangeCipherSpec
```

```
19 948675471.4797 (0.1800) C>S   Handshake
      Finished
20 948675471.4811 (0.0013) S>C   ChangeCipherSpec
21 948675471.4811 (0.0000) S>C   Handshake
      Finished
22 948675471.4848 (0.0037) S>C   application_data
    data: 423 bytes
    ---------------------------------
    HTTP/1.1 200 OK
    Date: Mon, 24 Jan 2000 00:57:48 GMT
    Server: Apache/1.3.9 (Unix) mod_ssl/2.4.10 OpenSSL/0.9.4
    Last-Modified: Sun, 23 Jan 2000 23:08:52 GMT
    ETag: "8e469-95-388b8a04"
    Accept-Ranges: bytes
    Content-Length: 149
    Connection: close
    Content-Type: text/html

    <HTML>
     <HEAD>
      <TITLE>Test</TITLE>
    </HEAD>
    <BODY>
    <H1>
    Client Auth Test Page
    </H1>
    This page must be fetched with client auth.
    <P>
    </BODY>
    </HTML>
    ---------------------------------
23 948675471.4895 (0.0047) S>C   Alert
        level       warning
        value       close_notify
Server FIN
24 948675471.5060 (0.0164) C>S   Alert
        level       warning
        value       close_notify
Client FIN
```

Figure 9.13 Rehandshake with client authentication

When the client first connects in Figure 9.13, the server does an ordinary SSL handshake, exactly as we saw before. Once the handshake is complete, the client sends the request in record 10. Again, this is identical to the request in Figure 9.4 except that

we're requesting a different resource `/secure/tmp.html` instead of `/tmp.html`.

At this point, the server examines its access control information and determines that the resource being requested requires client authentication and that the current connection doesn't have client authentication. Thus, the server will need to request a rehandshake in order to force the client to authenticate with its certificate. This is accomplished by sending the HelloRequest in record 11.

The HelloRequest causes the client to initiate a new handshake. Although technically the client is not required to do so immediately, the client can't do anything else until the server sends its response. The server sending the HelloRequest is a good indicator that the server may be waiting for the new handshake before it responds, so in practice the client will initiate a new handshake immediately, sending the ClientHello in record 12.

From here on in, the handshake is just like the SSL client authentication handshake we saw in Chapter 4. Note, however, that this entire process is taking place over the encrypted channel established in the first handshake. Finally, when the handshake completes in records 18–22, the server sends its response back (using the freshly negotiated session) in record 22.

Performance Consequences

Note that the client does not offer to resume the session when it sends its second ClientHello. Although it could do so, session resumption is incompatible with client authentication and so the server would have to refuse to resume the session in any case. As a consequence, the client and the server need to perform an entirely new handshake. This highlights the major drawback of the renegotiation approach: It doubles the amount of work that the server must do in order to establish the connection—and HTTPS was already much slower than bare HTTP. (Assuming that the keys are RSA keys, most of the client's work is doing its signature, so the extra cost of the extra handshake over simply authenticating in the first handshake to the client is insignificant.)

Other Approaches

Clearly, this problem represents a mismatch between the behavior of SSL and common HTTP usage. It's tempting to call it an SSL design error, but it's important to recall that SSL is a general protocol and that it's inherent in SSL design that the cryptographic keys must be negotiated before the first piece of application data is transmitted. Remember that the request might contain sensitive information, so it must be protected.

Nevertheless, it would be possible to improve the situation. First, SSL could allow you to perform client authentication on a resumed session. This is quite technically feasible and is simply a matter of slightly changing the behavior of the SSL messages. A number of possible modifications are possible, but one would be to use ServerHelloDone even in case of session resumption. Then, the server would have the opportunity to send a CertificateRequest. This should not cause a performance problem because the entire set of server messages can be transmitted in one handshake record and hence one TCP segment.

We cannot use the approach that we used with virtual servers where we indicated which server the client wanted to connect to. To work for this problem, the client would have to transmit the Request-URI in the clear over the wire. But the Request-URI itself may be sensitive information and an attacker cannot necessarily deduce it from sniffing. Thus, this approach cannot be used securely.

Another approach would be to have an indication in the URL that client authentication is required. This would also require SSL to have a mechanism for the client to unilaterally offer client authentication. The client could then make that offer based on the URL. This approach has the problem that if a server changed its access control settings it would then need to arrange to change all the relevant references. However, it would always be possible to renegotiate if the client made a connection using an out of date URL. There is currently no standard for how to do this, but it would be fairly straightforward to design one.

9.19 `Referrer`

In 1997, Daniel Klein observed that the `Referrer` header enabled a passive sniffing attack on HTTPS connections. The `Referrer` header simply contains the URL of the page that the user was viewing when he clicked the link that referenced the current page. `Referrer` provided a convenient way for sites to see which other sites were pointing at them, which was useful for a variety of statistics gathering purposes.

The attack was a result of the interaction of HTTP `GET` and referrer. Recall that when HTML forms are being used with `GET` the arguments are placed on the end of the Request-URI. Thus, when the user clicked on any link on the page resulting from the forms submission, the arguments to the form appeared in the `Referrer` header. If the link was an HTTP link, then the arguments would be transmitted in the clear on the next request. However, even if the link was an HTTPS link but was to a site other than the site to which the user submitted the form, that site would get the arguments in the `Referrer` header.

A user might have been understandably upset if one of the form fields revealed was his credit card number. The general response to this attack is to use forms that submit with HTTP `POST`. `POST` puts the arguments in the message body instead, thus neatly thwarting the attack.

9.20 Substitution Attacks

Recall from Chapter 7 that reference integrity requires that the references be reliably obtained. A number of different attacks have been described based on this principle. The general idea is to mount a man-in-the middle attack on the Web page that contains the first reference to an HTTPS page. The attacker then replaces the HTTPS URL with a reference to his own site instead. Because the URL matches the attacker's certificate, this completely bypasses the client's reference integrity checks.

Of course, it's possible (though unlikely) that the user will examine the URL displayed by the browser to see if it's what's expected. To counter this, the attacker can obtain a Web site with a name that looks like the victim's Web site (e.g., `www.foos0ft.com` instead of `www.foosoft.com`. Only a very astute user will notice this sort of substitution.

User Overrides

As we discussed earlier, even in cases where the reference does not match the server's certificate, many browsers give the user the opportunity to continue the connection. It's easy to see how to take advantage of this behavior to mount an attack. The attacker simply arranges for a certificate that the user might believe was appropriate (for instance, the `foos0ft` substitution in the previous section) and then takes over the client's HTTPS connection. In many cases, the user will allow the connection to go through, in which case the attack has succeeded.

9.21 Upgrade

RFC 2817 [Khare2000] defines a method for using the HTTP/1.1 `Upgrade` header to do upward negotiation from HTTP to HTTP over TLS. This approach was intended to provide an alternative to the popular HTTPS approach, without requiring the allocation of a separate port. In practice, however, there has been essentially no deployment of the Upgrade approach in the Web context. Most of the interest in Upgrade appears to be for other protocols which use HTTP, in part because the IESG has resisted standardizing HTTP-based protocols which use separate ports for TLS.

Client Requested Upgrade

The simplest case to understand is one in which the client requests upgrade. The client simply includes the header `Upgrade: TLS/1.0` in its request, as shown in Figure 9.14. Note that this request contains the `Host` header. Thus, the server knows which host the client intends to talk to and virtual hosting works properly without having multiple IP addresses. Note the use of the `TLS/1.0` token. No `SSL` token was standardized so there is no official way to upgrade to SSL.

```
GET /foo.bar HTTP/1.1
Host: www.example.com
Upgrade: TLS/1.0
Connection: Upgrade
(blank line)
```

Figure 9.14 An Upgrade request

If the server is willing to upgrade, it sends a dummy response with the `101` response code, as shown in Figure 9.15. Immediately after sending the dummy response, it begins TLS negotiation. When the client receives the response, it is expected to send a ClientHello. Only after the TLS connection has been negotiated does the server send its response to the client's original request. Note that the server's `Upgrade` header contains both the `TLS/1.0` and `HTTP/1.1` tokens. The tokens are intended to be passed in "bottom-up" order. Thus, the server is saying that it will speak HTTP over TLS. However, the server could choose not to upgrade and simply respond to the client's message.

```
HTTP/1.1 101 Switching Protocols
Upgrade: TLS/1.0, HTTP/1.1
Connection: Upgrade
(blank line)
```

Figure 9.15 A response accepting an Upgrade request

One drawback of this approach that should be immediately obvious is that the client's original request is sent in the clear. This is problematic because the client's request might contain sensitive information. RFC 2817 provides a mechanism called *mandatory upgrade*. What this means is that the client sends a dummy request to the server so that the actual request won't be revealed. The request that's used is an `OPTIONS` request, which requests general information about the server. The specification isn't clear on this point, but presumably the server could refuse to upgrade even with a mandatory upgrade. In that case, the client would have an opportunity to close the connection without any sensitive information having been transmitted.

Server Requested Upgrade

The server also has an opportunity to request TLS upgrading. The server can advertise that it supports TLS in an informational fashion merely by including the appropriate `Upgrade` header in its normal responses. However, in certain circumstances, the server may wish to enforce the use of TLS. It can do so by responding to requests that require TLS with the `426 Upgrade Required` error code, as shown in Figure 9.16.

```
HTTP/1.1 426 Upgrade Required
Upgrade: TLS/1.0, HTTP/1.1
Connection: Upgrade
(blank line)
```

Figure 9.16 A mandatory upgrade advertisement

The 426 response does not in and of itself initiate the TLS handshake. Rather, it's an error signaling that the client must use TLS to fetch this URL. When the client receives an Upgrade Required, it must use the methods described above to initiate an upgrade. Note that a mandatory upgrade advertisement is useful only in cases where only the data sent by the server is sensitive. The client has *already* transmitted *its* data over a clear channel, so if the request contained sensitive data, the damage has already been done.

HTTP Upgrade *does* allow HTTP over TLS to be negotiated over HTTP connections. In that sense, it can be considered a success. However, it has a number of problems that substantially reduce its attractiveness. First, its interaction with proxies is even worse than HTTPS's. Second, it doesn't provide for references to indicate that upgrade should be expected, thus creating security and performance problems.

Proxy Interactions

RFC 2616 [Fielding1999] defines the `Upgrade` header as a *hop-by-hop* header. This means that proxies are intended to act as if the header was intended for them. The intent is that clients should be able to negotiate better protocols with proxies (e.g., HTTP/1.1 from HTTP/1.0) even if the servers have not yet been upgraded.

However, because the client wishes to negotiate an end-to-end TLS connection with the server, the hop-by-hop nature of the `Upgrade` header presents a problem when upgrading to TLS. In order to work around this, the client needs to first use `CONNECT` to establish a tunnel to the server, just as it did with HTTPS. This is rather unfortunate because it means that clients can't upgrade to TLS at all without first creating a tunnel, which greatly reduces the value of proxies. A better approach would be to have a new end-to-end header to advertise client TLS support. Proxies would recognize this header (though not tamper with it) and if TLS were negotiated, then the proxy would go into a transparent mode.

No References

Another problem with HTTP Upgrade, is that there is no way for clients to be told in references that they should expect to be able to upgrade when they connect to the server. In fact, the specification specifically states that `https` behavior is totally unchanged: it still refers to HTTPS.

This omission has a number of unfortunate consequences. The first is the loss of even the minimal reference integrity protections provided by HTTPS. With Upgrade, the client cannot even detect that TLS should have been negotiated and wasn't. If the client wants to use TLS at all, even in an opportunistic mode, its only choice is to offer Upgrade on every connection. As we've said earlier, this means that the client must use `CONNECT` with any proxy, and thus proxies become completely ineffective.

Worse yet, unless the client is willing to accept worst-case performance behavior, the client's request is completely exposed. Recall that the client's request will contain the contents of any form submission and consider the case where the form submitted

requests a credit card number. Moreover, the client software cannot tell when requests contain sensitive information because some sites put such information directly in URLs or in browser cookies.

Thus, for maximal security the client always needs to first try an `OPTIONS` request. However, in the common case where TLS is not supported, this simply means an extra request/response round trip for no benefit. Even in cases where the upgrade is successful, HTTPS is faster than Upgrade because it does not need the dummy request. Thankfully, this overhead need only be incurred the first time the browser connects to a site if the server doesn't support security—the client can cache the fact that the server will not Upgrade.

Finally, the lack of a distinguished reference makes HTTP Upgrade vulnerable to a downgrade attack where the attacker removes the upgrade tokens, thus forcing the client and the server to negotiate HTTP. This attack is not possible with HTTPS because the client is only prepared to do HTTPS when dereferencing an `https://` URL.

All of these problems could be easily fixed with the introduction of a new URL method indicating that HTTP/TLS using Upgrade was expected. However, it would be unwise to attempt to co-opt `https:` for this purpose, since clients would then be unable to distinguish HTTP Upgrade-required servers from HTTPS servers.

9.22 Programming Issues

Chapter 8 discussed general programming issues with SSL, but each protocol is different and programming using HTTPS presents some specific problems. The rest of this chapter is devoted to covering two of these issues: implementing the HTTP `CONNECT` method discussed in Section 9.16 and supporting the large number of HTTPS connections that will be experienced by any active server.

9.23 Proxy `CONNECT`

The example programs we showed in Chapter 8 all connected directly to the server. As we discussed in Section 9.16, HTTPS clients often need to connect through a proxy rather than directly to the server. This requires that the programmer do some extra work. Figure 9.17 shows a new version of our client that connects through a proxy rather than directly to the server. Note that this client contains code only for connecting through a proxy. A real client would be able to switch-hit depending on configuration settings.

Programmer's note: Many firewalls allow the `CONNECT` method on port 443 only. We're using port 4433 for programming convenience because you can run servers on port 4433 without being root. However, you may find that you need to run your test servers on port 443 if you want to try proxying.

pclient.c

```
1   /* A proxy-capable SSL client.
2
3      This is just like sclient but it only supports proxies
4   */
5   #include <string.h>
6   #include "common.h"
7   #include "client.h"
8   #include "read_write.h"
9
10  #define PROXY "localhost"
11  #define PROXY_PORT 8080
12  #define REAL_HOST "localhost"
13  #define REAL_PORT 4433
14
15  int writestr(sock,str)
16    int sock;
17    char *str;
18    {
19      int len=strlen(str);
20      int r,wrote=0;
21
22      while(len){
23        r=write(sock,str,len);
24        if(r<=0)
25          err_exit("Write error");
26        len-=r;
27        str+=r;
28        wrote+=r;
29      }
30
31      return (wrote);
32    }
33
34  int readline(sock,buf,len)
35    int sock;
36    char *buf;
37    int len;
38    {
39      int n,r;
40      char *ptr=buf;
41
42      for(n=0;n<len;n++){
43        r=read(sock,ptr,1);
```

```
37              if(r<=0)
38                err_exit("Read error");
39              if(*ptr=='\n'){
40                *ptr=0;

41                /* Strip off the CR if it's there */
42                if(buf[n-1]=='\r'){
43                  buf[n-1]=0;
44                  n--;
45                }

46                return(n);
47              }

48              *ptr++;
49            }

50            err_exit("Buffer too short");
51          }

52          int proxy_connect(){
53            struct hostent *hp;
54            struct sockaddr_in addr;
55            int sock;
56            BIO *sbio;
57            char buf[1024];
58            char *protocol, *response_code;

59            /* Connect to the proxy, not the host */
60            if(!(hp=gethostbyname(PROXY)))
61              berr_exit("Couldn't resolve host");
62            memset(&addr,0,sizeof(addr));
63            addr.sin_addr=*(struct in_addr*)hp->h_addr_list[0];
64            addr.sin_family=AF_INET;
65            addr.sin_port=htons(PROXY_PORT);

66            if((sock=socket(AF_INET,SOCK_STREAM,IPPROTO_TCP))<0)
67              err_exit("Couldn't create socket");
68            if(connect(sock,(struct sockaddr *)&addr,sizeof(addr))<0)
69              err_exit("Couldn't connect socket");

70            /* Now that we're connected, do the proxy request */
71            sprintf(buf,"CONNECT %s:%d HTTP/1.0\r\n\r\n",REAL_HOST,REAL_PORT);
72            writestr(sock,buf);

73                                                                    (continued)
```

Figure 9.17 *(continued)*

```
74          Figure 9.17 (continued)

75              /* And read the response*/
76              if(readline(sock,buf,sizeof(buf))==0)
77                err_exit("Empty response from proxy");

78              if((protocol=strtok(buf," "))<0)
79                err_exit("Couldn't parse server response: getting protocol");
80              if(strncmp(protocol,"HTTP",4))
81                err_exit("Unrecognized protocol");
82              if((response_code=strtok(0," "))<0)
83                err_exit("Couldn't parse server response: getting response code");
84              if(strcmp(response_code,"200"))
85                err_exit("Received error from proxy server");

86              /* Look for the blank line that signals end of header*/
87              while(readline(sock,buf,sizeof(buf))>0) {
88                ;
89              }

90              return(sock);
91          }

92          int main(argc,argv)
93            int argc;
94            char **argv;
95            {
96              SSL_CTX *ctx;
97              SSL *ssl;
98              BIO *sbio;
99              int sock;

100             /* Build our SSL context*/
101             ctx=initialize_ctx(KEYFILE,PASSWORD);

102             /* Connect the TCP socket*/
103             sock=proxy_connect();

104             /* Connect the SSL socket */
105             ssl=SSL_new(ctx);
106             sbio=BIO_new_socket(sock,BIO_NOCLOSE);
107             SSL_set_bio(ssl,sbio,sbio);
108             if(SSL_connect(ssl)<=0)
109               berr_exit("SSL connect error");
110             check_cert_chain(ssl,HOST);
```

```
111             /* read and write */
112             read_write(ssl,sock);

113             destroy_ctx(ctx);
114         }
```

pclient.c

Figure 9.17 A proxy-capable client

A write function

12–27 Our proxy-capable client needs to be able to send the CONNECT request to the proxy server. We encapsulate the writing code in the `writestr()` function. This function simply takes an input string of arbitrary length and writes it to the specified socket.

A read function

28–51 We also need to be able to read the proxy server's response. The response should be a series of HTTP header lines separated by the carriage return (`0x0d`) and line feed (`0x0a`). The `readline()` function simply reads one line off the network into the buffer. This function is suitable only for demonstration purposes. It reads one byte from the network at a time, which is very inefficient. Moreover, it fails catastrophically if the buffer is too short to contain the line.

Connect to the proxy

52–69 The `proxy_connect()` function is intended to replace the `tcp_connect()` function that we used in Chapter 8. The beginning of the function simply replicates the connecting code that was in `tcp_connect()`. However, this time instead of connecting to the server, we connect to the proxy.

Write request

70-72 The first thing that the client needs to do is to construct the CONNECT request, containing the real host and port that we wish the proxy to connect to. The `"\r\n\r\n"` string at the end of the string signals the end of the HTTP header with a blank line. Then we simply use `writestr()` to transmit the request to the proxy.

Read response

73–88 Once the request has been transmitted to the proxy, we need to read back the response to make sure that the proxy successfully connected. We parse the first line of the response (the status line) to ensure that the protocol is HTTP and that the status code is 200. The rest of the header we simply read and throw away. Again, the end of the proxy's response is signaled by a blank line. Note that we must read all the header lines so that when the SSL engine looks for the ServerHello it doesn't read any of the proxy response.

main()

90–112 Our `main()` function is identical to the `main()` in `sclient.c` except that it calls `proxy_connect()` rather than `connect()` in line 101. Once the tunnel is established, the client can completely ignore the existence of the proxy.

Note that in order to use a proxy the client had to read and write directly to the TCP socket before doing the SSL handshake. This is possible with OpenSSL because the API requires that the programmer do the `connect()` first before starting the SSL session. However, if the `connect()` and the SSL handshake are integrated, as they are in PureTLS, this would not be possible.

> This problem is on the PureTLS known bug list, so with any luck the author will have fixed this problem by the time you read this.

9.24 Handling Multiple Clients

Any real server needs to be able to simultaneously handle multiple clients. This means being able to multiplex data reads and writes between those clients, serving clients that are ready to be served and ignoring those that aren't. Moreover, it needs to do so in a timely fashion, without serving only one client and starving the others. This is especially problematic in HTTPS where any given client may be requesting data on a number of separate connections.

It's possible to do this using the `select()` based I/O multiplexing technique we used in Chapter 8, but it's not very convenient. It turns out to be difficult to write programs this way, partly because it requires explicitly carrying a lot of state around.

To see this, consider the problem of reading the HTTP request. The server reads the data, but perhaps only part of the request is available. The server can't wait for the rest of it because it must serve other clients. Thus, it must store the data it has already read and finish reading the rest of the request when the client eventually sends it. This course requires creating an explicit context object for that client and somehow associating it with the client socket. This isn't especially difficult, but it is inconvenient.

The second problem is that the server needs to be very careful not to allocate too much time to a single operation. Imagine that a request requires performing some operation—such as a database query—that takes 5 seconds. Obviously, the requesting client must wait for the operation to complete, but equally obviously the server can't starve all the other clients because it's serving this particular client. Instead, it must break out of the operation and service other clients and then resume the operation. Again, this means carrying the operation state around explicitly rather than implicitly.

Moreover, the programmer must constantly be paying attention to how long various actions are expected to take, lest the server inadvertently fail to service other clients while performing a time-consuming operation. This can be a particular problem when using third-party libraries because they may not include facilities for breaking out of their API calls.

Multiprocess Servers

Unsurprisingly, most programmers choose a different approach toward serving multiple clients: Each client connection is assigned its own thread of control. On UNIX systems, this usually means a process and on Windows (or in Java) it means a thread. The scheduler is then responsible for distributing CPU time between the server processes/threads serving each client and can ensure that each client gets its fair share.

The traditional design for such a server is to have a single process/thread that listens for client connections. When it receives a new connection, it spawns off a new process/thread to handle that connection. Figure 9.18 shows pseudo-code for such a server.

The main server process simply loops forever. On each iteration of the loop, it calls `accept()`. If a new client connection has come in, `accept()` returns a socket corresponding to that new connection. Otherwise, the process is simply put to sleep until a new client comes in. Once a new connection is created, the server creates a new process to handle it. On UNIX, this is done using the `fork()` system call. On Windows, we'd be creating a new thread using `CreateThread()`. The parent process returns to the top of the loop and calls `accept()` again.

```
server_process() {
  server_socket=create_socket();

  for(;;){
    client=accept(server_socket);
    if(pid=fork()){
      /* Parent process */
      continue;
    }
    else {
      /* Child process */
      serve_request(client);
      exit(0);
    }
}
```

Figure 9.18 A simple server

On UNIX, we'd actually need the parent process to `close()` the socket connected to the client in order to free up server resources. Any given process is allowed to have only a finite number of file descriptors open at any given time, and if we don't close the socket, the parent process will quickly run out. This has no effect on the socket in the child process.

The child process or thread simply serves the client. Because the scheduler automatically interrupts it to service other server control threads, the child can simply concentrate on serving its one client. In particular, it can safely use blocking I/O or perform

Ex.1024
Askeladden, L.L.C. / Page 407 of 788

time consuming operations without worrying about breaking out of those operations to serve other clients.

Multiprocess Servers Using SSL

When adding SSL to one of these multiprocess servers, we must be careful. It's tempting to simply replace all the socket calls with calls to SSL sockets. This will not work. In particular, the SSL handshake must be executed in the child process rather than in the parent process. Otherwise, the server will be able to execute only one SSL handshake at once. The handshake represents a substantial bottleneck, especially when the round-trip time between client and server is significant. This bottleneck is particularly troublesome with HTTP because the number of connections is so large.

In order to avoid this bottleneck, the server needs to perform the TCP `accept()` in the parent process but the SSL handshake in the child process. This is fairly straightforward to do with OpenSSL, because `SSL_handshake()` is its own call. Figure 9.19 shows a variant of the main loop in our OpenSSL server program that behaves in this way.

mserver.c

```
24          while(1){
25            if((s=accept(sock,0,0))<0)
26              err_exit("Problem accepting");

27            if(pid=fork()){
28              close(s);
29            }
30            else {
31              sbio=BIO_new_socket(s,BIO_NOCLOSE);
32              ssl=SSL_new(ctx);
33              SSL_set_bio(ssl,sbio,sbio);

34              if((r=SSL_accept(ssl)<=0))
35                berr_exit("SSL accept error");

36              echo(ssl,s);
37            }
38          }
```

mserver.c

Figure 9.19 A simple OpenSSL-based multiprocess server

The only really new code here is the call to `fork()` in line 27. This creates a new process (on UNIX). That process is able to run the SSL handshake independently. The code would be analogous on Windows but the call would be to `CreateThread()` instead of to `fork()`.

Because PureTLS emulates Java's socket API, the `accept()` and SSL handshake all happen during the same API call. Thus, there is no way to arrange to execute the handshake in a child thread. This is an omission in PureTLS and a better approach would be to allow the programmer to tell `accept()` to just create the socket and then do the SSL handshake separately.

SSL Session Caching

Whenever we have a multiprocess or multithreaded server, we need to make accommodations in our session caching strategy. Session data is unique because it must be shared between all the different control threads and can also be updated by any process/thread. Other data, such as the SSL context data, may be shared by all processes/threads, but isn't constantly updated. Session cache data must be updated in at least two situations. First, whenever a new session is created, it must be added to the cache. Second, if a session is to be marked non-resumable (e.g., because an alert was received) it must be removed from the session cache. We need to ensure proper concurrency control so that simultaneous updates can't lead to cache corruption.

If we're using threads rather than processes, all of this is fairly simple. Because threads all share memory, we can simply use an in-memory representation for the session cache. However, it's still necessary to lock the cache data while reading and writing. The details of how to do this locking depend on the system and threading package being used. Java provides probably the simplest methods for doing this. Methods can be marked `synchronized`, in which case calls to those methods will be serialized. Figure 9.20 shows a fragment of PureTLS's internal session caching code, complete with synchronization.

SSLContext.java

```
438         protected synchronized void storeSession(String key,SSLSessionData sd){
439           SSLDebug.debug(SSLDebug.DEBUG_STATE,"Storing session under key"+key);
440           session_cache.put(key,(Object)sd);
441         }

442         protected synchronized SSLSessionData findSession(String key){
443           SSLDebug.debug(SSLDebug.DEBUG_STATE,"Trying to recover session using key"+key);
444           Object obj=session_cache.get(key);

445           if(obj==null)
446            return null;

447           return (SSLSessionData)obj;
448         }

449         protected synchronized void destroySession(String sessionLookupKey){
450           session_cache.remove(sessionLookupKey);
451         }
```

SSLContext.java

Figure 9.20 PureTLS synchronized session caching code

PureTLS stores the session data using a hash table (`session_cache`) which is an instance variable of the `SSLContext` class. This code fragment shows the minimal required three methods for accessing the cache: `storeSession()` creates an entry. `findSession()` looks up an entry and `destroySession()` deletes an entry from the cache.

Session Caching with Multiprocess Servers

If your server uses multiple processes rather than multiple threads, the situation becomes dramatically more difficult. Because processes do not share memory, it's not possible to simply have a pointer to some common structure that is accessed by all the processes. Instead, the server needs to rely on operating system services to carry the session data around.

There are a number of different techniques that can be used for this job. Most toolkits do not provide support for sharing session data between processes, so the problem usually needs to be solved on a server-by-server basis. One approach is to have a single *session server*. The session server stores all of its session data in memory. The SSL servers access the session server via *interprocess communication* mechanisms, typically some flavor of sockets. The major drawback to this approach is that it requires the creation of some entirely different server program to do this job, as well as the design of the SSL server to session server communications protocol. It can also be difficult to ensure access control so that only authorized server processes can obtain access.

Another approach is to use *shared memory*. Many operating systems provide some method of allowing processes to share memory. Once the shared memory segment is allocated, data can be accessed as if it were ordinary memory. Unfortunately, this technique isn't as useful as it sounds because there's no good way to allocate out of the shared memory pool, so the processes need to allocate a single large segment and then statically address inside of it. Worse yet, shared memory access is not easily portable.

The most commonly used approach is to simply store the data on a file on the disk. Then, each server process can open that file and read and write out of it. Standard file locking routines such as `flock()` can be used to provide synchronization and concurrency control. One might think that storing the data to disk would be dramatically slower than shared memory, but remember that most operating systems have disk caches and this data would likely be placed in such a cache.

A variant of this approach is to use one of the simple UNIX key-value databases (DBM and friends) rather than a flat file. This allows the programmer to simply create new session records and delete them without worrying about placing them in the file. If such a library is used, care must be taken to flush the library buffers after each write, because data stored in the buffers has not been written to disk.

OpenSSL does not provide any support for this sort of session caching at all, but it does provide hooks for programmers to use their own session caching code. ApacheSSL (based on OpenSSL) uses the session server approach. mod_ssl (also based on OpenSSL) can support either the disk-based database approach or a shared memory approach. The SPYRUS TLSGold toolkit uses the file-based approach without any database support. The code for all of these approaches is too complicated to show here, but Appendix A shows the mod_ssl session caching code.

Advanced Server Configurations

Early SSL servers operated as described earlier in this section. However, it was observed that the cost of the `fork()` on UNIX was very substantial. When Netscape introduced their server, they added a feature that *pre-forked* a number of child servers and arranged to distribute clients to those servers rather than creating a new process for each client. Most UNIX-based servers, including the popular free Apache server, now do this.

Later Netscape servers also operated multiple threads within the same server process. However, they continued to use multiple server processes as well. Thus, although these new configurations represented an improvement in HTTP server behavior, they still required that multiple processes be able to share session data.

9.25 Summary

SSL was designed for use with HTTP and so we'd expect them to work relatively well together. In general they do, as evidenced by the ubiquity of HTTPS clients and servers. However, when we examine the interaction in detail, we can see a number of situations in which clumsy workarounds have had to be used to compensate for deficiencies in SSL and HTTPS.

HTTPS is the dominant approach to securing HTTP. HTTPS is extremely simple to implement, which no doubt accounts for its rapid deployment. Despite a number of problems, it has yet to be replaced in any substantial way.

HTTPS uses hostnames for reference integrity. The HTTPS URLs contain the expected hostname for the server. The client is expected to check the hostname against the name in the server's certificate. No provision is made for certificates with names that don't match the host name.

Using HTTPS breaks proxies. When HTTPS is used, proxy caching is totally ineffective. Moreover, proxies require special support for the `CONNECT` method to be able to pass HTTPS at all.

Servers have trouble requesting different handshakes for different resources. Because the SSL handshake takes place before the server sees the request, the server has no way to condition its handshake on the resource to be requested. This can lead to expensive renegotiation.

HTTP Upgrade fixes some problems and introduces others. HTTP Upgrade helps reduce port consumption and works properly with virtual hosts. Unfortunately, its interaction with proxies is even worse than in HTTPS and it provides much weaker reference integrity.

Session caching is critical. Because so many Web page fetches require multiple HTTPS connections, SSL session caching is extremely important. Because Web servers often run as multiple processes, servers must share cache data between processes, creating implementation problems.

10

SMTP over TLS

10.1 Introduction

The second example protocol we'll consider is the *Simple Mail Transfer Protocol* (SMTP), described in RFC 821 [Postel1982]. SMTP is the basic protocol used to transport Internet electronic mail (e-mail). E-Mail is arguably the single most important Internet application and so securing it is an especially high priority. Thus, the standard for SMTP over TLS (RFC 2487 [Hoffman1999a]) was published essentially at the same time as the standard for TLS, RFC 2246.

As with the chapter on HTTP, we start by describing Internet mail and SMTP. We then consider the approach used by RFC 2487. As we'll see, it's very difficult to provide even basic security services for mail using TLS. This is not the fault of the author of RFC 2487. In fact, SMTP is an interesting case precisely because its security requirements are so poorly matched to the services that TLS is able to provide. Finally, we briefly discuss the programming issues involved in using SMTP with TLS.

10.2 Internet Mail Security

The process of sending e-mail should be familiar to most Internet users. The user has a program that permits him to enter the recipient and compose a message. When he's finished composing, he tells the program to send it. Usually, the sender has a local *mail server* to which he sends the mail. The mail server arranges for delivery to the recipient. The sender's security requirement is for confidentiality: He wants to ensure that the message goes to the intended recipient and that the contents are readable only by that recipient. He might also wish to know that the message arrived at the recipient undamaged.

Similarly, the recipient has a local mail server. The sender's mail server delivers the mail to the recipient's mail server. The recipient's mail program then obtains the mail from the mail server and displays it to the recipient. The recipient's security requirement is for message authentication and message integrity; he wants to know who sent the message and know that it has not been tampered with in transit.

Ex.1024
Askeladden, L.L.C. / Page 413 of 788

Basic Technologies

In order to understand how electronic mail works—and thus any potential security solution—it's important to understand the basic technologies in the e-mail infrastructure. The major three technologies that we need to consider are *SMTP*, the *RFC 822* and *MIME* message formats, and *e-mail addresses*.

SMTP

SMTP is the basic transport protocol for e-mail. Its job is to transport messages between senders (either client programs or mail servers) and receiving mail servers. In the early days of the net there were other protocols that did this job but currently SMTP is the only major mail delivery protocol.

RFC 822 and MIME

SMTP just provides the transport for messages. The actual specification for messages appeared in RFC 822[Crocker1982]. However, RFC 822 only specified formats for ASCII text messages and eventually the *Multipurpose Internet Mail Extensions* (MIME) [Freed1996a, Freed1996b, Moore1996, Freed1996c, Freed1996d] were added to support other forms of content.

E-mail addresses

In order to send mail to someone, it's necessary not only to know the recipient's name but also their e-mail address. The address not only tells the sender which server to connect to but also tells the receiving server what to do when it gets the mail.

Practical Considerations

We also need to be concerned with some features of the mail environment that are not obvious from the description in the previous paragraphs. Four of these features turn out to have particular security relevance: *mail relaying*, *virtual hosting*, *MX records*, and *client mail access*.

Mail Relaying

It's fairly frequent for mail not to travel directly between two servers but rather to go through a series of *mail relays* on the way. The obvious security problem is maintaining security through these relays.

Virtual Hosts

As with Web servers, it's quite common for ISPs to maintain mail servers for people's personal domains or small corporate domains. This allows users to each have their own domain and e-mail addresses without even a direct connection to the Internet. The problem we face here is the same as that with HTTPS: making sure that the receiving server provides the appropriate certificate to the sender.

MX Records

Using mail relaying, it's possible to send mail to machines that are not currently connected to the Internet. The sender simply connects to the correct relay and trusts the relay to deliver the mail. *Mail Exchanger* (MX) records serve to identify which relay to connect to.

Client Mail Access

When most mail was read on UNIX machines, the mail server simply placed the mail on the disk and the user's client would read it off the disk. In modern networking environments, most mail clients are on PCs without access to the filesystem of the server. Those clients use a variety of network protocols to access the mail server and retrieve their mail.

Security Considerations

Once we understand the Internet mail environment, we can consider how to offer the appropriate security services. As before, we need to consider the important problems discussed in Chapter 7: *protocol selection*, *client authentication*, *reference integrity*, and *connection semantics*.

10.3 Internet Messaging Overview

Any Internet mail system consists of at least two types of entities: clients (known as *mail user agents* (MUAs)) and servers (*mail transport agents* (MTAs)). The mail client is the program the user interacts with. It displays his mail and permits him to compose and send new mail. However, most clients have nothing to do with either mail receipt or delivery. These jobs are accomplished by the MTAs.

As we said earlier, mail does not go directly from sender to recipient. Instead, it passes through a series of servers along the way. Figure 10.1 shows a simple mail system delivering a single piece of e-mail. The sending client on the left is sending a message to the receiving client on the right. The user composes a message and then his mail client transmits it using SMTP to his local mail server. The local mail server then transmits the message (again using SMTP) to the recipient's local mail server.

It's actually quite common for the mail client to transmit the message to the first hop server using something other than SMTP. On UNIX systems, the mail client often invokes the MTA (usually Sendmail) as a program. Sendmail then uses SMTP to deliver the message to the next hop server. When the MTA is Microsoft Exchange, clients often use *Mail Application Programming Interface* (MAPI) to talk to the MTA.

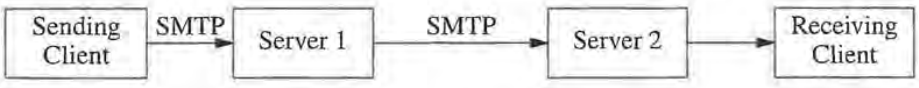


Figure 10.1 A simple mail system

The final link, from the recipient's local mail server to the recipient, is *not* accomplished via SMTP. A large number of protocols exist for reading mail, including *Post Office Protocol* (POP) [Myers1996] and *Internet Mail Access Protocol* (IMAP), defined in [Crispin1996]. On many UNIX systems, the mail server resides on the same machine that the recipient is using and the recipient can simply read the mail messages off of the disk. It is even possible to read one's mail using HTTP. In any case, this link does not use SMTP.

Consider what happens if the recipient's mail server is temporarily offline. The sender's local mail server will simply hold on to the mail, retrying every so often, until it can deliver it or the message times out. Similarly, the recipient need not be online at all for mail to be delivered to his local server. The local server will simply store the mail and wait for the recipient to pick it up. Thus, Internet mail systems handle disconnection gracefully in a way that HTTP does not. This is obviously necessary in order to accommodate users who do not have permanent Internet connections. This style of messaging is what's referred to as *store-and-forward.*

10.4 SMTP

As with HTTP, SMTP is conceptually simple but somewhat complicated in practice. The sending agent makes a connection to the receiving agent on TCP port 25. The sending and receiving agents then engage in a series of requests and responses to establish sending and receiving parameters, including the sender and the recipient. Finally, once all the parameters are established, the sending agent transmits the message to the server.

All communication is done in lockstep. The sending agent sends a command and the receiving agent sends a reply. Commands consist of a single line terminated by a CRLF. Responses consist of a three-digit number (the *reply code*) followed by some explanatory text. Responses can take up multiple lines. Each line except the last has a hyphen after the reply code to indicate that more lines are coming. Figure 10.2 shows a simple SMTP example.

```
New TCP connection: romeo(3094) <-> speedy(25)
1 949464306.6034 (0.0966) S>C
data: 101 bytes
-----------------------------------
220 speedy.rtfm.com ESMTP SendWhale 8.9.1/8.6.4 ready at 28.8K Tue, 1 ⏎
Feb 2000 20:05:03 -0800 (PST)
-----------------------------------

2 949464306.6036 (0.0969) C>S
data: 21 bytes
-----------------------------------
EHLO romeo.rtfm.com
-----------------------------------
```

```
3 949464306.6089 (0.1022) S>C
data: 173 bytes
-----------------------------------
250-speedy.rtfm.com Hello romeo.rtfm.com [216.98.239.227], pleased to ↵
meet you
250-EXPN
250-VERB
250-8BITMIME
250-SIZE
250-DSN
250-ONEX
250-ETRN
250-XUSR
250 HELP
-----------------------------------

4 949464306.6094 (0.1026) C>S
data: 41 bytes
-----------------------------------
MAIL From:<ekr@romeo.rtfm.com> SIZE=224
-----------------------------------

5 949464306.8004 (0.2937) S>C
data: 39 bytes
-----------------------------------
250 <ekr@romeo.rtfm.com>... Sender ok
-----------------------------------

6 949464306.8005 (0.2938) C>S
data: 24 bytes
-----------------------------------
RCPT To:<ekr@rtfm.com>
-----------------------------------

7 949464306.8302 (0.3234) S>C
data: 36 bytes
-----------------------------------
250 <ekr@rtfm.com>... Recipient ok
-----------------------------------

8 949464306.8303 (0.3236) C>S
data: 6 bytes
-----------------------------------
DATA
-----------------------------------
```

(continued)

Figure 10.2 *(continued)*

```
9 949464306.8570 (0.3502) S>C
data: 50 bytes
-----------------------------------
354 Enter mail, end with "." on a line by itself
-----------------------------------

10 949464306.8578 (0.3510) C>S
data: 445 bytes
-----------------------------------
Received: from romeo.rtfm.com (localhost [127.0.0.1]) by romeo.rtfm.com ↵
(8.9.3/8.6.4) with ESMTP id UAA46227 for <ekr@rtfm.com>; Tue, 1 Feb ↵
2000 20:05:06 -0800 (PST)
Message-Id: <200002020405.UAA46227@romeo.rtfm.com>
To: ekr@rtfm.com
Subject: Test message
Mime-Version: 1.0 (generated by tm-edit 7.108)
Content-Type: text/plain; charset=US-ASCII
Date: Tue, 01 Feb 2000 20:05:06 -0800
From: Eric Rescorla <ekr@rtfm.com>

This is a test
-----------------------------------

11 949464306.9991 (0.4924) C>S
data: 3 bytes
-----------------------------------
.
-----------------------------------

12 949464307.0654 (0.5587) S>C
data: 44 bytes
-----------------------------------
250 UAA09843 Message accepted for delivery
-----------------------------------

13 949464307.0864 (0.5797) C>S
data: 6 bytes
-----------------------------------
QUIT
-----------------------------------
```

```
14 949464307.1127 (0.6059) S>C
data: 40 bytes
-----------------------------------
221 speedy.rtfm.com closing connection
-----------------------------------

Client FIN
Server FIN
```

Figure 10.2 A simple SMTP example

When the sender first connects to the receiver, the receiver sends a banner with identifying information, usually including a hostname and a software version number. In this case, segment 1 shows that the server is Sendmail 8.9.1.

The first command that the sender sends should be either `HELO` or `EHLO`. Traditional SMTP as described in RFC 821 used the `HELO` command. RFC 1425 introduced a number of extensions to SMTP and the `EHLO` command indicates that a sender complies with that specification. Either command should contain the sender's hostname (as it believes it to be.) In this example, the sender's hostname is `romeo.rtfm.com`.

The receiver responds to the `EHLO` with its own hostname and a list of the extensions it supports. Thus, this server supports a number of extensions, including `EXPN`, `VERB`, and `8BITMIME`.

The rest of the connection proceeds in a similar fashion. The sender identifies the user who sent the message using the `MAIL` command and the recipient of the message using the `RCPT` command. In both cases, the receiving server responds with the `250` status code to indicate that things are fine.

Once the sender and receiver have been accepted, the sending agent can send the actual message using the `DATA` command. The receiver responds with a `354` indicating that the sender should go ahead and send the message. The actual message appears in segment 10. SMTP uses a single line containing only a period (i.e., .CRLF) to indicate the end of data, and we see that in segment 11. In segment 12, the receiver indicates that it's accepted the message and it's ready to deliver it.

At this point, the sending agent could repeat the `MAIL, RCPT, DATA` sequence to send another message. In this trace, however, there is no other message to send, so the sending agent uses `QUIT` to close out the connection.

10.5 RFC 822 and MIME

RFC 822 style messages are very simple. They consist of a number of header lines followed by a message body. Each header line is simply the name of the header, followed by a colon and the value, like so:

```
From: ekr@rtfm.com
```

The message body contents are largely freeform. RFC 822 restricted the message body data to ASCII characters, but MIME opened it up to binary data. Figure 10.3 shows a simple example mail message sent from the author to himself. Note that the two long `Received` header lines have been broken up into multiple lines. If a header line begins with a whitespace character it is automatically *folded* onto the end of the previous line.

```
From ekr@Network-Alchemy.COM  Tue Feb  8 09:25:10 2000
Received: from Hydrogen.Network-Alchemy.COM (Hydrogen.Network-Alchemy.COM
 [199.46.17.130]) by speedy.rtfm.com (8.9.1/8.6.4) with ESMTP id JAA04954
 for <ekr@rtfm.com>; Tue, 8 Feb 2000 09:25:09 -0800 (PST)
Received: (from ekr@localhost)  by Hydrogen.Network-Alchemy.COM
 (8.8.7/8.8.8) id JAA05750 for ekr@rtfm.com; Tue, 8 Feb 2000 09:23:52
 -0800 (PST)  (envelope-from ekr)
Date: Tue, 8 Feb 2000 09:23:52 -0800 (PST)
From: Eric Rescorla <ekr@Network-Alchemy.COM>
Message-Id: <200002081723.JAA05750@Hydrogen.Network-Alchemy.COM>
To: ekr@rtfm.com
Subject: Test message

How about this?
```

Figure 10.3 An e-mail message

This format should look very familiar from our discussion of HTTP requests and responses. This isn't surprising because HTTP messages were explicitly modeled after e-mail messages. Look at the `Received` lines in this message. The SMTP servers along the mail path add new header lines of the message. In fact, only the last two header lines `To` and `Subject` were created by the client's MUA. As each server processes the data, it may add its own header lines to the front of the message.

Thus, when the client's MUA passed the message on to his local MTA at `hydrogen.network-alchemy.com`, the server automatically added the `Date` field to indicate when it sent the message, the `From` field to indicate who the sender was, and the `Message-Id` field to provide "tracking number" for this message. Note that the handling of the `From` field is inconsistent; some mail clients will generate it themselves and it will usually not be tampered with by the server.

The `From` line at the top of the message and the first `Received` header were added by the recipient's mail server at `speedy.rtfm.com`. This line is only added by certain UNIX mailers as part of the UNIX "mbox" storage format.

Received Lines

The `Received` lines provide a record of the path that the message took through the mail system. Each server adds its own `Received` line to the front of the message when it processes it.

Examining these headers, we can see that the original message was sent by the user `ekr` on the machine `hydrogen.network-alchemy.com`. The user directly invoked the mail server rather than connecting to it via SMTP. (Actually, the user used the `Mail` command, which invokes Sendmail.) `hydrogen.network-alchemy.com` connected directly to `speedy.rtfm.com` (the mail server for the `rtfm.com` domain) and delivered the message. `speedy` wrote the message on disk, ready for the user to read.

Sender Identity

As you can see, there are two indications in the message of who the sender was. The first is the `From` line at the top of the message. Note that this is not a header line; it has no colon. It's added by Sendmail before writing the data to disk. Other mailers may use a different format. This can be thought of as indicating the *transport identity*. It's the identity that the sending SMTP agent used in its `MAIL` command. The `From` line in the message should be thought of as the *message identity*. It's the identity that the sender of the message placed in the message. SMTP doesn't care about this identity at all.

In this case, the sending agent was invoked by the user as a program rather than over the network and so it is able to verify the user's identity and generate the appropriate `MAIL` command. By contrast, the user can write a new `From` header and have it simply carried by the sending agent. Thus, the `From` header is untrustworthy. As we'll see later, the contents of the `MAIL` command are untrustworthy as well.

10.6 E-Mail Addresses

The form of identification used by Internet mail is the e-mail address. It's used both to describe how to reach someone (a reference) and as an indication of sender identity. Thus, if we're to understand how to do either of these things securely, we need to be familiar with how addressing works.

Internet mail addressing can be fiendishly complex. In particular, addresses can contain all sorts of routing information telling the servers along the way which server to transfer the mail to next. Before the DNS and MX records, addresses were often specified with routing information contained in "bang paths," as in the following:

```
foo!bar!ekr
```

The above address means to send the mail to host `foo`. `foo` should send it to user `ekr` on host `bar`. Luckily, in the modern Internet such things have mostly vanished and addresses can be described fairly simply as follows:

```
local-part@domain
```

The domain can be thought of as simply the DNS domain name of the target machine. This isn't strictly correct as we'll see when we discuss mail relaying and MX records in Section 10.9, but it's a good approximation.

The local-part can be loosely thought of as the mailbox. Again, this isn't strictly correct, but it is a useful approximation. It certainly may not correspond to any actual user. For instance, it might correspond to a mailing list run off the server, or an alias to another user. For instance, machines are supposed to have a `postmaster` address but they usually do not have a `postmaster` user but rather alias the address to an administrator's account.

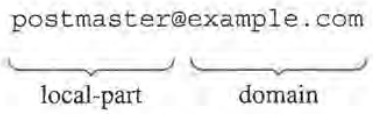


Figure 10.4 An e-mail address

Figure 10.4 shows an example e-mail address with pointers to the local-part and the domain. It refers to the mailbox `postmaster` at the domain `example.com`.

10.7 Mail Relaying

In the examples we've discussed so far, messages were sent directly from the sender's local mail server to the recipient's local mail server. However, in many cases there are other servers in between. This often occurs because of network segmentation, particularly from firewalls.

Recall from Chapter 7 that many firewalls only permit traffic to cross the firewall by way of proxy. This means that the firewall needs to have some sort of SMTP agent proxying the traffic. Unlike an HTTP proxy, an SMTP proxy is two-way. It receives messages from the outside world and sends them to the internal SMTP server as well as transferring messages from the inside world to outside servers. Thus, in an environment where both sender and receiver are behind firewalls, any message will pass through at least four SMTP servers: the sender's local server, the sender's firewall, the receiver's firewall, and the receiver's local server.

The security problem that we face whenever we have mail relaying is to ensure that malicious relays cannot read the mail or alter or forge mail. With HTTPS we solved this problem by simply cutting the relays out of the loop. The `CONNECT` method allowed us to make an end-to-end connection between client and server. Because SMTP is store-and-forward, this approach is infeasible. If we are to use SSL, we have to arrange that we connect only through trusted relays.

Organizational Servers

It's also quite common for companies to have multiple *organizational servers* so that different subunits of the same company may be hosted on different servers. For instance, accounting might be on one server and engineering might be on another. The engineering and accounting servers might even run different operating systems in order to accommodate their different needs, with the engineers on UNIX with Sendmail and the accountants on NT with Exchange.

Another reason to have organizational servers is for load sharing purposes. E-Mail often takes up quite a bit of disk space and a big organization may choose to have a number of machines with smaller disks rather than one machine with an enormous disk. This approach has the added benefit that it avoids having a single point of failure for the entire corporation.

However, it's often desirable to hide this complexity so that users can have a single mailbox that can be addressed from whichever organizational server they're actually hosted on. To implement this, all mail must go through some central server which dispatches it to the organizational servers. Figure 10.5 shows such a setup.

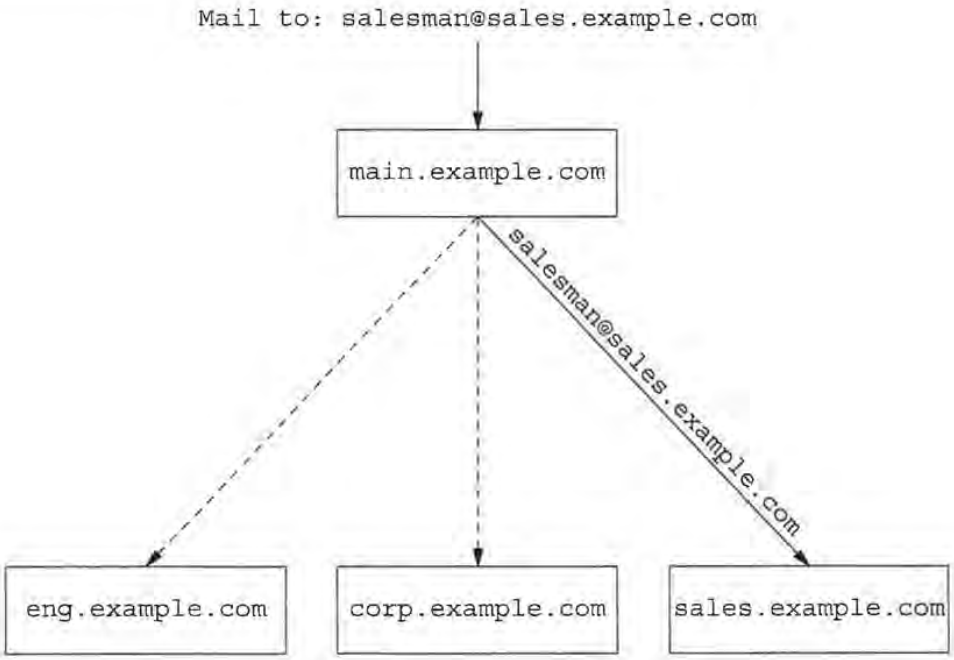


Figure 10.5 Organizational servers

In Figure 10.5, we have a single central server for the company ExampleCo (domain `example.com`). ExampleCo has three subdivisions, engineering, corporate management, and sales, each with its own SMTP server. When a message comes in addressed to `salesman`, it first goes to `main.example.com`, which figures out that it should relay it to `sales.example.com`.

One drawback of having all organizational servers have the same domain is the requirement to synchronize usernames between the servers. In order to avoid this, some organizations have organizational servers that serve separate domains. A typical e-mail address in such a system would be `foo@corp.example.com`. Such organizations might still choose to have a single mail server receive all mail from the outside purely for administrative or firewall reasons.

Smart Hosts

In a situation where organizations have their own mail servers, it's often desirable to have a single server that takes care of all transmissions to the outside world. The organizational servers are simply configured to relay all messages to that main server. This main server is often called a *smart host*.

The advantage of having a smart host is that all complicated mail configuration can be done at that single management point. The organizational mail servers are trivial to configure because routing is very simple: they always route to the smart host. In the modern Internet, mail routing is generally fairly simple, so this isn't as much of a problem as it used to be. However, smart hosts are still quite common because they allow a single administrative point for mail delivery.

Figure 10.6 shows the same server structure that we showed in Figure 10.5, but this time sending a message from a user of `eng.example.com` to the e-mail address `recipient@rtfm.com`. The machine `main.example.com` is the smart host, so `eng` sends the message to `main`, which arranges to send it to `rtfm.com`.

Open Relays

Before the rise of the Internet, it was extremely common for organizations not to have a full-time connection to other parts of the network. Rather, they might have an occasional connection to one or two other mail servers. These servers would accept mail for the network and relay their mail to the outside world. As a matter of tradition and courtesy, people would simply configure their mailers to accept mail traffic from any host to any host and do their best to deliver it. This configuration is called an *open relay*.

This sort of configuration has become much less common (though not yet nonexistent, especially outside the U.S.) but many mail agents are still configured as open relays. This has lately become an issue due to the rise of *spam*—unsolicited mass commercial e-mail—the e-mail equivalent of junk mail.

A common tactic to block spam is to simply refuse SMTP connections from the spammer's network. A number of organizations maintain lists of known spammers and you can configure your mailer to block mailers on those lists. In response, spammers send mail through innocent open relays. Administrators thus face the difficult choice of

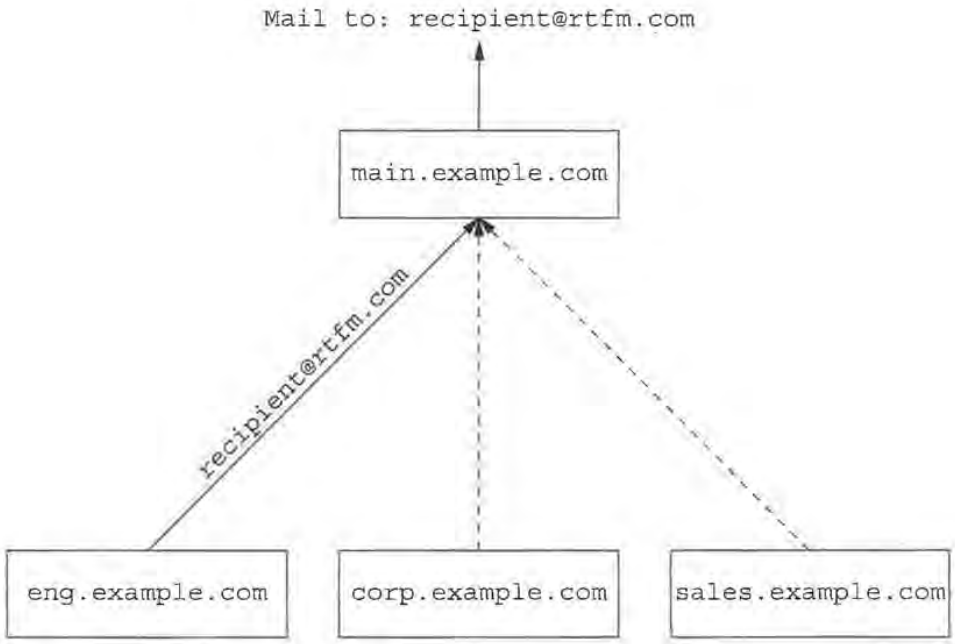


Figure 10.6 A smart host

blocking mail from legitimate machines or getting spam relayed through those machines. Those who choose to block can subscribe to lists of open relays in the same way as they subscribe to lists of known spammers.

The security problem here is to permit relaying for authorized machines—even if they are not part of your network—but to forbid relaying for unauthorized machines. SSL allows us to provide strong authentication for senders who want to use our server as a relay and only relay mail for senders who authenticate as an authorized machine.

10.8 Virtual Hosts

As indicated earlier, it's quite common for customers of ISPs to wish to have their own e-mail domain. Quite frequently, these customers are connected via dialup lines and therefore do not have a full-time connection to the Internet. In such cases, the ISP receives mail for them and the customers arrange to read the mail remotely. Because the SMTP `RCPT` command identifies the domain name of the mail recipient, supporting virtual hosts with SMTP is easy. However, as with HTTP, we need to be concerned that it works equally well when we add security to the mix.

10.9 MX Records

So far we've avoided the question of how one server knows which server to connect to in order to deliver mail to a certain recipient. Mail routing is conceptually simple but there are some subtleties we need to consider.

In general, mail servers consist of one of two types, dumb and smart. A dumb server is able to recognize mail destined for one of its mailboxes and deliver that. Any other kind of message is simply handed off to a smart host for delivery. Thus, dumb servers don't have a routing problem. They simply need to be configured to know what the smart host is.

One might think that smart host behavior would be fairly simple as well: look up the domain portion of the e-mail address and connect to the given server. However, consider the case of virtual hosts. For concreteness, let's assume that we have a single ISP server, `mail.exampleisp.com` and it's hosting the virtual domain `examplecustomer.com`. One approach would be to have a DNS A record pointing from `examplecustomer.com` to `mail.exampleisp.com`'s IP address. However, this would imply that other services such as Telnet were available to `examplecustomer.com` when in fact that machine isn't really on the net at all.

A better solution is to use a record specially designed for mail, the MX (*Mail Exchanger*) record. An MX record tells a sender that the target of the MX record is the mail relay for the requested domain. We can also have multiple MX records, as shown in Figure 10.7.

```
microsoft.com   preference = 10, mail exchanger = mail2.microsoft.com
microsoft.com   preference = 10, mail exchanger = mail3.microsoft.com
microsoft.com   preference = 10, mail exchanger = mail4.microsoft.com
microsoft.com   preference = 10, mail exchanger = mail5.microsoft.com
microsoft.com   preference = 10, mail exchanger = mail1.microsoft.com
```

Figure 10.7 MX records for `microsoft.com`

Figure 10.7 shows the MX records for the domain `microsoft.com` as fetched by `nslookup`. There are five records for machines `mail1` through `mail5`. Note that each record has a preference value. This can be used to indicate that the sender should try the mail servers in order. Thus, MX records can be used to preferentially direct traffic along certain links. This would not be possible if we'd used A records because they do not have preference values.

Unfortunately, this sort of mail relaying presents a security challenge. Because we can't trust the DNS we can't know that the MX records we're getting haven't been forged. Thus, just as with CNAMEs, we need to check the recipient's domain (e.g., `microsoft.com`) rather than the name of the mail exchanger (e.g., `mail1.microsoft.com`) against the server's certificate when we connect. As we'll see later, this presents an administrative problem.

10.10 Client Mail Access

Although many clients are on the same machine as their local mail server and can read data from disk, many must access their mail through network protocols such as POP or IMAP. Obviously, these connections must be conducted securely. Securing these protocols is out of scope for this chapter, but it's important to note that unless the client fetches mail securely, securing the inter-server links is useless. Naturally, client/server communication is often secured via SSL as described in [Newman1999]. As usual, each protocol has some subtleties that must be addressed when using SSL. The reader might find it instructive to examine these standards after reviewing the guidelines in Chapter 7.

10.11 Protocol Selection

As we saw in Figure 10.2, SMTP already has an extension mechanism. Thus, it's very natural to use this extension to implement an upward negotiation strategy. Moreover, this approach is much more palatable to systems administrators; since SMTP traffic nearly always needs to come in through firewalls in order to be delivered, it's attractive not to have to open up another port for secured SMTP traffic.

10.12 Client Authentication

In the mail environment, we're actually concerned with two kinds of client authentication: *authenticated relaying* and *originator authentication*. The purpose of authenticated relaying is to allow relaying but to restrict the service to certain sending servers. The purpose of originator authentication is to provide information to the recipient about who sent the message. As we indicated in Section 10.5, this is an important requirement for e-mail. The recipient would like to be able to authenticate the sender.

Unfortunately, the nature of SMTP makes originator authentication very difficult. Because the sending server is usually controlled not by the client but is rather a system process, it is not likely to have the user's keying material. Thus, the best service we are likely to be able to offer is authentication for the sending server. The sending server can, of course, assert the client's identity but the receiver must trust the sending server. As we'll see in Section 10.23, the situation becomes even worse when we consider the possibility of relays between client and server.

10.13 Reference Integrity

As we said earlier, Internet mail uses e-mail addresses as references. When we're concerned with reference integrity we're concerned with delivering mail to the right receiver. Unfortunately, the receiving mail server is likely to be some mail server not under direct control of the user. The user's machine may not even be on the net at the time of delivery. As with originator authentication, we're immediately forced to settle for something less desirable than we would like. At best, reference integrity can mean

that we're delivering to the correct server and that we trust it to deliver the mail safely to the correct user.

10.14 Connection Semantics

Because the number of mail messages to a typical mail server is much fewer than the number of HTTP requests to a typical Web server, we don't have to worry about session resumption as much. We still have to be concerned with preventing truncation attacks, but SMTP's command-oriented nature works in our favor here. It's typically fairly clear when connections should be closed and when they should not.

10.15 STARTTLS

Now that we understand the problem of e-mail security from a design perspective, we're prepared to talk about STARTTLS. The standard approach to SMTP over SSL is described in RFC 2487 [Hoffman1999a]. We're also prepared to evaluate how well STARTTLS does at meeting the requirements discussed in the previous part of the chapter.

We begin by examining a simple transaction that uses STARTTLS. This allows us to see how the STARTTLS extension is used to upgrade from SMTP to SMTP over TLS. As usual, we'll discuss how to handle *connection closure*.

Once we have a clear picture of the behavior specified by RFC 2487, we can examine its interaction with some real world requirements. In particular, we'll examine the possibility of requiring messages to be transmitted with TLS as well as the interaction of STARTTLS with *virtual hosts*.

Finally, we'll consider what security protections STARTTLS actually provides. We'll examine the problem of providing *security indicators* that show what sort of protections were used for each link in the message path. We'll also consider the problems of providing *authenticated relaying*, *originator authentication*, and *reference integrity* with SMTP.

10.16 STARTTLS Overview

RFC 2487 was published roughly at the same time as the TLS specification, RFC 2246. Although SMTP over SSL had been in use for some time, RFC 2487 codified the rules for its use. A separate ports strategy (SMTPS) was briefly in use but did not see wide deployment and has been obsoleted by STARTTLS.

STARTTLS uses the SMTP extensions mechanism, documented in RFC 1869 [Klensin1995]. The server offers STARTTLS as one of its extensions, and the client can respond with the STARTTLS command. The client and the server then engage in a TLS handshake. Once the handshake is finished, they start over again with EHLO as if they had just connected.

```
New TCP connection: romeo(2515) <-> mike(25)
1 950072479.5489 (0.0225) S>C
data: 25 bytes
-----------------------------------
220 mike.rtfm.com ESMTP
-----------------------------------

2 950072479.5490 (0.0226) C>S
data: 21 bytes
-----------------------------------
EHLO romeo.rtfm.com
-----------------------------------

3 950072479.5495 (0.0231) S>C
data: 63 bytes
-----------------------------------
250-mike.rtfm.com
250-PIPELINING
250-STARTTLS
250 8BITMIME
-----------------------------------

4 950072479.5496 (0.0232) C>S
data: 10 bytes
-----------------------------------
STARTTLS
-----------------------------------

5 950072479.5536 (0.0272) S>C
data: 19 bytes
-----------------------------------
220 ready for tls
-----------------------------------

6 950072479.6269 (0.1004) C>S SSLv2 compatible client hello
  Version 3.1
  cipher suites
      TLS_DHE_DSS_WITH_RC4_128_SHA
      TLS_DHE_DSS_WITH_RC2_56_CBC_SHA
      TLS_RSA_EXPORT1024_WITH_RC4_56_SHA
      TLS_DHE_DSS_EXPORT1024_WITH_DES_CBC_SHA
      Cipher suite list trimmed
```

(continued)

Figure 10.8 *(continued)*

```
7 950072479.7353 (0.1084) S>C   Handshake
      ServerHello
        session_id[32]=
          73 bf d5 04 ac 45 c1 7e a6 04 82 d0
          ae 5c ed db f8 8f b4 2b f5 9e d0 6c
          52 08 87 20 16 0e 4d 54
        cipherSuite           TLS_RSA_WITH_3DES_EDE_CBC_SHA
        compressionMethod           NULL
8 950072479.7353 (0.0000) S>C   Handshake
      Certificate
9 950072479.7353 (0.0000) S>C   Handshake
      ServerHelloDone
10 950072479.7380 (0.0027) C>S   Handshake
      ClientKeyExchange
11 950072479.7380 (0.0000) C>S   ChangeCipherSpec
12 950072479.7380 (0.0000) C>S   Handshake
      Finished
13 950072479.7702 (0.0321) S>C   ChangeCipherSpec
14 950072479.7702 (0.0000) S>C   Handshake
      Finished
```

Client restarts SMTP session with a new EHLO

```
15 950072479.7707 (0.0004) C>S   application_data
    data: 21 bytes
    ----------------------------------
    EHLO romeo.rtfm.com
    ----------------------------------
16 950072479.7714 (0.0007) S>C   application_data
    data: 49 bytes
    ----------------------------------
    250-mike.rtfm.com
    250-PIPELINING
    250 8BITMIME
    ----------------------------------
17 950072479.7716 (0.0002) C>S   application_data
    data: 32 bytes
    ----------------------------------
    MAIL FROM:<ekr@romeo.rtfm.com>
    ----------------------------------
18 950072479.7723 (0.0006) S>C   application_data
    data: 8 bytes
```

```
    ----------------------------------
    250 ok
    ----------------------------------
19 950072479.7724 (0.0001) C>S   application_data
    data: 29 bytes
    ----------------------------------
    RCPT TO:<ekr@mike.rtfm.com>
    ----------------------------------
20 950072479.7731 (0.0006) S>C   application_data
    data: 8 bytes
    ----------------------------------
    250 ok
    ----------------------------------
21 950072479.7732 (0.0001) C>S   application_data
    data: 6 bytes
    ----------------------------------
    DATA
    ----------------------------------
22 950072479.7749 (0.0016) S>C   application_data
    data: 14 bytes
    ----------------------------------
    354 go ahead
    ----------------------------------
```

Client transmits actual message

```
23 950072479.7752 (0.0003) C>S   application_data
    data: 244 bytes
    ----------------------------------
    Received: (qmail 21198 invoked by uid 556); 9 Feb 2000 05:01:19 -0000
    Date: 9 Feb 2000 05:01:19 -0000
    Message-ID: <20000209050119.21197.qmail@romeo.rtfm.com>
    From: ekr@romeo.rtfm.com
    To: ekr@mike.rtfm.com
    Subject: test

    test message
    .
    ----------------------------------
24 950072479.8403 (0.0650) S>C   application_data
    data: 27 bytes
    ----------------------------------
    250 ok 950048042 qp 25454
    ----------------------------------
```

(continued)

Figure 10.8 *(continued)*

```
    Client closes connection
25 950072479.8404 (0.0001) C>S   application_data
    data: 6 bytes
    ----------------------------------
    QUIT
    ----------------------------------
Client FIN
26 950072479.8412 (0.0007) S>C   application_data
    data: 19 bytes
    ----------------------------------
    221 mike.rtfm.com
    ----------------------------------
Server FIN
```

Figure 10.8 Upgrading with STARTTLS

The trace in Figure 10.8 shows the beginning of an SMTP connection that upgrades with STARTTLS. This trace was generated using the popular qmail mailer with a patch to support STARTTLS, but a similar trace would be generated by other mailers.

As before, in segment 1 the server sends the 220 response to identify itself. The client then sends the EHLO command. This time when the server responds it offers the STARTTLS extension, indicating that it is prepared to process the STARTTLS command (segment 3).

At this point in our original SMTP trace (Figure 10.2) the client issued the MAIL command to start sending the message. Here, however, it issues the STARTTLS command to initiate the TLS handshake in segment 4.

In segment 5 the server responds with `220 ready for tls`, indicating that the client should proceed with the ClientHello. Note that the server could advertise STARTTLS in response to the EHLO but refuse it when it was actually requested. In fact, this is exactly what qmail does when configured without a certificate.

Segment 6 contains the client's ClientHello. Note that the client actually offers an SSLv2 backward-compatible handshake. Unlike HTTP Upgrade, RFC 2487 isn't very clear on exactly what versions of SSL and TLS servers and clients are expected to support. The use of the term TLS implies that they support TLS but not SSLv2 or SSLv3. However, most of the currently available STARTTLS implementations are based on toolkits that support SSLv2 and SSLv3 as well, so they usually offer SSLv2.

The rest of the handshake (records 7–16) completes in the usual way. The one surprising feature is that the server never requests client authentication. This is a fairly common feature of STARTTLS implementations. Thus, the receiving server has no

cryptographic way of identifying the sending server. In fact, the standard implies that it shouldn't. We'll see the reason for this in Section 10.18.

Once the handshake is finished, the client sends its EHLO encrypted over the newly negotiated TLS channel and the mail delivery proceeds as normal, except that it is encrypted using TLS.

10.17 Connection Closure

Note that the client simply closed the connection in Figure 10.8 without sending a close_notify. It just sent the QUIT command and then shut down its side of the connection. Similarly, the server sent its 221 response and then shut down its side of the connection. Recall that the purpose of the close_notify is to prevent truncation attacks. The question we need to ask is, does this represent a threat?

Obviously, this connection was not under attack, but is it possible that it could have been? The answer is no. The exchange of QUIT and 221 clearly indicates that both sides want to close the connection. These messages were sent over TLS so they could not have been tampered with by the attacker. Recall that in Chapter 7 we said that close_notify wasn't required if the application protocol has its own end-of-data markers. This is exactly such a case.

Other Situations

SMTP has the convenient property that both sides always know whose turn it is to talk and how long a command or response is. Thus, it's easy to determine whether an unexpected close has happened within a protocol message or between protocol messages.

Because mail servers automatically retry (after some timed interval) when they encounter errors, the appropriate response to most sorts of unexpected close is simply to log them and continue. However, a repeated pattern of premature closes may represent a denial-of-service attack and should be investigated.

In general, no agent should act on a partial message. Thus, if an agent receives only part of a command or an e-mail message, it should not process it. Similarly, senders should not assume that mail has been received by the receiver unless they actually see the 250 response indicating that the mail has been accepted for transmission.

Resumption

As required by the SSL specification, agents must not resume sessions that were closed without a close_notify. Thus, the session shown in Figure 10.8 cannot be resumed. This isn't as important an issue with SMTP as it is with HTTP because senders do not usually reconnect repeatedly within a short period of time. Nevertheless, a correct implementation would send a close_notify.

10.18 Requiring TLS

One might wish to configure one's mail server to require messages to be transmitted via TLS. Actually, it's technically possible to configure outgoing traffic and incoming traffic separately. You might choose to do this if you didn't care about who sent mail to you but you wanted to ensure that mail you transmitted was sent securely. Or, you might choose to accept mail only from authenticated sources but to transmit mail to anyone. Unfortunately, it's not really desirable to require TLS for either incoming or outgoing traffic.

Outgoing Traffic

Requiring TLS to transmit mail is easy. Simply refuse to connect to servers that don't offer the `STARTTLS` extension. However, because most servers don't support `STARTTLS`, this means that the vast majority of the messages that you try to send will bounce. One might wish to allow users to require that a particular message be transmitted with TLS, but there's no standard way to do so because there is no standard way to mark messages for TLS transmission, so you can't require TLS on other MTAs in the relay chain.

Incoming Traffic

RFC 2487 provides support for requiring TLS for all incoming traffic. The receiving server simply responds to commands (other than `EHLO`, `NOOP`, `STARTTLS` and `QUIT`) with:

```
530 Must issue a STARTTLS command first
```

However, RFC 2487 also explicitly forbids publicly referenced mail servers from being configured this way, with one exception: servers may refuse to relay traffic unless it came over a proper TLS-authenticated channel (see Section 10.21). The intent of this rule is to prevent administrators from configuring their systems in such a way as to break interoperability; requiring TLS would cut off e-mail from most of the Internet. However, it also makes enforcement of security properties very difficult, because it prohibits receivers from requiring that mail senders be authenticated.

10.19 Virtual Hosts

We saw in Chapter 9 that an upward negotiation approach (Upgrade) made virtual hosts work properly with HTTP. We might expect that they would work similarly well with `STARTTLS`. Unfortunately, due to an oversight in the specification, they don't. In HTTP Upgrade, the client used the `Host` header to indicate the virtual host that it wished to connect to. Similarly, we'd expect the sender to identify the host somewhere in the first few segments.

However, nowhere in the TCP segments (1–5) that precede the TLS handshake does the sender identify the domain of the recipient (which identifies the virtual host). This only happens when the sender issues the `RCPT` command, which is too late since the TLS connection has already been negotiated.

The appropriate fix for this problem is for the sender to provide the recipient's domain as an argument to the `STARTTLS` command. It shouldn't provide the local-part because that's not necessary and we'd like to keep it secret. If this is done, virtual hosts will work properly. Note that this isn't a problem with the `STARTTLS` approach, it's merely a specification error, and it's been suggested that it be fixed before RFC 2487 advances to Draft Standard.

10.20 Security Indicators

It's clearly desirable for users to be able to determine the security properties that were used to transmit the message. Because messages are often written to disk and then read by the clients, the logical place to put this information is in the message headers. This also saves us the trouble of modifying our mail reading protocols to carry this information.

Figure 10.9 shows an example of this procedure. This is the same message we saw transmitted in Figure 10.8 as represented on disk after being delivered. Note the next-to-last `Received` header, which reads in part `DES-CBC3-SHA encrypted SMTP`. If the sender had used client authentication, this header line might also contain the certificate that the sender used to authenticate or some other equivalent indicator of the sender's identity.

```
From ekr@romeo.rtfm.com Tue Feb 08 22:14:02 2000
Return-Path: <ekr@romeo.rtfm.com>
Delivered-To: ekr@mike.rtfm.com
Received: (qmail 25454 invoked from network); 8 Feb 2000 22:14:02 -0000
Received: from romeo.rtfm.com (216.98.239.227)
  by mike.rtfm.com with DES-CBC3-SHA encrypted SMTP; 8 Feb 2000 22:14:02 -000
Received: (qmail 21198 invoked by uid 556); 9 Feb 2000 05:01:19 -0000
Date: 9 Feb 2000 05:01:19 -0000
Message-ID: <20000209050119.21197.qmail@romeo.rtfm.com>
From: ekr@romeo.rtfm.com
To: ekr@mike.rtfm.com
Subject: test

test message
```

Figure 10.9 A message delivered with STARTTLS

Interpreting the Indicators

There are no standards for exactly how to place this sort of security information in headers. RFC 822 provides some guidance as to where in the header the information will come (after the `with` keyword) but other than that the implementation can arrange things more or less however it wants. Similarly, a client program could attempt to interpret them but will probably have to fall back on showing them to the user. Note that under normal circumstances, many clients don't even show the users `Received` headers unless they ask.

Last Hop

It's important to recognize that the headers are not cryptographically authenticated in any way. Thus, at the very best, the user has direct information about the last set of `Received` headers: those created by his local mail server on the *last hop*. The recipient must assume that his local mail server is trustworthy or he has no security. Thus, he can trust his mail server not to create false headers. However, any other headers may have been tampered with by any agent along the way.

Thus, in order to have any assurance about the security properties of the message all the way from sender to receiver, the connection must trust all hosts along the way. Moreover, those hosts must be cryptographically authenticated. Otherwise, an attacker might pose as a sender that you trust. Because the format of the `Received` fields isn't fixed, this evaluation can't really be made automatically and has to be made by the user.

10.21 Authenticated Relaying

There are two possible uses for certificate-based client authentication with STARTTLS: authenticated relaying and originator authentication. This section discusses authenticated relaying and the next discusses originator authentication.

The purpose of authenticated relaying is to allow a server to provide relaying services for servers it recognizes while denying those it does not. As we discussed in Section 10.7, open relays are often abused by spammers. Nevertheless, some sites need to relay mail. If the only machines that you relay from are behind the same firewall as your mail server, then you can often use IP-based authentication. However, often this isn't the case, either because there is no firewall or because your mail server needs to relay for machines at other places on the net.

In such cases, it's quite appropriate to use certificate-based client authentication to restrict relaying services. There are two interesting cases to consider. In the first, the sender has already connected with TLS. In the second, the sender has connected without TLS. In either case, the server will need to examine the `RCPT` command to see whether relaying has been requested.

If the sender has connected with TLS, then the server needs to check its certificate against the list of servers permitted to request relay. If no certificate has been presented, then the server will need to request a rehandshake. This is analogous to the case we saw

in Chapter 9 where an HTTP server requested a rehandshake to cause the client to authenticate.

RFC 2487 isn't completely clear on what to do if the sender hasn't used TLS at all, but presumably the receiving server should send the `530` response, indicating that the sender must use TLS and then request client authentication during the TLS handshake.

10.22 Originator Authentication

The other possible use of client authentication is to provide originator authentication. The purpose of originator authentication is to provide the recipient assurance of the sender's identity. This is clearly valuable to recipients because they don't want to be fooled by forged e-mail. Unfortunately, originator authentication works very poorly with STARTTLS.

Because RFC 2487 forbids requiring TLS, it's clearly not possible to require client authentication via TLS. Thus, it's not really possible to enforce any sort of originator authentication. However, a server might choose to request client authentication whenever negotiating TLS. In that case, if the sending server uses `STARTTLS` and has a certificate, the receiving server will at least know who sent the data on the last hop. If the recipient is really lucky, every single hop will have used TLS and every sender will have client-authenticated to every receiver. In this case, the user has some hope of determining who the originator was.

In order to do this, however, the user will have to examine the `Received` headers to determine that every host that handled the message is trustworthy and should have been in the path of the message. This is particularly difficult because it is unlikely that each receiver server will add the entire certificate chain of each sender into the headers. Thus, the ultimate recipient cannot determine if the certificates used by the originator and each server in the path can be validated to a CA that he trusts.

At the end of the day, then, there is no good way to use SMTP with TLS to do originator authentication in most cases. This conclusion isn't restricted to the details of `STARTTLS`. It's a general problem with the interaction between TLS and the SMTP service model. Even if the various headers were standardized and it were thus technically possible to distinguish legitimate from illegitimate senders automatically, it would still require trusting every host along the way, which is a fundamentally unreasonable requirement. Thus, the receiver can have no assurance that the sender is who he says he is or even that the message was encrypted the entire way.

10.23 Reference Integrity Details

The situation with respect to reference integrity is little better. We said in Section 10.2 that the sender would like to be able to know that the message will be delivered to the recipient and that it will be protected in transit. Unfortunately, the mechanics of SMTP and `STARTTLS` make this nearly impossible.

Secure References

The first problem we face is that e-mail addresses don't contain any indication that you can send mail to them securely. It's obvious why this isn't the case: what would such an indication mean? The recipient doesn't know anything about the servers on the sender's side of the connection. The most he can possibly know about is the servers on his side of the connection.

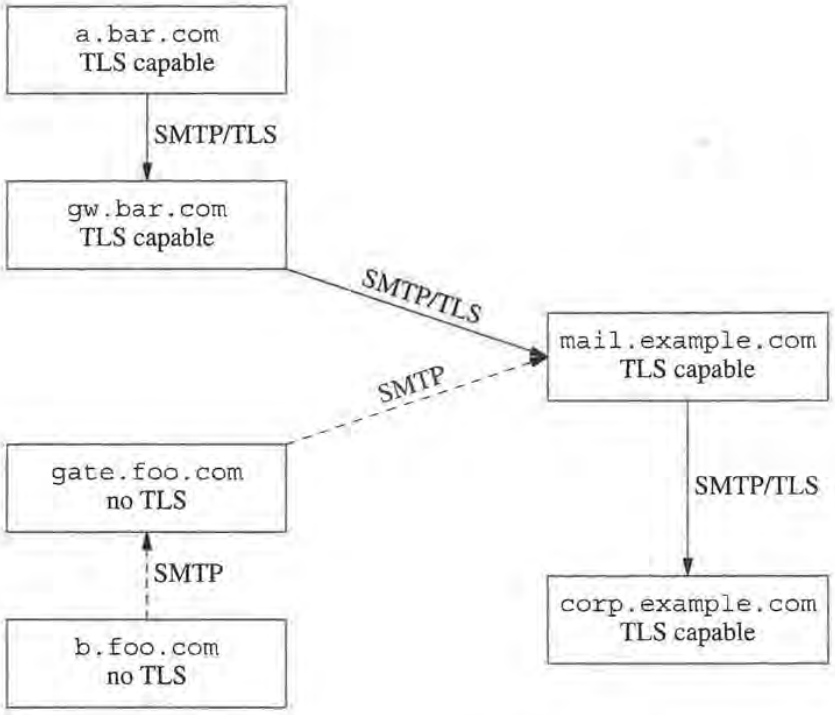


Figure 10.10 A network with secure and insecure servers

Consider the network setup shown in Figure 10.10 from the perspective of the user `foo@corp.example.com`. He knows that when servers look up the domain `example.com` they'll get an MX record pointing to `mail.example.com`. He also knows that `mail.example.com` and `corp.example.com` support TLS. Thus, he knows that the last hop will be delivered securely and that if the server connecting to `mail` is TLS-capable, then it might be able to connect securely, but that's the most he can be certain of.

In fact, two different senders get two different sets of security properties. A user on `a.bar.com` is able to have his mail sent over encrypted links all the way to

corp.example.com. By contrast, mail sent from b.foo.com travels over insecure channels all the way to mail.example.com, where it's encrypted for the final hop to corp.example.com. There's no reasonable way for the recipient to predict what the behavior will be for any given sender.

Obviously, we could use a weak indicator that said essentially: "If you're TLS enabled, you should be able to send mail to me with TLS." However, this would necessarily involve a different form of e-mail address, because our current forms have no room for such an indication. This seems like a high cost for little benefit.

Enforcing Security

Imagine that the sender has some external information to the effect that the recipient would like to receive e-mail over a TLS connection. Furthermore, the sender knows that his local servers are TLS-capable. He can certainly configure his SMTP agent to do TLS only. However, this doesn't buy very much.

Again, consider the network in Figure 10.10. A user on a.bar.com would like to send mail to foo@corp.example.com. He knows that he's security capable and that the recipient is, so he configures his agent to do TLS, which it does, all the way to gw.bar.com. However, gw.bar.com doesn't know to negotiate TLS with mail.example.com, so an active attacker can downgrade the connection to regular SMTP. All this requires is tampering with the EHLO response to remove the offer of STARTTLS. The rest of the connection can be left unchanged.

Clearly, if we're to get anywhere we'll need some way to instruct every server along the way to do TLS. This could be done either by having a new mail header or (more likely) with some argument to the STARTTLS extension. Of course, such an extension would mean that if any server in the path between sender and receiver doesn't speak TLS then the mail will simply bounce, but sometimes sacrificing interoperability is the price we pay for security, so this might be worth it. However, as we'll soon see, it isn't.

Relaying versus Security

Unfortunately, the possibility of SMTP relaying more or less destroys any attempt to enforce TLS all along the message path. Again, consider the message we sent in Figure 10.10, but under a DNS spoofing attack where the attacker forges MX records for corp.example.com. Ordinarily, they would read something like

```
corp.example.com  preference = 10, mail exchanger=mail.example.com
```

The attacker substitutes this with

```
corp.example.com  preference = 10, mail exchanger=mail.attacker.com
```

Now, when the a.bar.com goes to deliver the message, we get the path shown in Figure 10.11 (assuming that the attacker bothers to deliver the message at all instead of just dropping it). Note that every hop in the delivery has still been transmitted with TLS.

Moreover, the contents of the MX record match the attacking server's certificate. We've simply added another relay in the chain. Clearly, then, requiring TLS alone is insufficient. We need servers to enforce the identity of the relays they're transmitting to. This shouldn't be surprising, because we said in Chapter 7 that just having a valid certificate isn't enough for reference integrity.

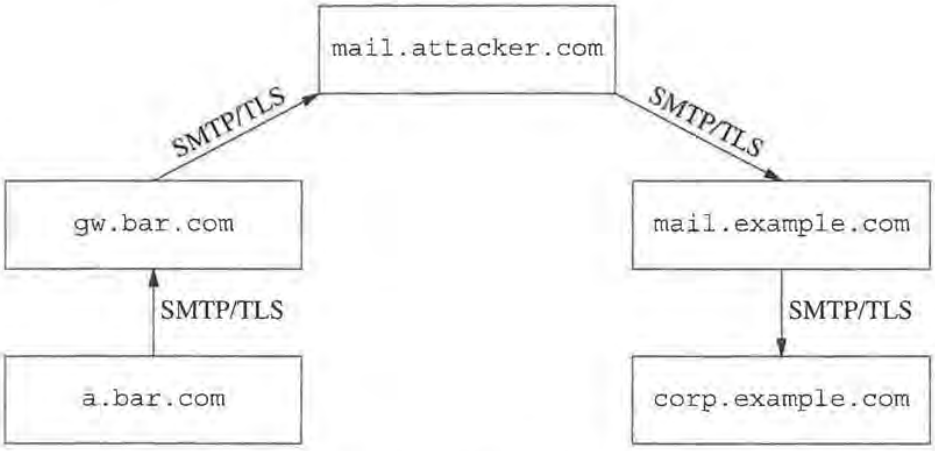


Figure 10.11 A fake mail exchanger

The problem we face is that there's nothing fundamentally wrong with the path shown in Figure 10.11. In this case, the extra relay is an attacker, but this sort of topology is relatively common. It's not even that uncommon for mail from organization A to organization B to be routed through a third organization, as we discussed in Section 10.7. Thus, we can't just check the hostnames on the certificates against the e-mail addresses in the `RCPT` command.

The only approach that's known to work at all is for every legitimate relay for a given host to have a certificate with that host's domain as its identity. Although this is secure, it's ultimately a configuration disaster, because every such relay must be manually configured. Worse yet, unless revocation is carefully done, the compromise of any such server compromises all traffic to the target domain permanently. If we had Secure DNS, then we could know that the MX records were correct and then we could check the name in the MX record against the server certificate. However, Secure DNS deployment has so far been minimal so this is not a real option.

At the end of the day, then, we're left with the conclusion that STARTTLS can't provide any real assurance to the sender of the security properties of the message he's sending. Even if he's willing to restrict himself to sending mail only over TLS, he can't be sure that the message will be securely delivered to the intended recipient. He certainly can't be sure that it was delivered undamaged. Once again, this problem doesn't

Ex.1024
Askeladden, L.L.C. / Page 440 of 788

have anything to do with the particular details of STARTTLS. It's simply inherent to using TLS to secure the hop-by-hop connections. Hop-by-hop security doesn't guarantee security between two endpoints.

10.24 Why Not CONNECT?

Clearly, the problems we're facing with SMTP are related to the existence of relays. HTTPS dealt with intermediaries like proxies by essentially bypassing them. The CONNECT method tells the proxy to get out of the way of the HTTPS connection. With an end-to-end channel between the client and server, HTTPS can guarantee the kind of security properties that STARTTLS can't.

This suggests the obvious question of whether it's possible to use some sort of CONNECT variant to tunnel through SMTP relays in the same way that we tunnel through Web proxies. Unfortunately, this is both technically and administratively problematic.

Technical Problems

One technical problem this approach faces is that this would completely break relaying for disconnected servers. As we said in Section 10.3, some organizations connect to the net only occasionally to get their mail. Such networks use a relay to store their mail for them. Clearly, it's not possible to tunnel to a server that's not connected to the net.

Conceivably, we could patch this problem by treating the final relay as if it were the recipient's local server. We'd issue it an appropriate certificate for the recipient's domain and allow it to negotiate the TLS connection with the sender. Because the alternative is for such networks not to receive encrypted mail at all, this might be acceptable.

Mail server software simply isn't designed to operate in this way. It expects to be able to queue messages and process them at leisure. This is important because it allows it to be robust against temporary network failures. Tunneling through the servers would entail major architectural changes to the software. Moreover, it would violate important invariants of the Internet mail environment; mail should be deliverable even if both networks are not online at the same time. That's what store-and-forward means.

Finally, consider the effect on the sending clients. In the current environment, mail is "fire and forget." A client can send mail to its local mail server and the user can disconnect from the network completely and still trust that his mail will be delivered. If the client is going to tunnel through to the end server, then it must remain online—and arrange to retry even if the recipient's server is temporarily down. Users frequently connect for only a few minutes at a time so this isn't really an acceptable result.

Administrative Problems

The more serious problems are administrative. Unlike Web servers, mail servers often are behind firewalls. HTTP CONNECT only allows clients to tunnel out from behind

firewalls. The approach we're considering would allow senders to tunnel *in*. Allowing an encrypted connection to a mail machine behind the firewall understandably makes administrators quite nervous, especially because Sendmail, the dominant UNIX mail program, is notorious for having security problems.

10.25 What's STARTTLS Good For?

In Section 10.2, we named a number of desirable properties for an Internet mail security protocol. We wanted senders to be sure that their mail would be delivered securely to recipients. We wanted recipients to be able to authenticate the sender and know that messages hadn't been tampered with in transit. In Chapter 9, we saw that HTTPS met all of these goals, yet we've just seen that STARTTLS doesn't allow us to have any of these properties. Naturally, the question arises: What is STARTTLS good for?

Passive Attacks

The attacks that we've described for STARTTLS all rely on the attacker tampering with the network, typically either by playing man-in-the-middle or by contaminating the sender's DNS cache. In essence, they rely on the attacker pretending to be one of the communicating parties and thus subverting our security protections. This is a perfectly reasonable form of attack to worry about, but it's also comparatively difficult to mount. Moreover, the attacker needs to be online and transmitting data, which makes it easier to track him down. Thus, it's reasonable to consider what sorts of protection STARTTLS provides against passive attacks.

STARTTLS is useful if we're merely interested in opportunistic cryptography, that is if we ignore active attacks. Because senders no longer have to worry about connecting to the wrong gateway, all the attacks based on relay substitution disappear, as do the downgrade attacks. All we really have to worry about is whether or not servers support STARTTLS. If all the servers in the path of message support it, then the message can be delivered confidentially. Of course, we still need to trust the gateways not to read our mail. Depending on if they belong to a third party, this may or may not be a reasonable assumption.

We don't even need to have an "enforce TLS" flag. The servers will automatically upgrade where appropriate. However, it would be possible to introduce such a flag to indicate that servers should bounce mail if they can't deliver it securely.

Note that because we're not worrying about active attack, certificates add very little value in this situation. Recall that the major purpose of certificates is to prevent the active attacker from substituting his key for that of the legitimate receiver. Since we've ruled that attack out of scope, certificates aren't buying very much. We could dispense with them entirely and simply use anonymous DH. In some sense, this is making a virtue of necessity, because we saw in Section 10.8 that checking to see if a certificate represented the legitimate receiver was one of the hard problems with SMTP.

No Sender Authentication

Although we can have confidentiality, it's meaningless to talk about authentication or message integrity. The whole point of authentication is that it provides assurance of the message in the face of active attack. Since we've ruled that sort of attack out of scope, authentication and integrity are simply no longer relevant.

10.26 Programming Issues

Implementing SMTP over TLS doesn't really require any idioms that we haven't seen before. However, there are a few minor issues with SMTP that we didn't have to be concerned with when discussing HTTPS. The rest of this chapter describes two of these issues: implementing the STARTTLS extension and server startup. These issues aren't difficult enough to require sample code, but they are important enough to take note of.

10.27 Implementing STARTTLS

Conceptually, implementing STARTTLS is the same as implementing the CONNECT method for HTTPS. In both cases, the client needs to perform an application-layer handshake in the clear and then transition to SSL/TLS. The server code is analogous to the client code. However, STARTTLS is a little more complicated than CONNECT. HTTPS clients know that they are going to a proxy and CONNECT handling is simple enough that the client side can be written without actually invoking the HTTP engine. However, because STARTTLS is an upward-negotiation strategy, it typically gets wired directly into the SMTP engine.

State

The primary issue we need to be concerned with is SMTP-engine state. RFC 2487 is quite clear that any extensions advertised or negotiated before the TLS handshake no longer apply to the connection once the handshake has been completed. Thus, the implementation must be prepared to throw away all state and renegotiate extensions once the TLS handshake has completed. There are two reasons for this requirement: first, the attacker might have interfered with the extension advertisements. Second, the server might be willing to offer different services when TLS is being used.

Network Access

For efficiency reasons, it's quite common for SMTP implementations to want to buffer network reads and writes. Some implementations use `stdio` but some use their own network buffering code. In any case, the SSL implementation needs to be able to read

and write to and from the network. Most SSL implementations provide some sort of abstract networking API which allows the application to provide its own I/O routines. OpenSSL does this using the `BIO` object, as shown in Figure 10.12.

sclient.c

```
26          ssl=SSL_new(ctx);
27          sbio=BIO_new_socket(sock,BIO_NOCLOSE);
28          SSL_set_bio(ssl,sbio,sbio);
```

sclient.c

Figure 10.12 OpenSSL BIO initialization

This abstraction allows the user to provide his own I/O routines as long as they can be represented as a `BIO`. sIn theory, PureTLS could do the same thing by having the user provide `InputStream` and `OutputStream` objects. However, as we discussed earlier, this functionality is not currently available in PureTLS. Note that if the buffering in the SMTP implementation is intended to read one line at a time, it will have to be deactivated before the TLS handshake because TLS data is not line-oriented.

If the SSL toolkit doesn't provide a replaceable networking API, it almost certainly reads and writes directly from sockets. In that case, the SMTP implementation will need to arrange to give a raw socket to the toolkit. This is relatively easy with `STARTTLS` because the length of commands and responses is always known, as is the start of the SSL handshake. Therefore, there is no need to be concerned that some of the SSL handshake will be trapped in the SMTP read/write buffers.

10.28 Server Startup

It's quite common to run mail servers in the same fashion as Web servers. We have a single server process that accepts connections from clients and then creates a new control thread to handle them. We already saw how to write this sort of program in Chapter 8. However, many mail servers on UNIX systems are configured differently. UNIX has a program called inetd. Inetd is a *superserver*. It listens for connections on a number of ports and has a configuration file that tells it which program to run to handle each kind of connection. Ordinarily, this approach works fine for a lightly loaded server, but it presents us with problems when we're using SSL.

Recall from Chapter 8 that SSL agents typically have a one-time initialization of an SSL context which contains keying material, randomness, etc. This context is then used for all subsequent connections. Clearly, if we're running our mail server out of inetd, we'll have to create our context object every time we receive a piece of e-mail. This presents two problems. First, we have to arrange to get access for the server to get access to its keying material. Second, we have to ensure that initialization is fast.

Keying Material

As we saw in Chapter 5, the standard way to protect keying material is to encrypt it under a passphrase. If we're running a single server process like an HTTPS server, we can simply prompt the user for the passphrase at system startup. This is somewhat inconvenient since it makes unattended startup difficult, but it's not unworkable. However, if our server runs out of inetd, this approach will no longer work.

The common approach to follow in these circumstances is simply to leave the private key on disk in the clear, but protected by file permissions or access control lists. A slightly more sophisticated solution is to have a single server process that is started at machine startup time. When it starts, it prompts the user for a password. The mail server program then contacts the server to obtain the keying material. Access to the server program is controlled via file permissions or ACLs.

The advantage of this approach is that the keying material is still encrypted on disk. Thus, if the machine is turned off, the passphrase will be needed to restart it. This provides some protection against server theft. Obviously, if the key is stored in hardware then the system will not be vulnerable to these attacks.

Fast Initialization

Recall that one of the major motivations for having a context object was so that time-consuming initialization could happen only once. Thus, if we have time-consuming initialization stages, they need to be somehow avoided. The common tasks that consume time at startup are generation of ephemeral keys and randomness seeding. This data can be precomputed and stored in files. As we saw in Chapter 9, our server can read this data from the files on startup.

10.29 Summary

Although the SSL designers intended that SSL could be used as a generic security layer, SSL was not specifically designed to work well with SMTP. However, the serious need for security for mail messages led to attempts to use SSL to secure SMTP, despite some serious mismatches between the SMTP model and the services SSL provides. In this chapter, we explored the standardized use of TLS with SMTP and examined some of the problems with this approach.

E-Mail is a store-and-forward system. The sending agent sends its message to the next server in the path using SMTP and that server then makes its best effort to deliver the message.

RFC 2487 specifies a method for using TLS to secure individual SMTP links. It introduces the `STARTTLS` extension which clients and servers can use to negotiate from a regular SMTP connection to SMTP over TLS.

Relays create problems. Because individual SMTP links are what is secured, each relay between client and server must be trusted. Moreover, because relay discovery is done via DNS, it's difficult to prevent attackers from posing as relays.

Senders can't determine security properties. Because e-mail addresses contain no information about the security of the transmission channel, a sender cannot know that his message will be sent confidentially.

Recipients can't get sender authentication. RFC 2487 forbids requiring TLS to receive mail. Even if TLS is used, the existence of relays makes it very difficult to determine the original sender.

STARTTLS provides protection against passive attack. If a sending and receiving server are both TLS capable, they will by default negotiate a TLS connection, thus protecting against sniffing.

11

Contrasting Approaches

"If you only have a hammer, you tend to see every problem as a nail."
—Abraham Maslow

11.1 Introduction

The previous chapters have focused on SSL. We've discussed its design goals, how it works, and how to use it. We've also examined two application-layer protocols that have been secured with SSL: HTTP and SMTP.

At this point, you should have a fairly good idea of how SSL works and what it can do. This, the final chapter, takes a broader view. As we saw in Chapters 9 and 10, certain security tasks can be very difficult to accomplish with SSL. In many cases, other security techniques can do the job much more cleanly and easily. Thus, it's important to be familiar with other techniques in order to be able to choose the best one. All too frequently, people use SSL to secure some application because it's the only approach they know about.

This chapter considers three other security protocols. First, we consider IPsec, which provides security for IP packets. IPsec operates at a lower layer than does SSL but provides many of the same security services. It also has some of the same limitations.

The second approach we consider is object security. Instead of encrypting the channel, object security sends protected objects over a clear channel. Because the individual protocol objects need to be secured, object security protocols are usually application specific. We'll examine Secure HTTP, which provides security for HTTP transactions and S/MIME, which provides security for Internet mail messages.

Ex.1024
Askeladden, L.L.C. / Page 447 of 788

11.2 The End-to-End Argument

In this chapter we survey protocols at several layers of the protocol stack. As we'll see, the higher—closer to the application layer—we get in the protocol stack, the better security services we're able to provide. This is one instance of the end-to-end argument, best stated by Saltzer, Reed, and Clark [Saltzer1984] as:

> The function in question can completely and correctly be implemented only with the knowledge and help of the application standing at the end points of the communication system. Therefore, providing that questioned function as a feature of the communication system is not possible. (Sometimes an incomplete version of the function provided by the communication system may be useful as a performance enhancement.)

[Saltzer1984] is one of the great papers in computer science and is important reading for anyone who's interested in understanding networking. The argument can be summarized as follows: Any communications system involves intermediaries, such as network devices, computers, and programs which are unaware of the total context of the communication being involved. These intermediaries are therefore incapable of ensuring that the data is processed correctly. [Voydock1983] makes this point specifically in the context of security.

This argument has obvious intuitive appeal and it coincides with our experience with HTTP and SMTP. HTTPS and SMTP/TLS worked relatively well until we started introducing proxies and relays. In order to get HTTPS to work properly with proxies we had to tunnel through the proxy—in essence creating an end-to-end channel—and SMTP/TLS security is badly broken by relaying.

11.3 The End-to-End Argument and SMTP

It's easiest to understand the end-to-end argument with an example. Consider the case of SMTP over TLS. The application at the "end-points of the communications system" is the e-mail program used by the sender and recipient. However, SMTP/TLS provides security at a lower level: mail transport between mail relays. As a consequence, it is incapable of providing correct security services. This manifests itself in a variety of ways, but we'll consider only two: last-hop delivery and relaying.

Last-Hop Delivery

Recall that mail delivery to the end user is not accomplished via SMTP. Rather, it's done by some other protocol such as IMAP or POP or merely by reading the mail off the disk. This means that no possible SMTP/TLS measure can ensure integrity and confidentiality of the data between the local server and the mail reading client, because the communications channel between the server and mail reader might be compromised.

Even if we could somehow secure that protocol, say by running it over SSL, it would still be possible for the local server to be compromised, thus allowing the message to be compromised. No link-level security measures can protect us against this problem. By contrast, if the security is provided end-to-end by the mail programs, then this attack is no longer possible.

Relaying

The problem we face with relaying is similar. SMTP/TLS provides *link-level* security. The links between relays are secured but any given relay might be compromised. Worse yet, SMTP/TLS has no way of indicating that the next hop must be encrypted—or even what its identity is—so attackers can substitute themselves as a man-in-the-middle or downgrade the SMTP connection. Again, if the security were provided end-to-end, then we could prevent this attack.

An End-to-End Solution

The solution is to provide security end-to-end. Because there isn't an SMTP connection between the recipient and the local server, we can't establish an SMTP tunnel as we did with HTTP `CONNECT`. The correct approach is to create secure objects and transmit them via ordinary SMTP. As we'll see in Sections 11.17–11.21 this is exactly what S/MIME does.

11.4 Other Protocols

We'll be considering three protocols: IPsec, Secure HTTP, and S/MIME. Each protocol is capable of doing some but not all of the jobs that we've seen SSL do. IPsec is a general IP security solution. Thus, it can be used to provide generic security for any application that runs over IP in the same way that SSL can be used to provide security for any application that runs over a connection-oriented transport. Secure HTTP and S/MIME are both *object security* protocols designed for specific application layer protocols. Secure HTTP provides security for HTTP requests and responses by treating each request or response as an individual protected object. S/MIME provides security for individual e-mail messages.

Figure 11.1 shows the relationship of the various protocols in the protocol stack. Security protocols are shaded. At the bottom we have IP, over which we carry essentially all our data. IPsec is simply a set of extensions to IP which add security services. Thus, any application that can run over IP (and hence over TCP and UDP) can easily run over IPsec. Note that although UDP is shown layered only over IPsec, it can obviously run over IP as well.

Although SSL can in theory run over any connection-oriented transport, in practice it's nearly always used over TCP. Protocols that would ordinarily be layered over TCP

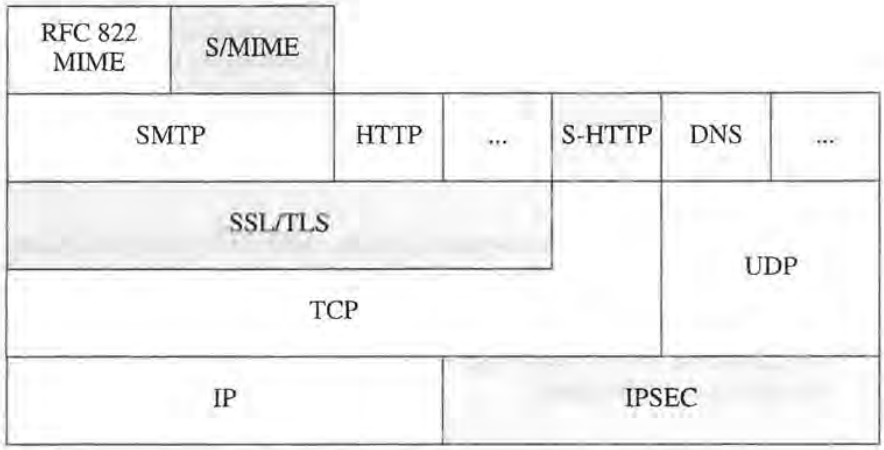


Figure 11.1 Where security fits in the protocol stack

can then be layered over SSL. Figure 11.1 shows three: SMTP, HTTP, and Secure-HTTP. Note that although we show them layered over SSL, of course HTTP and SMTP can be run directly over TCP.

At the top of Figure 11.1 we show RFC 822 and MIME messages, which are layered over SMTP. S/MIME messages are a specific kind of MIME message which provides cryptographic security. Thus, they can be carried over SMTP exactly as RFC 822/MIME messages are.

11.5 IPsec

IPsec is the umbrella term used for several different technologies standardized by the IETF IP Security working group and documented in RFC 2408: *Internet Security Association and Key Management Protocol* (ISAKMP) [Maughan1998]; RFC 2409: *Internet Key Exchange* (IKE) [Harkins1998]; RFC 2402: *Authentication Header* (AH) [Kent1998a]; and RFC 2406: *Encapsulating Security Payload* (ESP) [Kent1998b]. These technologies work together to provide security for IP traffic. RFC 2401 [Kent1998c] describes the architecture of IPsec.

Like SSL, IPsec has a key exchange and parameter management facility provided by ISAKMP and IKE and a data protection facility provided by AH and ESP. The glue between these facilities is provided by *security associations* (SAs). ISAKMP and IKE are used to establish SAs which are used by AH and ESP to protect the data. Unlike SSL, the key management and traffic protection functions are completely separate. It's possible to use AH and ESP without IKE, provided you have some other method of

establishing SAs. Theoretically, it's possible to use IKE to exchange keys for non-IPsec protocols, but in practice this doesn't happen.

Our Approach

In order to compare IPsec and SSL, we first need to have a grasp of the fundamentals of how IPsec works. Thus, we first provide a high-level discussion of IPsec. We start by describing the three major components: *SAs*, *ISAKMP/IKE*, and *AH/ESP*. Then we'll examine how they work together to protect IP traffic. Finally, we'll compare secure communications with IPsec to SSL to see the advantages and disadvantages of each approach.

11.6 Security Associations

An IPsec SA loosely corresponds to an SSL session. It refers to a set of negotiated algorithms and keying material to be used to transmit data between two hosts. A security association differs from an SSL session in that it is unidirectional. In order for a pair of hosts to communicate securely, they must have two SAs, one for each direction. Each SA has a *security parameter index* (SPI): a 32-bit value identifying the SA. The SPI may not be globally unique but each SPI, source IP address, protocol (AH or ESP) triplet must be unique. SPIs are intentionally short because they are carried in each protected packet. A longer SPI would consume excessive network bandwidth.

11.7 ISAKMP and IKE

Before any data can be sent using either AH or ESP, an SA needs to be established. The IPsec architecture document (RFC 2401) provides for two methods of SA establishment: manual keying and ISAKMP/IKE. Manual keying means setting the keys and algorithms manually on both hosts. ISAKMP and IKE provide automatic key and session management in a similar way to the SSL handshake.

ISAKMP provides a generic framework for negotiating security associations and keys. It describes handshake phases and messages, as well as describing how to negotiate algorithms. However, it does not provide any specific key exchange methods. IKE provides key exchange methods based on Diffie-Hellman.

The IKE key exchange is based on STS [Diffie1992], Oakley [Orman1998], and SKEME [Krawczyk1995]. The two parties exchange Diffie-Hellman shares (public keys) and use the shared key to derive traffic encryption and message authentication keys. The DH shares can be authenticated using digital signature, public key encryption, or a shared secret. Note that because we're using DH we automatically get *perfect forward secrecy* (PFS). (See Chapter 5 for a discussion of PFS.) IKE defines a number of well-known DH groups; implementations can use private groups but most implementations use one of the public ones. By contrast, SSL leaves the choice of group up to the server.

Identity Protection

One of the unique features offered by IKE is identity protection for the parties in the exchange. In *Main Mode*, the DH shares are exchanged in the clear but the certificates and authenticating data for the shares are encrypted. Figure 11.2 shows this mode in operation.

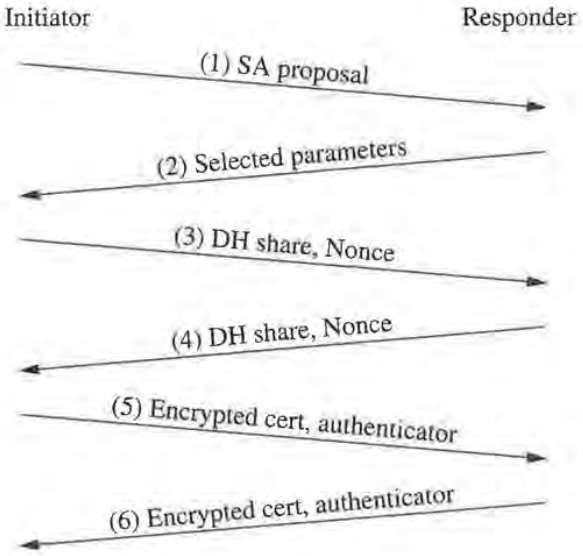


Figure 11.2 IKE Main Mode

IKE makes no distinction between client and server because this distinction does not exist at the IP layer. Instead, the sender of the first packet is called the *initiator* and the second the *responder*. We show the initiator on the left and the responder on the right. The handshake takes place in a series of *exchanges* where the initiator and responder each send a single message. Thus, Main Mode consists of three exchanges.

In the first exchange, we see the initiator sends a *proposal* for the algorithms that it's willing to support. The responder selects one from that set. In the second exchange, they each send a *nonce* (a random string analogous to the random values in SSL) and their respective DH shares. Finally, in the third exchange, they exchange signatures over the DH shares as well as over other parts of the exchange.

The end result of this handshake is roughly equivalent to an SSL handshake using DH and client authentication. The client and the server are mutually authenticated and they share an agreed upon set of keys and algorithms. There is no way in IKE to have one-way authentication in the way that SSL only authenticates the server.

The reason for having three exchanges in Main Mode is to protect the identity of the parties from sniffers. Identity protection is primarily useful where one host might have multiple identities and wants attackers to be unable to tell which one it's using for a specific handshake. By first doing an anonymous DH exchange and then conducting the rest of the exchange over an encrypted channel, IKE stops a passive attacker from determining the identities of the communicating parties. Although it's possible to provide this service in SSL by using an anonymous DH exchange followed by a normal exchange, this approach is far less clean than IKE Main Mode because it requires multiple key agreement phases. However, if this sort of protection is not desired, then all of these exchanges can be collapsed into a single exchange. This is called IKE *Aggressive Mode*.

Two-Phase Operation

Whereas SSL handshakes apply to a single connection, IKE is used to establish parameters for all traffic between two machines. It's very possible that different types of traffic will need different sorts of protection. For instance, Telnet sessions would probably need encryption, whereas DNS service needs only authentication.

In order to meet these different needs, IKE uses a *two-phase* approach. The first IKE handshake between two machines is used only to establish an ISAKMP SA. This SA is used only to transmit further IKE traffic. Thus, in order to actually transmit normal IP traffic, you need to do another IKE handshake to establish SAs for the particular type of traffic (protocol, port numbers, etc.) that you want to secure.

IKE provides a special *Quick Mode* for these new handshakes. Quick Mode is like SSL session resumption and can be used to generate new keying material from the original SA without doing a new key exchange. Alternatively, if PFS for the new SA is desired, a new DH key agreement can be performed. The authentication stages are still skipped. Because the exchanges take place over an authenticated channel, they are implicitly authenticated. Quick mode sets up a pair of SAs, one in either direction.

ISAKMP Transport

In SSL, the handshake happens in-band on the TCP connection that will be used to transport the data. Because ISAKMP is intended to establish communication parameters between two machines in general rather than over a single connection, this is obviously not possible. Instead, UDP port 500 has been designated for ISAKMP traffic.

Note that a result of using UDP is that there is no guarantee that packets will be delivered from sender to receiver. Thus, any ISAKMP implementation must maintain retransmit timers. If a given message is not responded to, the implementation must retransmit it. This adds some programming complexity not required by SSL.

One serious drawback to using UDP is the interaction of UDP with firewalls. Many firewalls block UDP entirely. Thus, if your IPsec host is behind a firewall, you'll need to configure your firewall to pass or proxy UDP traffic on port 500. This requires convincing your firewall administrator, which is often difficult, because holes in firewalls, especially for incoming UDP, are considered dangerous.

11.8 AH and ESP

Whereas SSL has a single unified record format, IPsec has two—AH and ESP—which do slightly different jobs. AH provides solely for message authentication and anti-replay. ESP provides for traffic encryption as well as optional message authentication and anti-replay.

AH

The idea behind AH is to provide a new IP header that authenticates the IP data and as much of the IP header as possible. It does this by MACing the data and most of the header. Because some of the header fields may change in transit, they are excluded from the MAC. Figure 11.3 thus shows a packet protected with AH, both before and after the addition of AH. MACed sections are shaded (note that even though the entire header is shaded, only parts of the header are MACed).

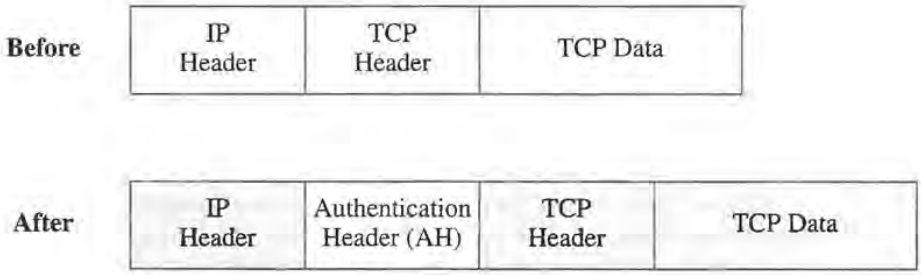


Figure 11.3 Protecting with AH

The AH header also contains a sequence number which is included in the MAC. This number can be used to provide replay detection. Note that normally functioning networks will sometimes see replays due to retransmission by intermediate routers. The purpose of replay detection is simply to ensure that the host's TCP and UDP layers see each packet only once. Replay prevention is primarily necessary for UDP, because TCP automatically handles and rejects replayed packets.

ESP

ESP provides encryption and optional authentication for the payload of an IP packet. Unlike AH, no protection is provided for the header at all. Only the payload of the message is protected. ESP can provide encryption, authentication, or both. Figure 11.4 shows a packet before and after ESP is applied. Protected sections of the packet are shaded.

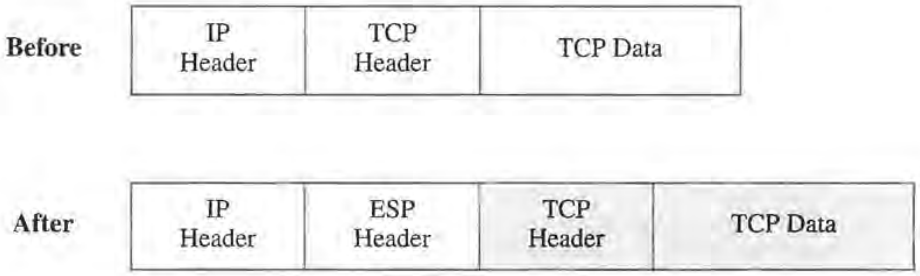


Figure 11.4 Protecting with ESP

Tunnel Mode

Note that because ESP provides no protection for the header, it's possible for an attacker to see (or change) the source or destination address. Both AH and ESP provide a mode called *tunnel mode* (the simpler mode we've just described is called *transport mode*) where the entire IP packet is protected. In essence, this is done by wrapping the packet to be protected in another packet which has AH or ESP applied. Figure 11.5 shows such a packet before and after ESP is applied.

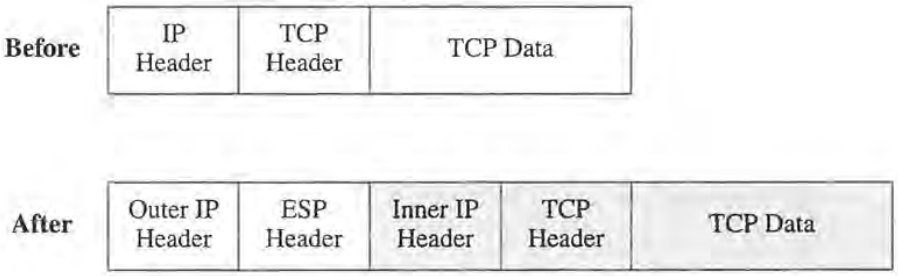


Figure 11.5 ESP in tunnel mode

In general, transport mode should be used for traffic between two hosts, because repeating the headers as in tunnel mode is unnecessary and wastes bandwidth. The sender and receiver address do not need to be cryptographically verified because they are implicit in the SA. Tunnel mode is primarily used in creating *virtual private networks* (VPNs). Two IPsec-capable routers can be configured to encrypt all traffic passing between them. This configuration acts as if the networks were directly connected via a private line, hence the term VPN. Tunnel mode should be used because the IP addresses may be that of any machine behind the router.

Ex.1024
Askeladden, L.L.C. / Page 455 of 788

11.9 Putting It All Together: IPsec

Let's consider what happens when a host wishes to transmit a packet to another host with which it has never communicated. It can't do so until it has an SA set up to protect the data. Thus, it holds the packet while it performs the IKE exchanges necessary to set up the required SA. Once the SA is negotiated, it protects and transmits the packet.

When a packet is received, the host first needs to look at the SPI (in the AH or ESP header) to determine if it corresponds to a known SA. If it does, it reverses the applied protections and passes the data to the IP stack. Note that all of these behaviors can happen transparently to the application program. It just uses normal network calls and the data is automatically protected and unprotected.

Policy

So far, we've skipped over the question of how hosts decide what sorts of protections IKE should negotiate. In general, it's anticipated that there will be two mechanisms. First, system administrators will be able to set policies for a host. These policies would describe what sort of protections to negotiate based on the destination host, port, and protocol. Thus, applications can get some default security without any change to their code at all.

In some cases, applications will wish to have direct control of security services. To facilitate this, operating systems can offer extensions to the sockets API allowing the application to control the security to be negotiated for that socket. Of course, the application program must then be changed to use these extensions.

When we have multiple policies for different kinds of traffic, it is possible that although we have an SA for a given host, it has the wrong sort of protection. In that case, IKE will have to negotiate a new SA. Similarly, a host might receive a packet which has an SA that doesn't match the policy for the kind of packet it is. In such cases, the IP stack will have to reject the data, often generating some kind of ICMP error. Upon receiving such an error, the sender should negotiate an appropriate IPsec SA.

11.10 IPsec versus SSL

Note that despite a number of differences, IPsec and SSL are conceptually very similar. Each protocol has a handshake phase where it negotiates keys and parameters and then a data transfer phase where it protects traffic. IPsec behaves most like SSL when using ESP in transport mode. It provides authentication for the traffic data but not any of the IP headers. However, with IPsec, because the TCP messages are protected, TCP FINs can be used to close the connection securely. There is no need for special IPsec closure messages.

In many ways, IPsec can be thought of as SSL, only more so. SSL can be used to secure any traffic that goes over TCP, but IPsec can be used to secure any traffic that goes over IP, including UDP. SSL requires merely replacing socket calls with SSL socket calls but IPsec can be added without changing applications at all. Even if the

application wishes to be IPsec aware, it can still use the basic socket calls without having to compensate for unusual I/O behavior due to SSL record framing. This is particularly convenient in situations where much of the network code is from a third party. Using SSL requires changing all of this code. IPsec requires changing essentially none of it.

Although IPsec does not require disrupting applications, it does require disrupting the operating system. The code for performing AH and ESP is necessarily part of the IP stack. In most systems, the stack is located in the operating system kernel. The ISAKMP/IKE code can be implemented in user space but must be able to create kernel table entries for SAs so that the AH/ESP code can access them. Thus, enabling IPsec can mean installing a whole new operating system.

Endpoint Authentication

Unlike SSL, ISAKMP requires that both sides authenticate. This is inconvenient because in many interactions the client identity isn't important. However, ISAKMP allows shared secret–based authentication whereas SSL does not. Note that these differences are simply design choices in the respective protocols. They're not inherent results of putting security at the transport or IP layers.

Intermediaries

We've seen that HTTPS can tunnel through proxies, but that SMTP over TLS must connect directly to relays and deliver the mail to them. The difference is due to the store-and-forward semantics of mail delivery. With IPsec, tunneling through intermediaries isn't even an option.

Recall that when we tunneled HTTPS through the proxy, we still had to make a TCP connection to the proxy. Thus, the proxy had to be able to read the TCP traffic—it was the application-layer traffic that it couldn't read. Because IPsec offers protection at the IP layer, this separation is no longer possible. The most that is possible is to offer IPsec to the intermediary. When application-level intermediaries are involved, IPsec can only offer link protection, whereas SSL can sometimes offer end-to-end protection.

Another approach to intermediaries would simply be for connections that are to be secured with IPsec to bypass the intermediaries entirely and connect directly to the target host. In general, however, this will not work. Firewall proxies are in place to control access, and punching holes—even for IPsec—negates this purpose. Moreover, in the case of SMTP, relays are a necessary part of the mail system and cannot be bypassed; the sending host doesn't know where to deliver the mail except to a relay.

Virtual Hosts

In Chapters 9 and 10, we saw that it was possible to use virtual hosts with SSL without IP aliasing, provided that we used an upward negotiation strategy that signaled the identity of the target host. With IPsec, however, because there is no application-layer

protocol data transmitted before the IPsec SAs are established, there's no such thing as upward negotiation. Thus, IP aliasing is the only workable approach to virtual hosting.

NAT

IPsec completely breaks when a router is performing *Network Address Translation* (NAT). This technique allows a large number of machines to use a small number of IP addresses. Each machine is assigned its own private IP address and then the router automatically translates the private IP numbers to one of the assigned public IP numbers on transmission and from the public numbers to private numbers on receipt.

Naturally, because IPsec depends on establishing SAs to specific hosts, identified by IP address, NAT causes substantial problems if you want to run IPsec from a machine behind the NAT router. (Although if only a small number of hosts were to use IPsec, the router could be programmed to give them fixed addresses.) By contrast, SSL is completely unaffected by NAT. There is wide (and often acrimonious) debate about whether NAT is a good or a bad idea ([Rekhter1994, Lear1994, Rekhter1996]), so this is either a bug or a feature depending on your perspective. Nevertheless, NAT is widely deployed and so it's problematic that hosts behind NAT routers can't use IPsec.

Bottom Line

The dominant argument against IPsec is the need to change the TCP/IP stack. The dominant argument for it is not needing to change applications in order to secure them. In practice, because most applications don't really *need* security and changing operating systems is very inconvenient, it's better to simply use SSL on an application-by-application basis. However, IPv6 requires security support, so if it is ever widely deployed in end systems, that would be a good time to add security to IPv4 as well. Also, Windows 2000 comes with built-in IPsec support which makes IPsec a very attractive choice for Windows 2000–only applications.

The most common use of IPsec is to create VPNs, which it does very well. Upgrading a single router is fairly straightforward, and IPsec-capable routers are widely available. Installing such a router on your network makes it easy to produce a secure connection to remote networks. Thus, any traffic from one network to the other is automatically encrypted by the gateway routers. This is by far the most straightforward way to create a VPN.

If we assume that in the future IPsec will be widely available on end-user machines, then we can expect its widespread use for opportunistic security. As we saw in the case of SMTP over TLS, it can be useful to opportunistically negotiate security even without any protection against active downgrading. This provides protection against passive sniffing attacks with almost no configuration or management overhead. Thus, while it's difficult to replace HTTPS with HTTP/IPsec, replacing SMTP/TLS with SMTP/IPsec is quite attractive.

11.11 Secure HTTP

Secure HTTP (S-HTTP) [Rescorla1999a] provides a syntax for securing messages sent using HTTP. Unlike HTTPS, which largely ignores the content and boundaries of individual HTTP messages, S-HTTP treats each HTTP request or response as a single unit and protects it individually. This allows S-HTTP to protect different messages between client and server differently as well as to offer message-level digital signature and nonrepudiation.

S-HTTP consists of two major components, a *message format* for encapsulating and protecting individual messages and a *negotiation syntax* which allows both client and server to express their opinions about how data should be protected and to provide keying material.

Full disclosure: I was one of the primary designers of Secure HTTP and for several years worked for a company that sold Secure HTTP toolkits. This gives me good perspective on the relative merits of Secure HTTP and SSL. However, it also introduces the suspicion of bias. Nevertheless, I've tried to present the pros and cons as objectively as I can.

Message Format

The S-HTTP message format is based on the *Cryptographic Message Syntax* (CMS), described in RFC 2630 [Housley1999b]. CMS is a variant of PKCS #7 [RSA1993c] designed for use with *Secure MIME* (S/MIME). In general, S-HTTP messages look like HTTP requests or responses, but the bodies are CMS messages. The protected body contains the actual HTTP request or response that the client or server sent. Thus, the headers and the message body are protected.

Cryptographic Options

One of the deliberate design goals of S-HTTP was that it have the same messaging model as HTTP. Thus, whereas SSL requires multiple round trips between client and server to perform the handshake, S-HTTP does not. Instead, Web pages containing S-HTTP links also contain negotiation information. This information is sufficient for the client to determine which protections to apply to a request generated by dereferencing a link—without first talking to the server. Similarly, S-HTTP clients put negotiation information in the headers of their requests. This information is sufficient to tell the server how to protect the response.

Our Approach

Our intent is to compare the HTTPS, discussed in Chapter 9, with S-HTTP. In order to do this, we need to understand how S-HTTP works well enough to appreciate the differences. Thus, we start with an overview of S-HTTP. First, we discuss CMS and then the format of S-HTTP messages. Then, we consider cryptographic options and the negotiation process. Finally, we discuss how negotiation and messaging work together to secure HTTP traffic and show an example.

Ex.1024
Askeladden, L.L.C. / Page 459 of 788

With a firm understanding of S-HTTP established, we're ready to discuss the advantages and disadvantages with regard to HTTPS. In general, S-HTTP's message orientation provides greater flexibility than HTTPS. It is able to provide message-level signature and nonrepudiation, which HTTPS cannot. It also interacts more cleanly with proxies and virtual hosts. The primary disadvantage of S-HTTP is that it is substantially more complicated to implement, both on clients and servers.

11.12 CMS

CMS is a fairly typical protocol for cryptographically securing messages. It provides encryption and signature for arbitrary content. Every CMS message has a type that describes the form of cryptographic enhancement that has been applied. CMS messages may be recursively encapsulated in order to apply multiple types of enhancement. CMS defines six basic types, Data, SignedData, EnvelopedData, EncryptedData, and DigestedData. However, S-HTTP uses only SignedData and EnvelopedData, so we'll talk only about them.

SignedData

A SignedData message consists of some content and one or more signatures over that content. As usual, the content is digested and then the digest is signed with the sender's public key. The message can—and usually does—contain the appropriate certificates needed to certify the key(s) used to sign the message. The message can also contain CRLs needed to check the validity of the certificates.

It's also possible to have a SignedData message with no content. This is called a *detached signature* and represents a signature over some data which must be externally specified.

EnvelopedData

An EnvelopedData message contains encrypted data. The data is encrypted with a randomly generated symmetric *content encryption key* (CEK). The encrypted content is encapsulated in a wrapper that contains one or more RecipientInfo blocks. Each RecipientInfo block contains the CEK encrypted for a given recipient.

The CEK may be encrypted in any of three ways. If RSA is used, the CEK is simply encrypted under the recipient's public key. If DH is used, the CEK is encrypted under the sender and recipient's pairwise DH shared secret. Note that you don't encrypt the message directly under the pairwise DH shared secret because this would require reencrypting the entire message for each recipient. Encrypting the CEK under the pairwise secret allows the sender to just generate a new wrapped CEK for each recipient. Finally, it is possible to simply encrypt the CEK under a symmetric *key encryption key* (KEK) shared by sender and recipient. How such a key would be shared is outside the scope of

CMS. However, S-HTTP provides a way of establishing such keys, as we discuss in Section 11.14.

Signing and Encrypting

CMS does not provide any content type that provides both signature and encryption. To apply both enhancements, we must use recursively enhanced content. Thus, we take the data and sign it to produce a SignedData. Then we take the SignedData and use it as input to the encryption process to produce an EnvelopedData. Reading such a message simply reverses this procedure.

11.13 Message Format

Unlike HTTPS, S-HTTP messages use the same port as HTTP messages. They are designed so that a server can easily distinguish an S-HTTP request and remove the enhancements before processing the request. In order to do this, S-HTTP uses a special request method: `Secure`. Because the Request-URI might contain sensitive information, it is replaced with a "*". Figure 11.6 shows a typical S-HTTP request.

```
Secure * Secure-HTTP/1.4
Content-Type: message/http
Content-Privacy-Domain: CMS

Binary CMS message deleted
```

Figure 11.6 An S-HTTP message

The `Content-Privacy-Domain: CMS` line indicates that the (deleted) request body is a CMS message. This is necessary because S-HTTP also supports another (less popular) message security mechanism called *MIME Object Security Services* (MOSS). When the CMS message is unwrapped, its contents are the HTTP request that the client would have sent if S-HTTP were not being used. (This is what `Content-Type: message/http` means.)

S-HTTP responses follow a similar format to that of S-HTTP requests. Because the success or failure of the response is potentially sensitive, we wrap the entire HTTP response; the status line always reads

```
Secure-HTTP/1.4 200 OK
```

S-HTTP was originally designed using PKCS #7 and then adapted to CMS. Although CMS has suport for symmetric message authentication using MACs, PKCS #7 did not. Thus, S-HTTP also provides a `MAC-Info` header line which allows a MAC to be attached to the message.

11.14 Cryptographic Options

As we discussed in Chapter 9, HTTPS URLs can be distinguished from HTTP URLs by the `https` scheme. However, this is the only information provided to clients. The determination of which cipher suites to use is made during the SSL handshake. S-HTTP also has its own scheme: `shttp`. However, most of the information is given in cryptographic options. These options indicate what sorts of transformations the recipient expects the sender to use when sending a message.

The server's options are usually placed in the associated HTML anchor. RFC 2659 [Rescorla1999b] describes the procedure. The client's options are placed in the inner header—the HTTP header, not the S-HTTP header—of the request. When a client dereferences a URL, it first finds the corresponding options and uses them to determine what kind of message to send. When the server sends a response, it first examines the client's options in the HTTP header of the client's request.

S-HTTP doesn't have any real negotiation. Rather, the sender merges his own preferences with the receiver's preferences to come up with a jointly acceptable set of enhancements. He then applies those enhancements to the message and sends it. Except for the location of the cryptographic options, the client and server behave identically in this respect.

The cryptographic options contain two different kinds of information: *keying material* and *negotiation headers*. Keying material includes keys, certificates, etc., to be used when protecting a message. Negotiation headers contain preferences for what sorts of protection to use, including algorithms and key lengths.

Keying Material

The keying material options allow one side to send another side keying material to be used to respond to that message. Thus, a server could include its certificate in a link on a HTML page so that a client could use the certificate to encrypt its request for that link. S-HTTP also allows the sharing of a symmetric encryption key using the `Key-Assign` line. Because RSA encryption is expensive, it's better if any client and server pair only do it once. Thus, if a client encrypts to a server, it can provide the server with a symmetric key which the server can then use as a KEK to encrypt the CEK for its response.

Negotiation Headers

In SSL, all of the possible enhancement options come as a package: key exchange algorithm, authentication algorithm, encryption algorithm, and message authentication algorithm are all defined by the cipher suite. By contrast, S-HTTP has rather more options and they are specified more or less orthogonally.

S-HTTP allows negotiation of three different privacy enhancements: digital signature over the content (`sign`), encryption of the content (`encrypt`), and symmetric message authentication of the content (`auth`). These enhancements may be present in any combination, although the presence of `auth` and `sign` together seems redundant.

For each of these enhancements, the algorithm may be negotiated. Additionally, it is possible to negotiate key exchange algorithms (public key versus symmetric key) independently of content encryption algorithms. It is also possible to provide information about what sort of key the sender should sign with if digital signature is to be applied.

Finally, the negotiation headers can contain information about which enhancements an agent will use when sending. Thus, a server can tell a client what sort of message to expect in response to its request. The client should then compare the server's actual response to the behavior promised in the cryptographic options. Figure 11.7 shows an example of a single negotiation header sent by a server.

```
SHTTP-Symmetric-Content-Algorithms: orig-optional=DES-CBC, DES-EDE3-CBC;
     recv-require=DES-EDE3-CBC
```

Figure 11.7 A negotiation header

The header line in Figure 11.7 says that this link must be dereferenced with 3DES (`recv-require=DES-EDE3-CBC`) but that the server is willing to respond with either DES or 3DES (`orig-optional=DES-CBC, DES-EDE3-CBC`).

S-HTTP defines eight such negotiable items, so the cryptographic options can get rather long. To alleviate this problem, RFC 2660 provides defaults. If the value of an option matches the default, then that option may be omitted.

All in all, S-HTTP allows negotiation of the following parameters:

- Message format—CMS or MOSS
- Certificate type—X509, PKCS7 wrapped
- Key exchange algorithm—DH, RSA, Inband, Outband
- Signature algorithm—RSA, DSS
- Message digest algorithm—MD5, SHA-1, HMAC
- Symmetric content encryption algorithm—DES, 3DES, DESX, CDMF, IDEA
- Symmetric key encryption algorithm—DES, 3DES, DESX, CDMF, IDEA
- Privacy enhancements—signature, encryption, authentication, or some combination

Note that there are no specifications for use of DES, DESX, or CDMF with CMS. However, there's really only one obvious way to plug in new content encryption algorithms. By advertising support for these algorithms, S-HTTP implementations imply that they support this approach.

11.15 Putting It All Together: S-HTTP

The easiest way to see how S-HTTP works is to see an example. Figure 11.8 shows a page with a Secure-HTTP link on it. How the client got this page is irrelevant, but likely it was fetched via HTTP.

```
<CERTS FMT=PKCS-7>
Certificate deleted
</CERTS>

<A DN="CN=Test Server, O=RTFM, Inc., C=US"
   CRYPTOPTS="
     SHTTP-Privacy-Enhancements: recv-required=encrypt;
     SHTTP-Key-Exchange-Algorithms: recv-required=RSA;
     SHTTP-Symmetric-Content-Algorithms: recv-required=DES-EDE3-CBC"
   HREF="shttp://www.rtfm.com/test.html">
   Click here to dereference</A>
```

Figure 11.8 A page with an S-HTTP link

The `CERTS` element contains a base64-encoded PKCS #7 certificate chain. We've deleted it from this example. The anchor contains the relevant cryptographic options. The `DN` field contains the distinguished name of the server. Note that the server's certificate and distinguished name are provided separately. The `CERTS` element is just advisory. The `DN` attribute is what's used to find the server's key.

The `CRYPTOPTS` attribute contains the negotiation headers. In this case, the server requires the client to encrypt, using RSA for key exchange and 3DES for symmetric encryption. Finally, the `HREF` attribute contains the URL of the resource, as usual, but with an `shttp` scheme.

When the client dereferences the link, it encrypts the request, using RSA for key exchange and 3DES for encryption, as required. It also provides a symmetric key to the server using the `Key-Assign` header. The encrypted request is shown in Figure 11.9. This request (and all the messages shown throughout the rest of the chapter) are shown in *crypto-vision*. Rather than displaying the encrypted content as a blob of binary data, we show the inner content that you would see if you decrypted it. This content is presented in a *`fixed-width italic`* font to differentiate it from the rest of the message.

Note that the headers that would ordinarily have been in the HTTP request, such as `Security-Scheme` and `User-Agent` are in the inner encrypted request. Again, these headers are potentially sensitive, so they must be protected.

The last four header lines are the S-HTTP negotiation headers. The `Key-Assign` line provides an *inband* key. This key can be used to allow the server to respond with an encrypted message without performing another RSA operation. It also allows the client to receive encrypted messages without having a private key, just as it can with SSL. The client should generate a fresh key for each request. This key has the label `1`.

The last three header lines tell the server that it must encrypt its response using 3DES. It must wrap the CEK using the inband key passed in this request, using 3DES. It must also provide a MAC over the entire message.

Finally, Figure 11.10 shows the server's response to this message, once again in crypto-vision. As instructed, the server has encrypted the message for the client with the key `inband:1` as the CEK. This is hidden in the CMS wrapper. The server has also

```
Secure * Secure-HTTP/1.4
Content-Type: message/http
Content-Privacy-Domain: CMS

GET /secret HTTP/1.0
Security-Scheme: S-HTTP/1.4
User-Agent: Web-O-Vision 1.2beta
Accept: *.*
Key-Assign: Inband,1,reply,des-ecb,des-ede3-ecb;
     7878787878787878787878787878787878787878787878
SHTTP-Privacy-Enhancements: recv-required=encrypt,auth
SHTTP-Symmetric-Content-Algorithms: recv-required=DES-EDE3-CBC
SHTTP-Key-Exchange-Algorithms: recv-required=inband
SHTTP-Symmetric-Header-Algorithms: recv-required=DES-EDE3-ECB
(blank line)
```

Figure 11.9 An S-HTTP request

used `inband:1` to compute the HMAC value found in the `MAC-Info` line. Note that the real HTTP headers are encrypted and MACed. They potentially contain sensitive data and so must be protected. Thus, it's possible to have a S-HTTP response where the S-HTTP status line is OK but the interior HTTP status line indicates an error.

```
Secure-HTTP/1.4 200 OK
Content-Type: message/http
MAC-Info:31ff8122,rsa-sha-hmac,
     a51d612e5f3d0fbb3d8837ca351fcec879e5e3a6,inband:1
Content-Privacy-Domain: CMS

HTTP/1.0 200 OK
Security-Scheme: S-HTTP/1.4
Content-Type: text/html

Congratulations, you've won.
```

Figure 11.10 An S-HTTP response

Client Authentication

S-HTTP authenticates the client by authenticating the request message. There are two options for authenticating a message: digital signature over the message and MAC. Digital signature provides for nonrepudiation whereas a MAC merely provides a fast

integrity and sender-authentication check. The digital signature binds the message's sender to the sender's certificate. The MAC allows the recipient to know that the message was sent by someone who has the MAC key.

Reference Integrity

Preserving S-HTTP reference integrity is very simple. Because the page containing the reference also contains the cryptographic information required to encrypt the request, the client can simply use that information. There is thus no requirement that the server's certificate actually matches the server's DNS name. All that is required is to check that the server's certificate verifies. The client knows the server's identity from its certificate.

Because each message is individually authenticated, we need to provide some way of binding the reference to the server's response. Otherwise, the attacker could replay an old response to the client. The way to do this is for the client to provide a key using `Key-Assign` and then require the server to encrypt or MAC its response using that key. Knowing the key depends on being able to decrypt the original request, which implies knowing the server's private key. Thus, the response is bound to the request.

Automatic Options Generation

The most notable feature of S-HTTP is that any given request, even for resources on the same site, might use a different set of cryptographic enhancements. Although this provides tremendous flexibility, it also presents a substantial burden on the administrator. Even with defaults, S-HTTP negotiation headers are difficult to write manually. Worse yet, writing the options into the HTML files means that if your policies ever change you have to edit all the pages with references to the affected resources.

A better approach is to have the server automatically create the options. To do this requires that the server have a primitive HTML parser capable of identifying links. When a link is found, the server then finds the access control policy associated with the resource and automatically emits the appropriate options along with that link. Unfortunately, early S-HTTP servers did not support this feature, making them very hard to administer.

Note the `Security-Scheme` header field in Figure 11.9. This is a new HTTP header added by S-HTTP. It allows a client to tell a server that it is S-HTTP capable. This allows the server to rewrite only pages destined for S-HTTP-capable browsers. Although S-HTTP's modifications to HTML are intended to be invisible when viewed in HTML compliant browsers, some browsers nevertheless displayed them as part of the page. This header prevented pages containing S-HTTP links from containing extra clutter in non-S-HTTP browsers.

Stateless Operation

We've shown how a client can use the `Key-Assign` header to provide the server with a symmetric encryption key for the server's response. Obviously, the server could do the

same thing in its cryptographic options, but this would ordinarily require it to remember the key. Because we have to contend with multiple server processes, this means arranging to communicate this key between them. Recall that we faced a similar problem with SSL session caching.

With S-HTTP, however, this problem can be attacked without creating server state. The `Key-Assign` header allows the sender to assign the key an arbitrary name. We can leverage this name to avoid storing temporary keys. All of our server processes share a single *master* CEK. Then, when we create a new temporary key for use with `Key-Assign`, we arrange that we can generate that key from the label. The simplest way to do this is to simply encrypt the key with the master CEK and use that as the label.

$$Label = E(MasterCEK, Key)$$

Alternatively, we can randomly generate the label and use that to generate the key, like so:

$$Key = HMAC(MasterCEK, Label)$$

We could imagine using such a trick with SSL by using the encrypted master secret as the session ID. Unfortunately, this is impossible for two reasons. First, the session ID would need to be as long as the master secret, but it's only 32 bytes and the master secret is 48 bytes. Second, SSL requires us not to resume sessions that aren't closed properly. This requires us to be able to invalidate sessions, which would require interprocess communication. Note, however, that invalidating sessions doesn't really add that much security and that many implementations don't do it in any case. The major obstacle is that the session ID is too short.

11.16 S-HTTP versus HTTPS

In general, the closer to the application-layer protocol that you provide security, the more flexible and powerful security the services you are able to offer. However, this flexibility often comes at the cost of complexity and implementation effort. Thus, although S-HTTP is significantly more flexible than HTTPS, and interacts better with the Web environment, it's also significantly harder to implement and deploy.

Flexibility

The most noticeable difference between HTTPS and S-HTTP is that S-HTTP is far more flexible than HTTPS. In principle, every resource on an S-HTTP-enabled Web server can require a different set of cryptographic enhancements in order to be fetched. In practice, this much flexibility is rarely required, however it's quite common to want three to five policies. HTTPS has difficulty handling even this many policies and usually needs to resort to clumsy hacks like virtual hosts or multiple ports (one for each policy). As we saw in Chapter 9, requiring client authentication on only part of an HTTPS Web

site requires an entire rehandshake for those pages. With S-HTTP it's a simple matter of changing the options for the appropriate links.

However, S-HTTP's flexibility comes at a high price. Having to make page-by-page decisions is daunting for administrators. This administrative burden can be minimized with good server design and user interface, but this requires substantially more programming than writing an HTTPS-enabled server.

Nonrepudiation

One S-HTTP feature that cannot be achieved with HTTPS is nonrepudiation. Since SSL knows nothing about the various HTTP message boundaries and uses MACs for message authentication, there is no way to provide proof that a given request or response was made. By contrast, because S-HTTP provides whole-message signature, this is a simple matter of signing the appropriate message. It's equally easy for the client and server to request this service by setting the appropriate option.

The signing feature of S-HTTP can also be used to allow servers to cache presigned documents. This allows the server to execute the signature once and simply hand out the same document repeatedly to multiple clients. This can be useful for static objects that need to be integrity protected but do not need to be secret.

Proxies

In the simplest case, S-HTTP proxies can behave exactly like HTTPS `CONNECT` proxies. However, because the message structure of S-HTTP is apparent to the proxy, a more sophisticated proxy can take advantage of this information. For instance, a proxy could notice where message boundaries are and shut down idle connections. A really sophisticated proxy can cache responses for static, unencrypted documents fetched with S-HTTP. Thus, it's possible for S-HTTP proxies to cache signed content such as programs or long-lived data files.

Virtual Hosts

S-HTTP's interaction with virtual hosts is simple. Because S-HTTP provides the DN of the server in the cryptographic options, there is no need to use different certificates for different virtual hosts. Even if this is desired, because the CMS message clearly indicates the DN of the recipient, the server can easily decrypt the message. Once the message is decrypted, the server has access to the `Host` header and can act accordingly.

User Experience

One drawback with the strategy of placing cryptographic options in anchors is that the URL now becomes insufficient to fetch a resource. The client also needs the cryptographic options from the anchor. Thus, it's possible for a user to type an `https:` URL into a browser directly, but a `shttp:` URL is much less useful. RFC 2660 describes a

hack whereby the browser may ask the server for some generic cryptographic options, but because the server doesn't know what resource is being requested, S-HTTP is no more flexible than HTTPS in this case.

When S-HTTP and HTTPS were first introduced, the insufficiency of `shttp:` URLs seemed like a major drawback. Users are quite commonly in the habit of typing URLs directly into the browser and if the entire Web was to be secured then it was assumed that they would want to type in secure URLs. However, it turns out that most Web sites use security for only a very limited portion of the site, using `https:` links to transition from the insecure to secure portions. Thus, because the URL is nearly always found in an HTML page, there is a location to put the cryptographic options.

Ease of Implementation

The difficulty of implementing S-HTTP and SSL is roughly comparable. However, the difficulty of integrating them into preexisting software is not. We've already mentioned that S-HTTP requires extensive support from the server in order to rewrite HTML pages to insert cryptographic options. None of this effort is required with HTTPS. Note, however, that this means that Web designers need to explicitly edit the HTML to make resources accessible via HTTPS. With a rewriting strategy, such as should be employed with S-HTTP the administrator can change only the access control settings and the server will automatically convert pages to secure access as appropriate.

S-HTTP also requires substantially more support on the client side than does HTTPS. Primarily, the HTML parser needs to be modified in order to allow the cryptographic options to be extracted. None of this work is necessary with HTTPS. Also, because S-HTTP allows more operational modes, the user interface should reflect exactly what sorts of processing a given message had, whereas with HTTPS it's more realistic to simply indicate *secure* or *insecure*.

Whether this is a bug or a feature depends on your perspective. The reason it's insufficient to say *secure* or *insecure* with S-HTTP is that S-HTTP has a far more flexible security model. Thus, if you think that the additional features that S-HTTP adds are worth having, then you need some way to represent them to the user. On the other hand, if you think the features are unnecessary, then they just make extra user interface effort.

In sum, implementing S-HTTP requires substantially more programming effort than does HTTPS. Moreover, this programming effort cannot be isolated into a toolkit but requires tight integration with the browser and server. This difference isn't surprising, because S-HTTP is far more tightly integrated with HTML and HTTP than HTTPS.

Bottom Line

In terms of technical merits, S-HTTP's greater flexibility and tight integration with the Web is both a cost and a benefit. It allows S-HTTP to offer features that HTTPS cannot, such as nonrepudiation and pre-signed data. It also means that S-HTTP's interaction with virtual hosts and proxies is far cleaner. However, it also means that implementing S-HTTP is far harder than implementing HTTPS.

Ex.1024
Askeladden, L.L.C. / Page 469 of 788

Moreover, the subtleties involved in implementing HTTPS involve making it totally secure. The subtleties involved in implementing S-HTTP involve making it usable. Thus, a naive HTTPS implementation might fail to deal with closure or reference integrity properly. A naive S-HTTP implementation might require the administrator to manually edit HTML documents in a particularly unpleasant way. Because users tend to value convenience over security, a naive S-HTTP implementation is much less valuable than a naive HTTPS implementation.

In 1995 and 1996, there was substantial doubt as to whether HTTPS or S-HTTP would become the dominant Web security protocol. There is no longer any doubt; HTTPS is the undisputed winner. Whatever their respective technical merits, every major browser and server implements HTTPS whereas none implement S-HTTP. Thus, there isn't really a choice of whether to use S-HTTP or HTTPS.

11.17 S/MIME

S/MIME provides message-oriented security services for e-mail messages. Before the Web, e-mail was—and arguably still is—the most important networking service. Thus, many attempts have been made to standardize it. In the IETF alone, there have been no less than four e-mail security standards: PEM, MOSS, OpenPGP, and S/MIME. All of these protocols take essentially the same approach. They treat an e-mail message as a single object and provide security services for that object. The differences between them are largely details of message format and trust model.

S/MIME version 2 [Dusse1998] was originally developed by RSA Labs and used RSA's PKCS #7 [RSA1993c] cryptographic messaging format. PKCS #7 supported only RSA for key exchange and thus S/MIME v2 only had support for RSA. When the IETF S/MIME working group was created to standardize S/MIME v3 [Ramsdell1999], one of the primary goals was to add support for other algorithms. Thus, PKCS #7 was revised to create CMS [Housley1999b], which also supported other key exchange and signature algorithms.

Our Approach

Our intent is to compare securing e-mail transport via SMTP/TLS with S/MIME. As with S-HTTP, we'll first present enough detail to explain how S/MIME works. We've already covered CMS in Section 11.12, but there are some subtleties in S/MIME's use of CMS, particularly when data is signed but not encrypted. Finally, we'll cover S/MIME algorithm selection; because e-mail is store-and-forward, ensuring that the sender knows which algorithm to use takes some special thought.

With our knowledge of S/MIME in hand, we then discuss its advantages and disadvantages over SMTP over TLS. As we discussed in Chapter 10, SMTP over TLS really doesn't do that good a job of securing e-mail. S/MIME, however, does an excellent job. The only real advantage that SMTP over TLS has is that it doesn't require as much client-side support.

11.18 Basic S/MIME Formatting

MIME allows e-mail messages to carry arbitrary content through the use of a `Content-Type` line. S/MIME uses the `application/pkcs7-mime` content type to indicate that the content type is a CMS-encapsulated MIME message. That is to say, when the CMS body is unwrapped, it itself contains a MIME body. Figure 11.11 shows such an example, once again in crypto-vision.

```
From: ekr@rtfm.com
Subject:Test message
Content-Type: application/pkcs7-mime; smime-type=enveloped-data

Content-Type: text/plain

This is an encrypted message.
```

Figure 11.11 An encrypted S/MIME message

Note that the `Subject` line of the message appears in the unencrypted portion. Thus, the `Subject` is not cryptographically protected in any way. In fact, none of the normal mail headers are. If they are to be protected, they must appear in the header section of the inner content.

11.19 Signing Only

Figure 11.11 showed an encrypted message. The simplest way to prepare a signed message would be to do exactly the same thing as before, except using the CMS SignedData type. Of course, the `smime-type` parameter would now be `signed-data`. This approach is simple but has a substantial drawback. The message can't be read by a non-S/MIME-capable client. An encrypted message by definition requires an encryption-capable client to read it. This is not the case with a carefully structured signed message.

If a message is signed but not encrypted, then it's desirable for it to be readable by recipients without S/MIME-capable clients. This would encourage senders to sign messages whether or not the recipient was S/MIME capable. One could also send signed messages to multiple recipients—a useful feature when some of them can verify the message but it must be readable by all.

However, simply presenting the message as a CMS SignedData using `application/pkcs7-mime` defeats this purpose. Most MIME readers simply save content with unrecognized types to disk. Moreover, even if the reader attempts to view the data, since CMS is a binary format, it's very inconvenient to distinguish the CMS wrapper from the data. Often, standard tools will not display binary data at all.

Multipart/Signed

The solution to this problem is to make use of CMS detached signatures. We also need to use a different wrapping method, `multipart/signed`. As well as arbitrary data content types, MIME allows multiple content types to exist in a single message, using the `multipart` types. A `multipart/signed` message contains two parts. The first part is the actual object as it would have appeared if it weren't S/MIME wrapped. The second is a CMS detached signature over the message. Figure 11.12 shows an example of this.

```
Content-Type: multipart/signed; protocol="application/pkcs7-signature";
   micalg=sha1; boundary=boundaryYYY

--boundaryYYY
Content-Type: text/plain

This is a clear-signed message.
--boundaryYYY
Content-Type: application/pkcs7-signature
Content-Transfer-Encoding: base64

Signature deleted
--boundaryYYY--
```

Figure 11.12 A multipart/signed message

The `--boundaryYYY` lines indicate the breaks between various body parts. Note that any MIME-compliant agent can notice that the first body part is of type `text/plain` and display it to the user. It can either ignore the second body part or offer to save it to disk. In either case the user will be able to read the signed message without the signature getting in the way. Even if the user does not have a MIME-compliant mailer, he will have a much better chance of viewing the content if the signed data is separated from the binary clutter of the CMS wrapper.

S/MIME and S-HTTP

S/MIME encapsulation provides essentially the same set of features as S-HTTP's message format. The reason that S-HTTP provides its own message format is that it was designed before S/MIME. It would be perfectly possible to use S/MIME encapsulation with S-HTTP negotiation, and this would be the best choice if S-HTTP were being designed today.

11.20 Algorithm Choice

Because of the store-and-forward nature of e-mail, it's not possible for the sender and recipient to negotiate algorithms or keys. In principle, it's possible that the sender and recipient have never communicated before. The sender needs only to have the recipient's public key, which he could have gotten from a directory.

However, when the first message has been sent, the sender has an opportunity to communicate to the recipient what sorts of enhancements he would like the recipient to use when he replies. This is particularly important in order to ensure interoperation between export and domestic clients. Thus, S/MIME signed messages can include a *capabilities* attribute which indicate the signer's preferences.

Capabilities

The SMIMECapabilities attribute is used to indicate which cryptographic algorithms the sender supports. It's simply a list of the algorithms that the sender wishes to advertise. It can include at minimum the following types of algorithm:

- signature algorithms
- symmetric encryption algorithms
- key encipherment algorithms (how the CEK is encrypted)

The capabilities attribute is a fairly general mechanism for advertising a user's preferences. Algorithms are intended to be listed in order of preference so a user can describe not only what he supports but the order in which he wants it used. It's even possible to represent that you would prefer people to sign the messages they sent you, using the preferSignedData capability. Implementations are supposed to ignore capability identifiers they don't recognize, so the set can be extended in the future.

Algorithm Choice

The only algorithm choice that we really have to be concerned with is the encryption algorithm. The key establishment algorithm is defined by the recipient's key, so there is no choice to be made there, except in the unlikely case where the recipient has multiple keys—in which case the sender can choose either one. Theoretically, there might be an issue of which digest algorithm to use, but in practice support for SHA-1 is nearly universal. However, there is substantial variation in encryption algorithm support. In particular, there are a large number of exportable clients that support only RC2-40. For obvious reasons, it's important to encrypt your message with an algorithm the recipient supports. RFC 2633 details three cases.

Known capabilities. In the case where a sender knows the recipient's capabilities, the situation is simple. It should choose one of the recipient's supported algorithms. As a matter of courtesy, the RFC 2633 says that it should use the recipient's most preferred cipher. However, a sender might choose to use a cipher that the recipient liked less but the sender liked more. If a directory exists, then it is quite likely that the sender would be able to get the recipient's capabilities when it fetches the certificate.

Unknown capabilities, known use of encryption. In the case where the sender has received an encrypted message from the recipient but doesn't know any capabilities, then it's fairly safe to use the same encryption algorithm as the recipient used in that message.

Unknown capabilities. S/MIME v3 required support for 3DES, so if the sender knows that the recipient supports S/MIME v3, then it's safest to send using 3DES. However, S/MIME v2 required support for RC2-40 and many export clients supported only that, so absolutely maximal interoperability (at a tremendous security cost) can be achieved by using RC2-40. In general, RC2-40 is so weak that it's generally best to use 3DES.

Capability Discovery

The result of these rules is that over the course of several e-mail exchanges, the parties develop an understanding of their respective capabilities and a sort of algorithm negotiation takes place. Consider the case where we have two parties, Alice and Bob. They both have RSA keys and support 3DES, but neither knows the other's capabilities. The first three messages they send are shown in Figure 11.13.

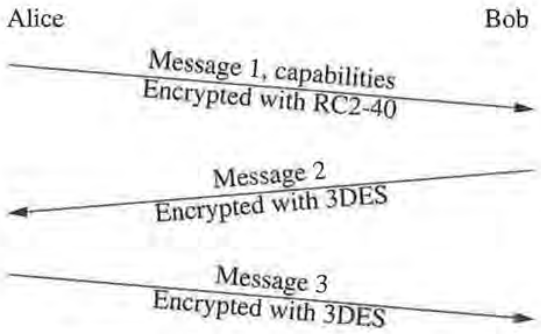


Figure 11.13 Capability discovery with S/MIME

Alice wants to be absolutely sure that Bob can read the first message she sends, so she sends it using RC2-40. (She could also send it in the clear.) She supports 3DES, so she

also includes a capabilities attribute indicating that she does. Thus, when Bob replies, he encrypts with 3DES. Now, he doesn't even have to include capabilities for Alice to know that he supports 3DES—he just used it—so Alice's final message also uses 3DES.

11.21 Putting It All Together: S/MIME

Now that we've seen the various components of S/MIME, let's examine how they work together to provide a coherent set of security services. Imagine that Alice wants to send a message to Bob. She already has his certificate and his capabilities, presumably by obtaining them from some directory.

Endpoint Identification

Alice has fetched Bob's certificate from the directory and verified it, but how does she know that it really is Bob's certificate? Recall that with SSL we often faced the problem of comparing a certificate to a DNS name. Here, we have a similar problem but we want to compare to an e-mail address. The solution is also similar: certificates can contain e-mail addresses either as a portion of the distinguished name or in the emailAddress value of the subjectAltName extension. Thus, Bob's certificate is the one that contains Bob's e-mail address in one of these locations.

Message Sending

With Bob's certificate in hand, generating a message that can only be read by Bob is simple. Alice encrypts the message under a randomly generated CEK and encrypts the CEK under Bob's public key, as usual. Optionally, she signs the message before encryption. She then transmits it—over any channel at all—to Bob. At this point it's perfectly safe to use normal unencrypted SMTP.

Sender Authentication

Once Bob receives the message, he wants to be able to do two things: read the message and authenticate the message and its sender. Reading the message is trivial. As usual, he simply decrypts the CEK using his private key and the message using the CEK. Naturally, all this is done automatically by his MUA.

Verifying the message signature is equally straightforward. First, Bob verifies the cryptographic identity of the message. However, as we saw in Chapter 10, e-mail messages also contain non-cryptographic indicators of the message sender in the `From` header. Because this header is under the control of the sender, the receiver's MUA must verify that it matches the certificate that was used to sign the message. If it doesn't, the MUA must somehow inform the user to prevent him from being fooled.

Multiple Signers

In certain circumstances, it's desirable to have a message signed by multiple senders. Consider the case where a message is really being sent by two people. In an ordinary letter, they would both sign it. Similarly, with a digitally signed message, we'd like both to sign it. This is trivial with S/MIME, even if the signers are on totally separate networks and computers, because CMS supports multiple signatures on a single message.

Multiple Recipients

Even more common than multiple signers is multiple recipients. It's very common to want to send a message to a number of recipients, while encrypting to all of them. This is very efficient in S/MIME. One simply encrypts the message with a single CEK and then encrypts the CEK individually for each recipient in a separate RecipientInfo structure. Thus, the message needs to be encrypted only once, saving both CPU time and bandwidth. For large mailing lists, S/MIME even includes a feature allowing the whole mailing list to share a single symmetric *mailing list key* under which the CEK is encrypted.

Receipts

One nice feature offered by S/MIME is the ability to provide a receipt, as described in RFC 2634 [Hoffman1999b]. Receipts allow a sender to be sure that the recipient received a message and that it was undamaged. A sender can use an attribute to indicate that he would like a receipt from the recipient. When the recipient receives the message, he generates a signed response containing the receipt. Note that the sender cannot force the recipient to generate a receipt. It requires the cooperation of the recipient. Although the recipient's software should generate the receipt automatically, this feature can be bypassed. Thus, although a receipt proves that the recipient received the mail, the lack of a receipt is not evidence that he did not.

11.22 Implementation Barriers

Being able to use S/MIME requires having three things: an S/MIME-capable client, a certificate, and access to the certificates of others. S/MIME-capable clients are becoming ubiquitous; S/MIME is no more difficult to implement than SSL, and many of the browsers that support SSL now support S/MIME, including Netscape Communicator and Microsoft's Outlook Express. As of this writing, OpenSSL even has some S/MIME support. However, the requirements for certificates and certificate access are more problematic.

Certificates

Certificates are fairly widely available from commercial CAs. However, S/MIME differs fundamentally from any system we've discussed so far in that it requires end users—not

just servers—to have certificates. This is a significant entry barrier because the number of users far exceeds the number of servers. Thus, to date, deployment of certificates has been slow.

One problem with certificate deployment is that the certificates need to vouch for the e-mail addresses of users. This presents a social problem because it's not clear who can vouch for the claim that a user has a particular e-mail address. Clearly the user can't be trusted. Theoretically the owner of the domain in which the e-mail address lives should vouch for the user's identity, but this requires the system administrator to be involved, which is highly inconvenient.

A weak but fairly common approach is for the CA to force the the user to prove that he can receive mail at a given e-mail address. Typically, the CA e-mails the user a random string which must be returned in order to get a certificate. This protocol is obviously subject to various kinds of attack, including simple sniffing, but because it has to be done only once for a given user, it's not completely worthless. Nevertheless, better certificate issuance procedures, probably rooted in Secure DNS, are needed.

Certificate Discovery

The second deployment problem is obtaining the certificates of others. It's obviously impossible to send encrypted mail to someone unless you have his certificate. In a perfect world, certificates would be available from a directory service, but despite some deployment, directories are far from ubiquitous.

In the absence of global directories, the best strategy is to sign every message one sends. Thus, anyone who receives a message from you will have your certificate and be able to encrypt to you. When you receive a signed message, you remember the e-mail address to certificate mapping. Thus, you can send encrypted e-mail to anyone from whom you've ever received e-mail. Over time, most people you communicate with will have your certificate and can send you encrypted e-mail. Until directories are ubiquitous, local caching is probably the best we can do.

11.23 S/MIME versus SMTP/TLS

In Chapter 10, we saw that SMTP/TLS wasn't very good at securing e-mail. By contrast, S/MIME does an excellent job. This is largely due to approaching the problem at the right layer of abstraction, the sender and recipient MUAs.

End-to-End Security

S/MIME provides end-to-end security. Messages are encrypted and signed by the sender and decrypted and verified by the recipient. They transit the network in a protected form. By contrast, with SMTP/TLS, we were often unable to protect the "last hop" link between the mail client and its local server because these links didn't use SMTP at all but rather some other protocol such as MAPI or POP. With S/MIME this is no problem because the encrypted messages are simply MIME data and can use any transport.

Nonrepudiation

SSL and TLS are completely unable to provide nonrepudiation because they use MACs for message integrity. Because S/MIME uses digital signature and signs the whole message, signed S/MIME messages can be forwarded from one recipient to another while leaving the signature intact and verifiable.

Relays

Even if both the sender's and recipient's local mail servers support SMTP/TLS, if the mail has to pass through a non-TLS relay along the way, then the message won't be secured. By contrast, S/MIME doesn't require any of the mail servers to be security capable. It merely requires support for S/MIME in the end user programs.

S/MIME does require one sort of support from relays; they must not damage the content. Because the signature is over a specific representation of bytes, if the relays damage the byte stream, then the signature will no longer verify. SMTP-compliant relays do not cause this problem, but mail often passes through other kinds of relays. To some extent, S/MIME is hardened against known transformations, but it is still possible for a relay to damage the message beyond repair. However, such relays would surely not be SMTP/TLS compliant, so the situation for S/MIME is no worse than for SMTP/TLS in this case.

Virtual Hosts

S/MIME doesn't require any special support from the mail servers at all, so virtual hosts work perfectly. And because the sender and recipient certificates are bound to their addresses, not to any particular server, users can easily move from ISP to ISP without changing any of their credentials.

Bottom Line

In general, S/MIME is a far superior solution to SMTP/TLS. The only significant barriers to using S/MIME are deployment barriers. However, S/MIME-capable mail clients are becoming increasingly popular and certificates are now available if not ubiquitous. The only significant advantage that SMTP/TLS offers over S/MIME is the ability to opportunistically secure e-mail in transit. However, this need would ultimately be better met using IPsec.

11.24 Choosing the Appropriate Solution

Now that we've seen a number of alternative approaches, we have enough information to draw some more general conclusions about when SSL is appropriate and when we'd be better served by attacking the problem at some higher or lower layer.

In general, SSL works well when the communications model is simple. Communications model simplicity has a number of aspects, but in general what we mean is that the interaction should be as much like a direct TCP connection between the two end parties as possible. The more complicated the environment is, the more difficult providing adequate security with SSL becomes.

Direct Connection

Providing adequate security with SSL more or less requires having a direct connection between client and server. Many of the problems we encountered with SMTP occurred because we didn't have a direct connection. With HTTPS we were able to get adequate results only because we were able to simulate a direct connection by convincing the proxy to provide us with a tunnel.

Whenever we don't have a direct connection, an object security solution suits us better. S-HTTP interacts somewhat better than SSL with proxies and S/MIME is of course completely oblivious to mail relaying. In general, as long as you can avoid your objects being damaged by intermediaries, an object security protocol is a better choice.

TCP Only

Consider a protocol such as DNS that communicates via UDP. Because SSL requires a TCP channel, it's impossible to secure it with SSL. However, because IPsec lives at a lower layer in the protocol stack, it's capable of providing security for UDP data. Similarly, an object security protocol such as DNSSEC (described in RFC 2535 [Eastlake1999]) that secures the actual DNS data works fine with UDP. In this case, DNSSEC is a better solution because it doesn't require setting up an IPsec channel just to do name resolution.

Only Two Communicating Parties

Because SSL connections are inherently one-to-one, SSL works best if there are only two communicating parties. Object security tends to support many-to-many communications much more cleanly. Even when we can simulate a many-to-many interaction with SSL, it is usually far clumsier than the corresponding object security solution.

Consider the case of sending a message to a large number of recipients. With SSL, we need to reencrypt the message every time we send to a new person. With an object solution, we can simply rewrap the CEK for each new recipient. This saves both CPU time and bandwidth. If the message is simply signed, then we don't even need to reencrypt the CEK. We can simply transmit the same message to multiple recipients. This capability is especially important when we're repeatedly transmitting the same signed content, such as a signed software distribution. We can sign the content offline and then have the pre-signed file on some insecure server. Thus, even if the server is compromised, the attacker cannot create a fake distribution.

Simple Security Services

SSL works best when the security services to be provided are simple. Because it doesn't know anything about the objects being transmitted it can't offer services like nonrepudiation, timestamping, or return receipt. These services require an object security protocol, which is inherently an application-layer service.

11.25 Summary

SSL is a useful and flexible security tool, but it is not the only tool in our toolbox. It's important to understand our other alternatives in order to know when to choose the right one. This chapter has surveyed three other security protocols—IPsec for securing arbitrary network data, Secure HTTP for securing HTTP transactions, and S/MIME for securing e-mail messages—in order to provide this perspective.

The end-to-end argument requires that correct security for applications be provided by the application. Because HTTP and e-mail are application-layer protocols, security for them must be provided at the application layer. HTTPS mostly meets this test. SMTP/TLS does not, which accounts for many of its shortcomings.

Higher-level security protocols are more flexible. In accordance with the end-to-end argument, higher-level protocols are more capable of providing exactly the security that an application needs. In particular, object security protocols can provide advanced features such as nonrepudiation. In general, if it's not too much trouble, it's better to use a higher-level security mechanism.

Lower-level security protocols are more general. The lower in the protocol stack that we put security, the more traffic it can account for. Thus, whereas S-HTTP and S/MIME can secure only one service, SSL can secure anything running on TCP and IPsec can secure any TCP/IP traffic.

IPsec provides security for any network data. Unlike SSL, IPsec provides security for any traffic between two hosts. This means that once IPsec is installed, all applications gain some security. However, because IPsec is lower in the protocol stack than is SSL, it is even more sensitive to interference by intermediaries in the communications channel.

Secure-HTTP provides security for HTTP transactions. S-HTTP treats each HTTP request or response as an individual message. This allows more flexible negotiation of security properties as well as object security for the messages.

S/MIME provides security for e-mail messages. Each individual message is protected. This allows end-to-end message protection and fits well with the e-mail store-and-forward model. It also allows true sender and recipient authentication.

SSL works well with real-time TCP services. If the communicating parties are directly connected using TCP, then SSL works well. If they are using something other than TCP, then IPsec should probably be used. If they are not directly connected, then an application-level security protocol is more appropriate. SSL is a compromise between application security, which offers better protection, and IP security, which offers more generality.

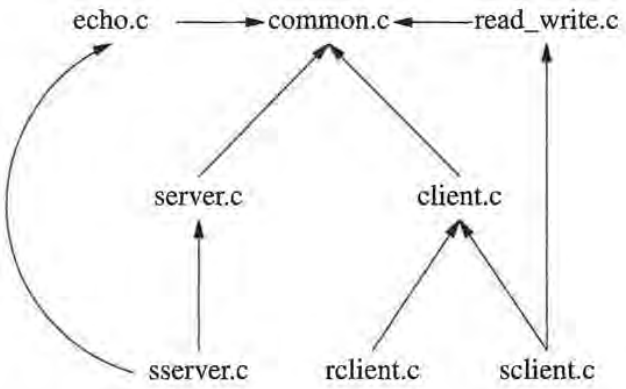
Appendix A

Example Code

A.1 Chapter 8

This section contains complete versions of the example programs from Chapter 8. Machine readable versions of these programs can also be downloaded from the author's Web site at `http://www.rtfm.com/sslbook/examples`.

A.1.1 C Examples



The above picture shows the dependency graph for the various C source files in the examples.

`common.c` provides a set of common utility functions including a common initializer for client and server and error exit routines.

Ex.1024
Askeladden, L.L.C. / Page 481 of 788

`server.c` provides server utility functions, mainly for initialization but also the `tcp_listen()` encapsulation for creating and binding sockets.

`client.c` provides our `tcp_connect()` wrapper for `connect()` and the `check_cert_chain()` verification routine.

`echo.c` contains the main server echo loop.

`sserver.c` contains the server main program which arranges to listen on a socket and then loops between `accept()` and `echo()`.

`read_write.c` contains multiplexed `select()`-based I/O handling for the client.

`sclient.c` is our main client program. It connects to the server and then uses `read_write()` to move data back and forth.

`rclient.c` is a stub client that demonstrates how to do session resumption with OpenSSL.

common.h

```
1   #ifndef _common_h
2   #define _common_h

3   #include <stdio.h>
4   #include <stdlib.h>
5   #include <errno.h>
6   #include <sys/types.h>
7   #include <sys/socket.h>
8   #include <netinet/in.h>
9   #include <netinet/tcp.h>
10  #include <netdb.h>
11  #include <fcntl.h>
12  #include <signal.h>

13  #include <openssl/ssl.h>

14  #define CA_LIST "root.pem"
15  #define HOST "localhost"
16  #define RANDOM  "random.pem"
17  #define PORT 4433
18  #define BUFSIZZ 1024

19  extern BIO *bio_err;
20  int berr_exit (char *string);
21  int err_exit(char *string);

22  SSL_CTX *initialize_ctx(char *keyfile, char *password);
23  void destroy_ctx(SSL_CTX *ctx);

24  #endif
```

common.h

common.c

```
1   #include "common.h"

2   BIO *bio_err=0;
3   static char *pass;
4   static int password_cb(char *buf,int num,int rwflag,void *userdata);
5   static void sigpipe_handle(int x);

6   /* A simple error and exit routine*/
7   int err_exit(string)
8     char *string;
9     {
10      fprintf(stderr,"%s\n",string);
11      exit(0);
12    }

13  /* Print SSL errors and exit*/
14  int berr_exit(string)
15    char *string;
16    {
17      BIO_printf(bio_err,"%s\n",string);
18      ERR_print_errors(bio_err);
19      exit(0);
20    }

21  /*The password code is not thread safe*/
22  static int password_cb(char *buf,int num,int rwflag,void *userdata)
23    {
24      if(num<strlen(pass)+1)
25        return(0);

26      strcpy(buf,pass);
27      return(strlen(pass));
28    }

29  static void sigpipe_handle(int x){
30  }

31  SSL_CTX *initialize_ctx(keyfile,password)
32    char *keyfile;
33    char *password;
34    {
35      SSL_METHOD *meth;
36      SSL_CTX *ctx;
```

```
37          if(!bio_err){
38            /* Global system initialization*/
39            SSL_library_init();
40            SSL_load_error_strings();

41            /* An error write context */
42            bio_err=BIO_new_fp(stderr,BIO_NOCLOSE);
43          }

44          /* Set up a SIGPIPE handler */
45          signal(SIGPIPE,sigpipe_handle);

46          /* Create our context*/
47          meth=SSLv3_method();
48          ctx=SSL_CTX_new(meth);

49          /* Load our keys and certificates*/
50          if(!(SSL_CTX_use_certificate_file(ctx,keyfile,SSL_FILETYPE_PEM)))
51            berr_exit("Couldn't read certificate file");

52          pass=password;
53          SSL_CTX_set_default_passwd_cb(ctx,password_cb);
54          if(!(SSL_CTX_use_PrivateKey_file(ctx,keyfile,SSL_FILETYPE_PEM)))
55            berr_exit("Couldn't read key file");

56          /* Load the CAs we trust*/
57          if(!(SSL_CTX_load_verify_locations(ctx,CA_LIST,0)))
58            berr_exit("Couldn't read CA list");
59          SSL_CTX_set_verify_depth(ctx,1);

60          /* Load randomness */
61          if(!(RAND_load_file(RANDOM,1024*1024)))
62            berr_exit("Couldn't load randomness");

63          return ctx;
64        }

65      void destroy_ctx(ctx)
66        SSL_CTX *ctx;
67        {
68          SSL_CTX_free(ctx);
69        }
```

common.c

echo.h

```
1  #ifndef _echo_h
2  #define _echo_h

3  void echo(SSL *ssl,int sock);

4  #endif
```

echo.h

echo.c

```
1  #include "common.h"

2  void echo(ssl,s)
3    SSL *ssl;
4    int s;
5    {
6      char buf[BUFSIZZ];
7      int r,len,offset;

8      while(1){
9        /* First read data */
10       r=SSL_read(ssl,buf,BUFSIZZ);
11       switch(SSL_get_error(ssl,r)){
12         case SSL_ERROR_NONE:
13           len=r;
14           break;
15         case SSL_ERROR_ZERO_RETURN:
16           goto end;
17         default:
18           berr_exit("SSL read problem");
19       }

20       /* Now keep writing until we've written everything*/
21       offset=0;

22       while(len){
23         r=SSL_write(ssl,buf+offset,len);
24         switch(SSL_get_error(ssl,r)){
25           case SSL_ERROR_NONE:
26             len-=r;
27             offset+=r;
28             break;
29           default:
30             berr_exit("SSL write problem");
31         }
```

```
32              }
33            }
34          end:
35            SSL_shutdown(ssl);
36            SSL_free(ssl);
37            close(s);
38          }
```

echo.c

read_write.h

```
1       #ifndef _read_write_h
2       #define _read_write_h

3       void read_write(SSL *ssl,int sock);

4       #endif
```

read_write.h

read_write.c

```
1       #include "common.h"

2       /* Read from the keyboard and write to the server
3          Read from the server and write to the keyboard

4          we use select() to multiplex
5       */
6       void read_write(ssl,sock)
7         SSL *ssl;
8         {
9           int width;
10          int r,c2sl=0,c2s_offset=0;
11          fd_set readfds,writefds;
12          int shutdown_wait=0;
13          char c2s[BUFSIZZ],s2c[BUFSIZZ];
14          int ofcmode;

15          /*First we make the socket nonblocking*/
16          ofcmode=fcntl(sock,F_GETFL,0);
17          ofcmode|=O_NDELAY;
18          if(fcntl(sock,F_SETFL,ofcmode))
19            err_exit("Couldn't make socket nonblocking");


20          width=sock+1;
```

```
21            while(1){
22              FD_ZERO(&readfds);
23              FD_ZERO(&writefds);

24              FD_SET(sock,&readfds);

25              /*If we've still got data to write then don't try to read*/
26              if(c2sl)
27                FD_SET(sock,&writefds);
28              else
29                FD_SET(fileno(stdin),&readfds);

30              r=select(width,&readfds,&writefds,0,0);
31              if(r==0)
32                continue;

33              /* Now check if there's data to read */
34              if(FD_ISSET(sock,&readfds)){
35                do {
36                  r=SSL_read(ssl,s2c,BUFSIZZ);

37                  switch(SSL_get_error(ssl,r)){
38                    case SSL_ERROR_NONE:
39                      fwrite(s2c,1,r,stdout);
40                      break;
41                    case SSL_ERROR_ZERO_RETURN:
42                      /* End of data */
43                      if(!shutdown_wait)
44                        SSL_shutdown(ssl);
45                      goto end;
46                      break;
47                    case SSL_ERROR_WANT_READ:
48                      break;
49                    default:
50                      berr_exit("SSL read problem");
51                  }
52                } while (SSL_pending(ssl));
53              }

54              /* Check for input on the console*/
55              if(FD_ISSET(fileno(stdin),&readfds)){
56                c2sl=read(fileno(stdin),c2s,BUFSIZZ);
57                if(c2sl==0){
58                  shutdown_wait=1;
59                  if(SSL_shutdown(ssl))
```

```
60                  return;
61                }
62                c2s_offset=0;
63              }

64              /* If we've got data to write then try to write it*/
65              if(c2sl && FD_ISSET(sock,&writefds)){
66                r=SSL_write(ssl,c2s+c2s_offset,c2sl);

67                switch(SSL_get_error(ssl,r)){
68                  /* We wrote something*/
69                  case SSL_ERROR_NONE:
70                    c2sl-=r;
71                    c2s_offset+=r;
72                    break;

73                    /* We would have blocked */
74                  case SSL_ERROR_WANT_WRITE:
75                    break;

76                    /* Some other error */
77                  default:
78                    berr_exit("SSL write problem");
79                }
80              }


81            }
82          end:
83            SSL_free(ssl);
84            close(sock);
85            return;
86          }
```

read_write.c

server.h

```
1       #ifndef _server_h
2       #define _server_h

3       #define KEYFILE "server.pem"
4       #define PASSWORD "password"
5       #define DHFILE "dh1024.pem"

6       int tcp_listen(void);
7       void load_dh_params(SSL_CTX *ctx,char *file);
```

```
8       void generate_eph_rsa_key(SSL_CTX *ctx);
9       #endif
```

server.h

server.c

```
1       #include "common.h"
2       #include "server.h"
3       int tcp_listen()
4         {
5           int sock;
6           struct sockaddr_in sin;
7           int val=1;
8           if((sock=socket(AF_INET,SOCK_STREAM,0))<0)
9             err_exit("Couldn't make socket");
10          memset(&sin,0,sizeof(sin));
11          sin.sin_addr.s_addr=INADDR_ANY;
12          sin.sin_family=AF_INET;
13          sin.sin_port=htons(PORT);
14          setsockopt(sock,SOL_SOCKET,SO_REUSEADDR,&val,sizeof(val));
15          if(bind(sock,(struct sockaddr *)&sin,sizeof(sin))<0)
16            berr_exit("Couldn't bind");
17          listen(sock,5);
18          return(sock);
19        }
20      void load_dh_params(ctx,file)
21        SSL_CTX *ctx;
22        char *file;
23        {
24          DH *ret=0;
25          BIO *bio;
26          if ((bio=BIO_new_file(file,"r")) == NULL)
27            berr_exit("Couldn't open DH file");
28          ret=PEM_read_bio_DHparams(bio,NULL,NULL,NULL);
29          BIO_free(bio);
30          if(SSL_CTX_set_tmp_dh(ctx,ret)<0)
31            berr_exit("Couldn't set DH parameters");
```

```
32          }

33        void generate_eph_rsa_key(ctx)
34          SSL_CTX *ctx;
35          {
36            RSA *rsa;

37            rsa=RSA_generate_key(512,RSA_F4,NULL,NULL);

38            if (!SSL_CTX_set_tmp_rsa(ctx,rsa))
39              berr_exit("Couldn't set RSA key");

40            RSA_free(rsa);
41          }
```

server.c

client.h

```
1         #ifndef _client_h
2         #define _client_h

3         #define KEYFILE "client.pem"
4         #define PASSWORD "password"

5         int tcp_connect(void);
6         void check_cert_chain(SSL *ssl,char *host);

7         #endif
```

client.h

client.c

```
1         #include "common.h"

2         int tcp_connect()
3          {
4             struct hostent *hp;
5             struct sockaddr_in addr;
6             int sock;

7             if(!(hp=gethostbyname(HOST)))
8               berr_exit("Couldn't resolve host");
9             memset(&addr,0,sizeof(addr));
10            addr.sin_addr=*(struct in_addr*)hp->h_addr_list[0];
11            addr.sin_family=AF_INET;
```

```
12            addr.sin_port=htons(PORT);

13            if((sock=socket(AF_INET,SOCK_STREAM,IPPROTO_TCP))<0)
14              err_exit("Couldn't create socket");
15            if(connect(sock,(struct sockaddr *)&addr,sizeof(addr))<0)
16              err_exit("Couldn't connect socket");

17            return sock;
18          }

19        /* Check that the common name matches the host name*/
20        void check_cert_chain(ssl,host)
21          SSL *ssl;
22          char *host;
23          {
24            X509 *peer;
25            char peer_CN[256];

26            if(SSL_get_verify_result(ssl)!=X509_V_OK)
27              berr_exit("Certificate doesn't verify");

28            /*Check the cert chain. The chain length
29              is automatically checked by OpenSSL when we
30              set the verify depth in the ctx

31              All we need to do here is check that the CN
32              matches
33            */
34            /*Check the common name*/
35            peer=SSL_get_peer_certificate(ssl);
36            X509_NAME_get_text_by_NID(X509_get_subject_name(peer),
37              NID_commonName, peer_CN, 256);
38            if(strcasecmp(peer_CN,host))
39            err_exit("Common name doesn't match host name");
40          }
```

client.c

```
1         /* A simple SSL echo server */
2         #include "common.h"
3         #include "server.h"
4         #include "echo.h"

5         static int s_server_session_id_context = 1;
```

sserver.c

```
6   int main(argc,argv)
7     int argc;
8     char **argv;
9     {
10      int sock,s;
11      BIO *sbio;
12      SSL_CTX *ctx;
13      SSL *ssl;
14      int r;

15      /* Build our SSL context*/
16      ctx=initialize_ctx(KEYFILE,PASSWORD);
17      load_dh_params(ctx,DHFILE);
18      generate_eph_rsa_key(ctx);

19      SSL_CTX_set_session_id_context(ctx,(void*)&s_server_session_id_context,
20        sizeof s_server_session_id_context);

21      sock=tcp_listen();

22      while(1){
23        if((s=accept(sock,0,0))<0)
24          err_exit("Problem accepting");

25        sbio=BIO_new_socket(s,BIO_NOCLOSE);
26        ssl=SSL_new(ctx);
27        SSL_set_bio(ssl,sbio,sbio);

28        if((r=SSL_accept(ssl)<=0))
29          berr_exit("SSL accept error");

30        echo(ssl,s);
31      }
32      destroy_ctx(ctx);
33      exit(0);
34    }
```

sserver.c

sclient.c

```
1   /* A simple SSL client.

2      It connects and then forwards data from/to the terminal
3      to/from the server
4   */
5   #include "common.h"
6   #include "client.h"
7   #include "read_write.h"
```

```
8   int main(argc,argv)
9     int argc;
10    char **argv;
11    {
12      SSL_CTX *ctx;
13      SSL *ssl;
14      BIO *sbio;
15      int sock;

16      /* Build our SSL context*/
17      ctx=initialize_ctx(KEYFILE,PASSWORD);

18      /* Connect the TCP socket*/
19      sock=tcp_connect();

20      /* Connect the SSL socket */
21      ssl=SSL_new(ctx);
22      sbio=BIO_new_socket(sock,BIO_NOCLOSE);
23      SSL_set_bio(ssl,sbio,sbio);
24      if(SSL_connect(ssl)<=0)
25        berr_exit("SSL connect error");
26      check_cert_chain(ssl,HOST);

27      /* read and write */
28      read_write(ssl,sock);

29      destroy_ctx(ctx);
30    }
```

sclient.c

rclient.c

```
1   /* SSL client demonstrating session resumption */
2   #include "common.h"
3   #include "client.h"
4   #include "read_write.h"

5   int main(argc,argv)
6     int argc;
7     char **argv;
8     {
9       SSL_CTX *ctx;
10      SSL *ssl;
11      BIO *sbio;
12      SSL_SESSION *sess;
13      int sock;

14      /* Build our SSL context*/
```

```
15          ctx=initialize_ctx(KEYFILE,PASSWORD);

16          /* Connect the TCP socket*/
17          sock=tcp_connect();

18          /* Connect the SSL socket */
19          ssl=SSL_new(ctx);
20          sbio=BIO_new_socket(sock,BIO_NOCLOSE);
21          SSL_set_bio(ssl,sbio,sbio);
22          if(SSL_connect(ssl)<=0)
23            berr_exit("SSL connect error (first connect)");
24          check_cert_chain(ssl,HOST);

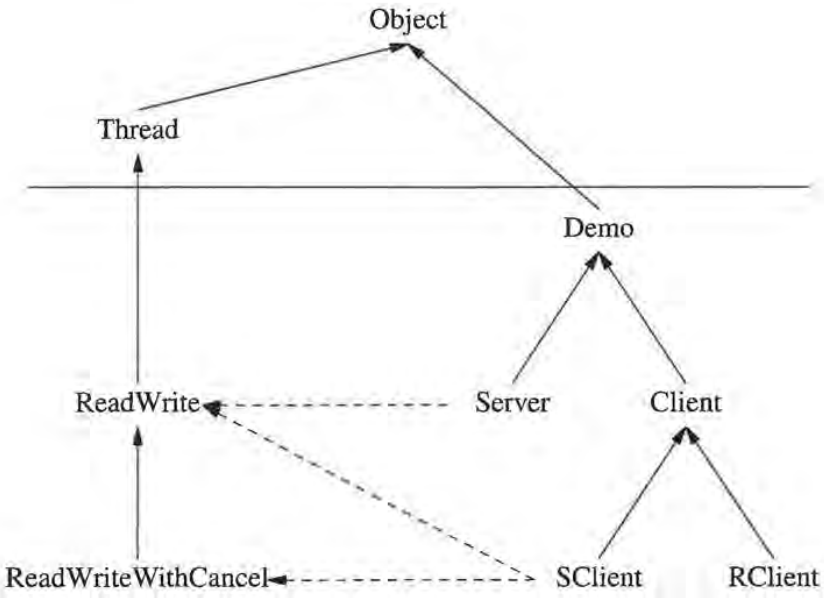
25          /* Now hang up and reconnect */
26          sess=SSL_get_session(ssl); /*Collect the session*/
27          SSL_shutdown(ssl);
28          close(sock);

29          sock=tcp_connect();
30          ssl=SSL_new(ctx);
31          sbio=BIO_new_socket(sock,BIO_NOCLOSE);
32          SSL_set_bio(ssl,sbio,sbio);
33          SSL_set_session(ssl,sess); /*And resume it*/
34          if(SSL_connect(ssl)<=0)
35            berr_exit("SSL connect error (second connect)");
36          check_cert_chain(ssl,HOST);

37          /*Now close everything down again*/
38          SSL_shutdown(ssl);
39          close(sock);
40          destroy_ctx(ctx);
41        }
```

rclient.c

A.1.2 Java Examples



The above picture shows the class hierarchy for the Java classes in the examples. Classes above the line are standard Java classes. Classes below the line are classes in the examples. Dashed lines indicate non-inheritance dependencies. So, `Server` extends `Demo` but simply uses `ReadWrite`.

`Demo` provides a set of common constant definitions and our common initialization routine.

`ReadWrite` provides a thread instance that copies data from an `InputStream` and to an `OutputStream`. When it receives an end of data on the `InputStream` it sends a close_notify down the `OutputStream`.

`ReadWriteWithCancel` is a specialization of `ReadWrite` that notifies another thread when it receives an end of data.

`Server` is our server program. It creates an `SSLServerSocket` and then loops between `accept()` and `ReadWrite`.

`Client` provides common client functionality. In particular, it contains our generic connect function and the server certificate check.

`SClient` is our main client program. It connects to the server and uses `ReadWrite` and a `ReadWriteWithCancel` to copy data to and from the server.

`RClient` is a stub client that demonstrates session resumption using PureTLS.

ReadWrite.java

```
1   import COM.claymoresystems.sslg.*;
2   import COM.claymoresystems.ptls.*;
3   import java.io.*;

4   /** This class simply copies anything from the in InputStream
5       to the out OutputStream
6   */
7   public class ReadWrite extends Thread {
8       protected SSLSocket s;
9       protected InputStream in;
10      protected OutputStream out;

11      /** Create a ReadWrite object

12          @param s the socket we're using
13          @param in the stream to read from
14          @param out the stream to write to
15      */
16      public ReadWrite(SSLSocket s,InputStream in,OutputStream out){
17        this.s=s;
18        this.in=in;
19        this.out=out;
20      }

21      /** Copy data from in to out.*/
22      public void run() {
23        byte[] buf=new byte[1024];
24        int read;

25        try {
26          while(true){
27            // Check for thread termination
28            if(isInterrupted())
29              break;

30            // Read data in
31            read=in.read(buf);

32            // Exit if there is no more data available
33            if(read==-1)
34              break;
```

```
35                  // Write the data out
36                  out.write(buf,0,read);
37                }

38              } catch (IOException e){
39                // run() can't throw IOException
40                throw new InternalError(e.toString());
41              }

42              // Finalize
43              onEOD();
44            }

45            protected void onEOD(){
46              try {
47                s.sendClose();
48              } catch (IOException e){
49                ; // Ignore broken pipe
50              }
51            }
52        }
```

ReadWrite.java

ReadWriteWithCancel.java

```
1     import COM.claymoresystems.sslg.*;
2     import COM.claymoresystems.ptls.*;
3     import java.io.*;

4     /** This class is a simple extension of ReadWrite.

5         When it receives an end of data on its in socket it
6         sends an interrupt to the 'cancel' thread.
7     */
8     public class ReadWriteWithCancel extends ReadWrite {
9         protected ReadWrite cancel;

10        public ReadWriteWithCancel(SSLSocket s,InputStream in,OutputStream out,
11          ReadWrite cancel){
12          super(s,in,out);
13          this.cancel=cancel;
14        }

15        protected void onEOD(){
16          if(cancel.isAlive()){
17            cancel.interrupt();
```

```
18              try {
19                cancel.join();
20              } catch (InterruptedException e){
21                throw new InternalError(e.toString());
22              }
23            }
24          }
25      }
```

ReadWriteWithCancel.java

Demo.java

```
1   import COM.claymoresystems.sslg.*;
2   import COM.claymoresystems.ptls.*;

3   public class Demo {
4       public static final String host=  "localhost";
5       public static final int port=  4433;
6       public static final String root=  "root.pem";
7       public static final String random= "random.pem";

8       static SSLContext createSSLContext(String keyfile,String password){
9         SSLContext ctx=new SSLContext();

10        try {
11          ctx.loadRootCertificates(root);
12          ctx.loadEAYKeyFile(keyfile,password);
13          ctx.useRandomnessFile(random,password);
14        } catch (Exception e){
15          throw new InternalError(e.toString());
16        }

17        return ctx;
18      }
19  }
```

Demo.java

Server.java

```
1  /** This class is a simple SSL echo server */
2  import COM.claymoresystems.sslg.*;
3  import COM.claymoresystems.ptls.*;
4  import java.io.*;

5  public class Server extends Demo {
6      protected static final String keyfile="server.pem";
7      protected static final String password="password";
8      protected static final String dh="dh1024.pem";
9      protected static boolean requireclientauth=true;

10     protected SSLSocket sock;
11     protected InputStream in;
12     protected OutputStream out;

13     public static void main(String []args)
14       throws IOException {
15       SSLContext ctx=createSSLContext(keyfile,password);

16       // Load our DH group
17       ctx.loadDHParams(dh);

18       SSLServerSocket listen=new SSLServerSocket(ctx,port);

19       while(true){
20         SSLSocket s=(SSLSocket)listen.accept();

21         // Process this connection in a new thread
22         ReadWrite rw=new ReadWrite(s,s.getInputStream(),
23           s.getOutputStream());
24         rw.start();
25       }
26     }
27 }
```

Server.java

SClient.java

```
1  /* A simple SSL client.

2     It connects and then forwards data from/to the terminal
3     to/from the server
4  */
5  import COM.claymoresystems.sslg.*;
6  import COM.claymoresystems.ptls.*;
7  import java.io.*;
```

```
8       public class SClient extends Client {
9           public static void main(String []args)
10            throws IOException,InterruptedException {
11            SSLContext ctx=createSSLContext(keyfile,password);
12            SSLSocket s=connect(ctx,host,port);

13            // This thread reads from the console and writes to the server
14            ReadWrite c2s=new ReadWrite(s,System.in,s.getOutputStream());
15            c2s.start();

16            // This thread reads from the server and writes to the console
17            ReadWriteWithCancel s2c=new ReadWriteWithCancel(s,
18              s.getInputStream(),System.out,c2s);
19            s2c.start();
20            s2c.setPriority(Thread.MAX_PRIORITY);
21            s2c.join();
22          }
23      }
```

SClient.java

RClient.java

```
1       /* SSL client demonstrating session resumption */
2       import COM.claymoresystems.sslg.*;
3       import COM.claymoresystems.ptls.*;
4       import java.io.*;

5       public class RClient extends Client {
6           public static void main(String []args)
7             throws IOException {
8             SSLContext ctx=createSSLContext(keyfile,password);

9             /* Connect and close*/
10            SSLSocket s=connect(ctx,host,port);
11            s.close();

12            /* Now reconnect: resumption happens automatically*/
13            s=connect(ctx,host,port);
14            s.close();
15          }
16      }
```

RClient.java

A.2 Chapter 9

This section contains complete versions of the example programs from Chapter 9: `pclient.c` and `mserver.c`. It also contains the section of mod_ssl responsible for doing cross-process session caching.

A.2.1 HTTPS Examples

The programs presented in Chapter 9 demonstrated various SSL idioms that are commonly used with HTTPS. They build on the same common framework as the programs from Chapter 8. Again, machine readable versions of these programs can also be downloaded from the author's Web site at `http://www.rtfm.com/sslbook/examples`.

`pclient` is a proxy-capable client. It uses the HTTP `CONNECT` method to connect to a proxy and then tunnel through to the server.

`mserver` is an SSL server capable of handling multiple clients simultaneously in different server process. Essentially, this is the same job that `Server` does in Java.

pclient.c

```
1   /* A proxy-capable SSL client.

2      This is just like sclient but it only supports proxies
3   */
4   #include <string.h>
5   #include "common.h"
6   #include "client.h"
7   #include "read_write.h"

8   #define PROXY "localhost"
9   #define PROXY_PORT 8080
10  #define REAL_HOST "localhost"
11  #define REAL_PORT 4433

12  int writestr(sock,str)
13    int sock;
14    char *str;
15    {
16      int len=strlen(str);
17      int r,wrote=0;

18      while(len){
19        r=write(sock,str,len);
20        if(r<=0)
21          err_exit("Write error");
```

```
22              len-=r;
23              str+=r;
24              wrote+=r;
25            }

26            return (wrote);
27          }

28        int readline(sock,buf,len)
29          int sock;
30          char *buf;
31          int len;
32          {
33            int n,r;
34            char *ptr=buf;

35            for(n=0;n<len;n++){
36              r=read(sock,ptr,1);

37              if(r<=0)
38                err_exit("Read error");

39              if(*ptr=='\n'){
40                *ptr=0;

41                /* Strip off the CR if it's there */
42                if(buf[n-1]=='\r'){
43                  buf[n-1]=0;
44                  n--;
45                }

46                return(n);
47              }

48              *ptr++;
49            }

50            err_exit("Buffer too short");
51          }

52        int proxy_connect(){
53            struct hostent *hp;
54            struct sockaddr_in addr;
55            int sock;
56            BIO *sbio;
```

```
57          char buf[1024];
58          char *protocol, *response_code;

59          /* Connect to the proxy, not the host */
60          if(!(hp=gethostbyname(PROXY)))
61            berr_exit("Couldn't resolve host");
62          memset(&addr,0,sizeof(addr));
63          addr.sin_addr=*(struct in_addr*)hp->h_addr_list[0];
64          addr.sin_family=AF_INET;
65          addr.sin_port=htons(PROXY_PORT);

66          if((sock=socket(AF_INET,SOCK_STREAM,IPPROTO_TCP))<0)
67            err_exit("Couldn't create socket");
68          if(connect(sock,(struct sockaddr *)&addr,sizeof(addr))<0)
69            err_exit("Couldn't connect socket");

70          /* Now that we're connected, do the proxy request */
71          sprintf(buf,"CONNECT %s:%d HTTP/1.0\r\n\r\n",REAL_HOST,REAL_PORT);
72          writestr(sock,buf);

73          /* And read the response*/
74          if(readline(sock,buf,sizeof(buf))==0)
75            err_exit("Empty response from proxy");

76          if((protocol=strtok(buf," "))<0)
77            err_exit("Couldn't parse server response: getting protocol");
78          if(strncmp(protocol,"HTTP",4))
79            err_exit("Unrecognized protocol");
80          if((response_code=strtok(0," "))<0)
81            err_exit("Couldn't parse server response: getting response code");
82          if(strcmp(response_code,"200"))
83            err_exit("Received error from proxy server");

84          /* Look for the blank line that signals end of header*/
85          while(readline(sock,buf,sizeof(buf))>0) {
86            ;
87          }

88          return(sock);
89        }

90        int main(argc,argv)
91          int argc;
92          char **argv;
93          {
```

```
94          SSL_CTX *ctx;
95          SSL *ssl;
96          BIO *sbio;
97          int sock;

98          /* Build our SSL context*/
99          ctx=initialize_ctx(KEYFILE,PASSWORD);

100         /* Connect the TCP socket*/
101         sock=proxy_connect();

102         /* Connect the SSL socket */
103         ssl=SSL_new(ctx);
104         sbio=BIO_new_socket(sock,BIO_NOCLOSE);
105         SSL_set_bio(ssl,sbio,sbio);
106         if(SSL_connect(ssl)<=0)
107           berr_exit("SSL connect error");
108         check_cert_chain(ssl,HOST);

109         /* read and write */
110         read_write(ssl,sock);

111         destroy_ctx(ctx);
112       ]
```

pclient.c

mserver.c

```
1       /* A multiprocess SSL server */
2       #include "common.h"
3       #include "server.h"
4       #include "echo.h"

5       int main(argc,argv)
6         int argc;
7         char **argv;
8         {
9           int sock,s;
10          BIO *sbio;
11          SSL_CTX *ctx;
12          SSL *ssl;
13          int r;
14          pid_t pid;

15          /* Build our SSL context*/
16          ctx=initialize_ctx(KEYFILE,PASSWORD);
```

```
17          load_dh_params(ctx,DHFILE);
18          generate_eph_rsa_key(ctx);

19          sock=tcp_listen();

20          while(1){
21            if((s=accept(sock,0,0))<0)
22              err_exit("Problem accepting");

23            if(pid=fork()){
24              close(s);
25            }
26            else {
27              sbio=BIO_new_socket(s,BIO_NOCLOSE);
28              ssl=SSL_new(ctx);
29              SSL_set_bio(ssl,sbio,sbio);

30              if((r=SSL_accept(ssl)<=0))
31                berr_exit("SSL accept error");

32              echo(ssl,s);
33            }
34          }
35          destroy_ctx(ctx);
36          exit(0);
37        }
```

mserver.c

A.2.2 mod_ssl Session Caching

mod_ssl is a module for the popular Apache Web server. It uses OpenSSL to add SSL/TLS support to Apache. Apache is primarily used on UNIX systems and thus uses multiple processes to handle multiple clients at the same time. As we discussed in Chapter 9, we thus need special support in order to share sessions between these processes. The rest of this section examines this code.

OpenSSL provides its own session caching code that works within a single program. In order to accommodate application programs with more sophisticated needs, it provides a number of hooks which allow programs to provide their own session handling code. We first examine the hooks that mod_ssl installs.

Next, we consider the actual session caching code in mod_ssl. Actually, mod_ssl provides two kinds of session caching support. The first (more portable) version uses dbm to save the session ids to disk. The second (faster) version uses shared memory segments. mod_ssl's hooks call generic functions which switch-hit between the different implementations.

Hooks

ssl_engine_kernel.c

```
167     /*
168      *  This callback function is executed by OpenSSL whenever a new SSL_SESSION is
169      *  added to the internal OpenSSL session cache. We use this hook to spread the
170      *  SSL_SESSION also to the inter-process disk-cache to make share it with our
171      *  other Apache pre-forked server processes.
172      */
173     int ssl_callback_NewSessionCacheEntry(SSL *ssl, SSL_SESSION *pNew)
174     {
175         conn_rec *conn;
176         server_rec *s;
177         SSLSrvConfigRec *sc;
178         long t;
179         BOOL rc;

180         /*
181          * Get Apache context back through OpenSSL context
182          */
183         conn = (conn_rec *)SSL_get_app_data(ssl);
184         s    = conn->server;
185         sc   = mySrvConfig(s);

186         /*
187          * Set the timeout also for the internal OpenSSL cache, because this way
188          * our inter-process cache is consulted only when it's really necessary.
189          */
```

```
190             t = sc->nSessionCacheTimeout;
191             SSL_set_timeout(pNew, t);
192             /*
193              * Store the SSL_SESSION in the inter-process cache with the
194              * same expire time, so it expires automatically there, too.
195              */
196             t = (SSL_get_time(pNew) + sc->nSessionCacheTimeout);
197             rc = ssl_scache_store(s, pNew, t);
198             /*
199              * Log this cache operation
200              */
201             ssl_log(s, SSL_LOG_TRACE, "Inter-Process Session Cache: "
202                     "request=SET status=%s id=%s timeout=%ds (session caching)",
203                     rc == TRUE ? "OK" : "BAD",
204                     ssl_scache_id2sz(pNew->session_id, pNew->session_id_length),
205                     t-time(NULL));
206             /*
207              * return 0 which means to OpenSSL that the pNew is still
208              * valid and was not freed by us with SSL_SESSION_free().
209              */
210             return 0;
211         }
212         /*
213          *  This callback function is executed by OpenSSL whenever a
214          *  SSL_SESSION is looked up in the internal OpenSSL cache and it
215          *  was not found. We use this to lookup the SSL_SESSION in the
216          *  inter-process disk-cache where it was perhaps stored by one
217          *  of our other Apache pre-forked server processes.
218          */
219         SSL_SESSION *ssl_callback_GetSessionCacheEntry(
220             SSL *ssl, unsigned char *id, int idlen, int *pCopy)
221         {
222             conn_rec *conn;
223             server_rec *s;
224             SSL_SESSION *pSession;
225             /*
226              * Get Apache context back through OpenSSL context
227              */
228             conn = (conn_rec *)SSL_get_app_data(ssl);
229             s    = conn->server;
```

```
230         /*
231          * Try to retrieve the SSL_SESSION from the inter-process cache
232          */
233         pSession = ssl_scache_retrieve(s, id, idlen);

234         /*
235          * Log this cache operation
236          */
237         if (pSession != NULL)
238             ssl_log(s, SSL_LOG_TRACE, "Inter-Process Session Cache: "
239                     "request=GET status=FOUND id=%s (session reuse)",
240                     ssl_scache_id2sz(id, idlen));
241         else
242             ssl_log(s, SSL_LOG_TRACE, "Inter-Process Session Cache: "
243                     "request=GET status=MISSED id=%s (session renewal)",
244                     ssl_scache_id2sz(id, idlen));

245         /*
246          * Return NULL or the retrieved SSL_SESSION. But indicate (by
247          * setting pCopy to 0) that the reference count on the
248          * SSL_SESSION should not be incremented by the SSL library,
249          * because we will no longer hold a reference to it ourself.
250          */
251         *pCopy = 0;
252         return pSession;
253     }

254     /*
255      *  This callback function is executed by OpenSSL whenever a
256      *  SSL_SESSION is removed from the the internal OpenSSL cache.
257      *  We use this to remove the SSL_SESSION in the inter-process
258      *  disk-cache, too.
259      */
260     void ssl_callback_DelSessionCacheEntry(
261         SSL_CTX *ctx, SSL_SESSION *pSession)
262     {
263         server_rec *s;

264         /*
265          * Get Apache context back through OpenSSL context
266          */
267         s = (server_rec *)SSL_CTX_get_app_data(ctx);
268         if (s == NULL) /* on server shutdown Apache is already gone */
269             return;
```

```
270             /*
271              * Remove the SSL_SESSION from the inter-process cache
272              */
273             ssl_scache_remove(s, pSession);

274             /*
275              * Log this cache operation
276              */
277             ssl_log(s, SSL_LOG_TRACE, "Inter-Process Session Cache: "
278                     "request=REM status=OK id=%s (session dead)",
279                     ssl_scache_id2sz(pSession->session_id,
280                     pSession->session_id_length));

281             return;
282         }
```

ssl_engine_kernel.c

OpenSSL provides for three session caching hooks: *session creation*, *session fetching*, and *session_deletion*. The mod_ssl hook functions are basically wrappers around the cache functions that we'll discuss in the next section. All they really do is call the cache functions and log the results.

Session creation

173–211 The session creation code is the only complicated hook. mod_ssl's interprocess cache is intended to be used only if the OpenSSL internal cache fails. Thus, if a session was created by the same process that it's being resumed in, we don't have to go to the interprocess cache to resume it. However, because entries have to be created in both caches, `ssl_callback_NewSessionCacheEntry()` is always called when a new session is created. It does three jobs: sets the timeout in the internal cache, creates an interprocess cache entry using `ssl_scache_store()`, and logs the results.

Session fetching

219–253 `ssl_callback_GetSessionCacheEntry()` is called whenever a client requests session resumption but OpenSSL can't find the session in the internal cache. It calls `ssl_scache_retrieve()` to get the cache entry and then logs the result.

Session deletion

260–282 `ssl_callback_DelSessionCacheEntry()` is called whenever a session becomes invalid, such as when we catch an alert or have a premature close. It simply calls `ssl_scache_remove()` to delete the session and then logs the result.

Generic Session Caching Code

ssl_engine_scache.c

```
108     BOOL ssl_scache_store(server_rec *s, SSL_SESSION *pSession, int timeout)
109     {
110         SSLModConfigRec *mc = myModConfig();
111         ssl_scinfo_t SCI;
112         UCHAR buf[MAX_SESSION_DER];
113         UCHAR *b;
114         BOOL rc = FALSE;

115         /* add the key */
116         SCI.ucaKey = pSession->session_id;
117         SCI.nKey   = pSession->session_id_length;

118         /* transform the session into a data stream */
119         SCI.ucaData    = b = buf;
120         SCI.nData      = i2d_SSL_SESSION(pSession, &b);
121         SCI.tExpiresAt = timeout;

122         /* and store it... */
123         if (mc->nSessionCacheMode == SSL_SCMODE_DBM)
124             rc = ssl_scache_dbm_store(s, &SCI);
125         else if (mc->nSessionCacheMode == SSL_SCMODE_SHM)
126             rc = ssl_scache_shm_store(s, &SCI);

127     #ifdef SSL_VENDOR
128         ap_hook_use("ap::mod_ssl::vendor::scache_store",
129                     AP_HOOK_SIG3(void,ptr,ptr), AP_HOOK_ALL, s, &SCI);
130     #endif

131         /* allow the regular expiring to occur */
132         ssl_scache_expire(s, time(NULL));

133         return rc;
134     }

135     SSL_SESSION *ssl_scache_retrieve(server_rec *s, UCHAR *id, int idlen)
136     {
137         SSLModConfigRec *mc = myModConfig();
138         SSL_SESSION *pSession = NULL;
139         ssl_scinfo_t SCI;
140         time_t tNow;

141         /* determine current time */
```

```
142             tNow = time(NULL);
143             /* allow the regular expiring to occur */
144             ssl_scache_expire(s, tNow);
145             /* create cache query */
146             SCI.ucaKey     = id;
147             SCI.nKey       = idlen;
148             SCI.ucaData    = NULL;
149             SCI.nData      = 0;
150             SCI.tExpiresAt = 0;
151             /* perform cache query */
152             if (mc->nSessionCacheMode == SSL_SCMODE_DBM)
153                 ssl_scache_dbm_retrieve(s, &SCI);
154             else if (mc->nSessionCacheMode == SSL_SCMODE_SHM)
155                 ssl_scache_shm_retrieve(s, &SCI);
156         #ifdef SSL_VENDOR
157             ap_hook_use("ap::mod_ssl::vendor::scache_retrieve",
158                         AP_HOOK_SIG3(void,ptr,ptr), AP_HOOK_ALL, s, &SCI);
159         #endif
160             /* return immediately if not found */
161             if (SCI.ucaData == NULL)
162                 return NULL;
163             /* check for expire time */
164             if (SCI.tExpiresAt <= tNow) {
165                 if (mc->nSessionCacheMode == SSL_SCMODE_DBM)
166                     ssl_scache_dbm_remove(s, &SCI);
167                 else if (mc->nSessionCacheMode == SSL_SCMODE_SHM)
168                     ssl_scache_shm_remove(s, &SCI);
169         #ifdef SSL_VENDOR
170                 ap_hook_use("ap::mod_ssl::vendor::scache_remove",
171                             AP_HOOK_SIG3(void,ptr,ptr), AP_HOOK_ALL, s, &SCI);
172         #endif
173                 return NULL;
174             }
175             /* extract result and return it */
176             pSession = d2i_SSL_SESSION(NULL, &SCI.ucaData, SCI.nData);
177             return pSession;
178         }
```

```
179     void ssl_scache_remove(server_rec *s, SSL_SESSION *pSession)
180     {
181         SSLModConfigRec *mc = myModConfig();
182         ssl_scinfo_t SCI;

183         /* create cache query */
184         SCI.ucaKey     = pSession->session_id;
185         SCI.nKey       = pSession->session_id_length;
186         SCI.ucaData    = NULL;
187         SCI.nData      = 0;
188         SCI.tExpiresAt = 0;

189         /* perform remove */
190         if (mc->nSessionCacheMode == SSL_SCMODE_DBM)
191             ssl_scache_dbm_remove(s, &SCI);
192         else if (mc->nSessionCacheMode == SSL_SCMODE_SHM)
193             ssl_scache_shm_remove(s, &SCI);

194     #ifdef SSL_VENDOR
195         ap_hook_use("ap::mod_ssl::vendor::scache_remove",
196                     AP_HOOK_SIG3(void,ptr,ptr), AP_HOOK_ALL, s, &SCI);
197     #endif

198         return;
199     }

200     void ssl_scache_expire(server_rec *s, time_t now)
201     {
202         SSLModConfigRec *mc = myModConfig();
203         SSLSrvConfigRec *sc = mySrvConfig(s);
204         static time_t last = 0;

205         /*
206          * make sure the expiration for still not-accessed session
207          * cache entries is done only from time to time
208          */
209         if (now < last+sc->nSessionCacheTimeout)
210             return;
211         last = now;

212         /*
213          * Now perform the expiration
214          */
215         if (mc->nSessionCacheMode == SSL_SCMODE_DBM)
216             ssl_scache_dbm_expire(s, now);
```

```
217             else if (mc->nSessionCacheMode == SSL_SCMODE_SHM)
218                 ssl_scache_shm_expire(s, now);

219             return;
220         }
```

ssl_engine_scache.c

The mod_ssl session caching code is divided between the generic code and the underlying implementations of interprocess sharing. In particular, the generic code arranges for expiration to happen at the appropriate times and makes sure that expired sessions can't be resumed. Other than that, the heavy lifting is all done by the underlying interprocess sharing code.

Session storage

108–134 `ssl_scache_store()` arranges to serialize the session object. It then calls the appropriate session caching method to actually share the session. Finally, it calls the cache expiration function.

Session retrieval

135–178 `ssl_scache_retrieve()` first calls the appropriate session caching method to load the session object. If the object is found, it then checks to see if it's expired. If the session is expired, it automatically removes it from the cache. Otherwise, it deserializes the object and returns it.

Session deletion

179–199 `ssl_scache_remove()` merely calls the underlying cache methods to remove the cache entry.

Cache expiry

200–245 `ssl_scache_expire()` is never called directly by the mod_ssl session caching hooks. However, it's necessary that we occasionally go through the session cache and delete old entries, otherwise the session cache would grow without bound. Instead, it's automatically called whenever a new session object is created. mod_ssl allows the administrator to set expiries to happen after a certain interval, so most calls to `ssl_scache_expire()` are within that interval and return at line 210. Otherwise, `ssl_scache_expire()` calls the appropriate expiration method.

DBM Session Caching

ssl_engine_scache.c

```
368 BOOL ssl_scache_dbm_store(server_rec *s, ssl_scinfo_t *SCI)
369 {
370     SSLModConfigRec *mc = myModConfig();
371     DBM *dbm;
372     datum dbmkey;
373     datum dbmval;

374     /* be careful: do not try to store too much bytes in a DBM file! */
375 #ifdef SSL_USE_SDBM
376     if ((SCI->nKey + SCI->nData) >= PAIRMAX)
377         return FALSE;
378 #else
379     if ((SCI->nKey + SCI->nData) >= 950 /* at least less than approx. 1KB */)
380         return FALSE;
381 #endif

382     /* create DBM key */
383     dbmkey.dptr  = (char *)(SCI->ucaKey);
384     dbmkey.dsize = SCI->nKey;

385     /* create DBM value */
386     dbmval.dsize = sizeof(time_t) + SCI->nData;
387     dbmval.dptr  = (char *)malloc(dbmval.dsize);
388     if (dbmval.dptr == NULL)
389         return FALSE;
390     memcpy((char *)dbmval.dptr, &SCI->tExpiresAt, sizeof(time_t));
391     memcpy((char *)dbmval.dptr+sizeof(time_t), SCI->ucaData, SCI->nData);

392     /* and store it to the DBM file */
393     ssl_mutex_on(s);
394     if ((dbm = ssl_dbm_open(mc->szSessionCacheDataFile,
395                             O_RDWR, SSL_DBM_FILE_MODE)) == NULL) {
396         ssl_log(s, SSL_LOG_ERROR|SSL_ADD_ERRNO,
397                 "Cannot open SSLSessionCache DBM file '%s' for writing (store)",
398                 mc->szSessionCacheDataFile);
399         ssl_mutex_off(s);
400         free(dbmval.dptr);
401         return FALSE;
402     }
403     if (ssl_dbm_store(dbm, dbmkey, dbmval, DBM_INSERT) < 0) {
404         ssl_log(s, SSL_LOG_ERROR|SSL_ADD_ERRNO,
405                 "Cannot store SSL session to DBM file '%s'",
```

```
406                     mc->szSessionCacheDataFile);
407             ssl_dbm_close(dbm);
408             ssl_mutex_off(s);
409             free(dbmval.dptr);
410             return FALSE;
411         }
412         ssl_dbm_close(dbm);
413         ssl_mutex_off(s);
414         /* free temporary buffers */
415         free(dbmval.dptr);
416         return TRUE;
417     }
418     void ssl_scache_dbm_retrieve(server_rec *s, ssl_scinfo_t *SCI)
419     {
420         SSLModConfigRec *mc = myModConfig();
421         DBM *dbm;
422         datum dbmkey;
423         datum dbmval;
424         /* initialize result */
425         SCI->ucaData    = NULL;
426         SCI->nData      = 0;
427         SCI->tExpiresAt = 0;
428         /* create DBM key and values */
429         dbmkey.dptr  = (char *)(SCI->ucaKey);
430         dbmkey.dsize = SCI->nKey;
431         /* and fetch it from the DBM file */
432         ssl_mutex_on(s);
433         if ((dbm = ssl_dbm_open(mc->szSessionCacheDataFile,
434                                 O_RDONLY, SSL_DBM_FILE_MODE)) == NULL) {
435             ssl_log(s, SSL_LOG_ERROR|SSL_ADD_ERRNO,
436                     "Cannot open SSLSessionCache DBM file '%s' for reading (fetch)",
437                     mc->szSessionCacheDataFile);
438             ssl_mutex_off(s);
439             return;
440         }
441         dbmval = ssl_dbm_fetch(dbm, dbmkey);
442         ssl_dbm_close(dbm);
443         ssl_mutex_off(s);
```

```
444         /* immediately return if not found */
445         if (dbmval.dptr == NULL || dbmval.dsize <= sizeof(time_t))
446             return;
447         /* copy over the information to the SCI */
448         SCI->nData   = dbmval.dsize-sizeof(time_t);
449         SCI->ucaData = (UCHAR *)malloc(SCI->nData);
450         if (SCI->ucaData == NULL) {
451             SCI->nData = 0;
452             return;
453         }
454         memcpy(SCI->ucaData, (char *)dbmval.dptr+sizeof(time_t), SCI->nData);
455         memcpy(&SCI->tExpiresAt, dbmval.dptr, sizeof(time_t));
456         return;
457     }
458     void ssl_scache_dbm_remove(server_rec *s, ssl_scinfo_t *SCI)
459     {
460         SSLModConfigRec *mc = myModConfig();
461         DBM *dbm;
462         datum dbmkey;
463         /* create DBM key and values */
464         dbmkey.dptr  = (char *)(SCI->ucaKey);
465         dbmkey.dsize = SCI->nKey;
466         /* and delete it from the DBM file */
467         ssl_mutex_on(s);
468         if ((dbm = ssl_dbm_open(mc->szSessionCacheDataFile,
469                                 O_RDWR, SSL_DBM_FILE_MODE)) == NULL) {
470             ssl_log(s, SSL_LOG_ERROR|SSL_ADD_ERRNO,
471                     "Cannot open SSLSessionCache DBM file '%s' for writing (delete)",
472                     mc->szSessionCacheDataFile);
473             ssl_mutex_off(s);
474             return;
475         }
476         ssl_dbm_delete(dbm, dbmkey);
477         ssl_dbm_close(dbm);
478         ssl_mutex_off(s);
479         return;
480     }
481     void ssl_scache_dbm_expire(server_rec *s, time_t tNow)
```

```
482         {
483             SSLModConfigRec *mc = myModConfig();
484             DBM *dbm;
485             datum dbmkey;
486             datum dbmval;
487             pool *p;
488             time_t tExpiresAt;
489             int nElements = 0;
490             int nDeleted = 0;
491             int bDelete;
492             datum *keylist;
493             int keyidx;
494             int i;

495             /*
496              * Here we have to be very carefully: Not all DBM libraries are
497              * smart enough to allow one to iterate over the elements and at the
498              * same time delete expired ones. Some of them get totally crazy
499              * while others have no problems. So we have to do it the slower but
500              * more safe way: we first iterate over all elements and remember
501              * those which have to be expired. Then in a second pass we delete
502              * all those expired elements. Additionally we reopen the DBM file
503              * to be really safe in state.
504              */

505         #define KEYMAX 1024

506             ssl_mutex_on(s);
507             for (;;) {
508                 /* allocate the key array in a memory sub pool */
509                 if ((p = ap_make_sub_pool(NULL)) == NULL)
510                     break;
511                 if ((keylist = ap_palloc(p, sizeof(dbmkey)*KEYMAX)) == NULL) {
512                     ap_destroy_pool(p);
513                     break;
514                 }

515                 /* pass 1: scan DBM database */
516                 keyidx = 0;
517                 if ((dbm = ssl_dbm_open(mc->szSessionCacheDataFile,
518                                         O_RDWR, SSL_DBM_FILE_MODE)) == NULL) {
519                     ssl_log(s, SSL_LOG_ERROR|SSL_ADD_ERRNO,
520                             "Cannot open SSLSessionCache DBM file '%s' for scanning",
521                             mc->szSessionCacheDataFile);
522                     ap_destroy_pool(p);
```

```
523                 break;
524             }
525             dbmkey = ssl_dbm_firstkey(dbm);
526             while (dbmkey.dptr != NULL) {
527                 nElements++;
528                 bDelete = FALSE;
529                 dbmval = ssl_dbm_fetch(dbm, dbmkey);
530                 if (dbmval.dsize <= sizeof(time_t) || dbmval.dptr == NULL)
531                     bDelete = TRUE;
532                 else {
533                     memcpy(&tExpiresAt, dbmval.dptr, sizeof(time_t));
534                     if (tExpiresAt <= tNow)
535                         bDelete = TRUE;
536                 }
537                 if (bDelete) {
538                     if ((keylist[keyidx].dptr = ap_palloc(p, dbmkey.dsize)) != NULL) {
539                         memcpy(keylist[keyidx].dptr, dbmkey.dptr, dbmkey.dsize);
540                         keylist[keyidx].dsize = dbmkey.dsize;
541                         keyidx++;
542                         if (keyidx == KEYMAX)
543                             break;
544                     }
545                 }
546                 dbmkey = ssl_dbm_nextkey(dbm);
547             }
548             ssl_dbm_close(dbm);

549             /* pass 2: delete expired elements */
550             if ((dbm = ssl_dbm_open(mc->szSessionCacheDataFile,
551                                     O_RDWR, SSL_DBM_FILE_MODE)) == NULL) {
552                 ssl_log(s, SSL_LOG_ERROR|SSL_ADD_ERRNO,
553                         "Cannot re-open SSLSessionCache DBM file '%s' for expiring",
554                         mc->szSessionCacheDataFile);
555                 ap_destroy_pool(p);
556                 break;
557             }
558             for (i = 0; i < keyidx; i++) {
559                 ssl_dbm_delete(dbm, keylist[i]);
560                 nDeleted++;
561             }
562             ssl_dbm_close(dbm);

563             /* destroy temporary pool */
564             ap_destroy_pool(p);
```

```
565                 if (keyidx < KEYMAX)
566                     break;
567             }
568             ssl_mutex_off(s);

569             ssl_log(s, SSL_LOG_TRACE, "Inter-Process Session Cache (DBM) Expiry: "
570                     "old: %d, new: %d, removed: %d", nElements, nElements-nDeleted, nDeleted);
571             return;
572         }
```

ssl_engine_scache.c

DBM is a standard UNIX package that offers a simple key-value database. It allows the creation of arbitrary pairs of lookup keys and data values in a one-to-one association. Essentially, it's a large, self-resizing hash table. However, the data is stored in disk files so it can be shared between processes.

DBM maintains an index of all the keys for fast lookups. Unfortunately, DBM was not designed for concurrent access. Therefore, there's no way for it to detect that the database has been updated and reload the index. The only time the index is reloaded is when the database is opened. Similarly, portions of the database may be cached and get written out only when it is closed. Thus, for multiple processes to use the database, they must open and close it each time they want to access some data. Most of the complication in the mod_ssl DBM code is concerned with arranging this.

Session creation

368–417 The first job is to create the DBM keys and values. The key is simply the session ID. The value contains the time and the serialized session object. Because the value must be a single byte string, they must be packed together, which is done in lines 385-391. Once the key and value are ready to store, the database must be locked so that no other process can write to it. mod_ssl has an interprocess mutex that it uses for this purpose (line 393). Once the database is locked, mod_ssl can open the database and then store our key-value pair. When this is all finished, mod_ssl closes the database and releases the lock.

Session retrieval

418–457 Session retrieval is just session storage in reverse. Note that mod_ssl makes no distinction between mutexes for reading and writing. It's generally safe for two processes to read the same DBM file at the same time. In an active system with a lot more session resumption than session creation, it would pay to have a read lock and a write lock. Thus, multiple readers would be able to read the database concurrently rather than blocking each other. Once the session is fetched, the expiry time and the session object must be unpacked and returned to the caller (lines 448–453).

Session removal

458–480 `ssl_scache_dbm_remove()` follows a now familiar pattern. The database is locked and then the deletion is performed. Then the database is unlocked.

Cache expiry

481–572 Session deletion is tricky. In theory, we'd like to do a linear scan of the database and delete the expired elements as we go. Unfortunately, many old DBM implementations would corrupt the database if you tried to scan and delete at the same time. Thus, expiration is done in two passes. In the first pass (lines 515–548) the database is scanned and a list of all the sessions to be expired is created. In the second pass (lines 549–567) each entry on the list is deleted. This is inefficient, and you'd like not to do it very often.

Shared Memory Session Caching

DBM session caching is very slow. Every cache store or fetch requires us to lock the database, open two files (the index and data files), read data from the disk, close the files, and unlock the database. This adds a very substantial amount of overhead to the process. Thus, mod_ssl has a faster approach based on shared memory. The general idea is to create a key-value database that stores its data in memory rather than on disk. Then this memory can be located in a shared memory segment so that multiple processes can access it.

ssl_session_cache.c

```
752     BOOL ssl_scache_shm_store(server_rec *s, ssl_scinfo_t *SCI)
753     {
754         SSLModConfigRec *mc = myModConfig();
755         void *vp;

756         ssl_mutex_on(s);
757         if (table_insert_kd(mc->tSessionCacheDataTable,
758                             SCI->ucaKey, SCI->nKey,
759                             NULL, sizeof(time_t)+SCI->nData,
760                             NULL, &vp, 1) != TABLE_ERROR_NONE) {
761             ssl_mutex_off(s);
762             return FALSE;
763         }
764         memcpy(vp, &SCI->tExpiresAt, sizeof(time_t));
765         memcpy((char *)vp+sizeof(time_t), SCI->ucaData, SCI->nData);
766         ssl_mutex_off(s);
767         return TRUE;
768     }

769     void ssl_scache_shm_retrieve(server_rec *s, ssl_scinfo_t *SCI)
770     {
771         SSLModConfigRec *mc = myModConfig();
```

```
772         void *vp;
773         int n;

774         /* initialize result */
775         SCI->ucaData    = NULL;
776         SCI->nData      = 0;
777         SCI->tExpiresAt = 0;

778         /* lookup key in table */
779         ssl_mutex_on(s);
780         if (table_retrieve(mc->tSessionCacheDataTable,
781                            SCI->ucaKey, SCI->nKey,
782                            &vp, &n) != TABLE_ERROR_NONE) {
783             ssl_mutex_off(s);
784             return;
785         }

786         /* copy over the information to the SCI */
787         SCI->nData   = n-sizeof(time_t);
788         SCI->ucaData = (UCHAR *)malloc(SCI->nData);
789         if (SCI->ucaData == NULL) {
790             SCI->nData = 0;
791             ssl_mutex_off(s);
792             return;
793         }
794         memcpy(&SCI->tExpiresAt, vp, sizeof(time_t));
795         memcpy(SCI->ucaData, (char *)vp+sizeof(time_t), SCI->nData);
796         ssl_mutex_off(s);

797         return;
798     }

799     void ssl_scache_shm_remove(server_rec *s, ssl_scinfo_t *SCI)
800     {
801         SSLModConfigRec *mc = myModConfig();

802         /* remove value under key in table */
803         ssl_mutex_on(s);
804         table_delete(mc->tSessionCacheDataTable,
805                      SCI->ucaKey, SCI->nKey, NULL, NULL);
806         ssl_mutex_off(s);
807         return;
808     }

809     void ssl_scache_shm_expire(server_rec *s, time_t tNow)
```

```
{
    SSLModConfigRec *mc = myModConfig();
    table_linear_t iterator;
    time_t tExpiresAt;
    void *vpKey;
    void *vpKeyThis;
    void *vpData;
    int nKey;
    int nKeyThis;
    int nData;
    int nElements = 0;
    int nDeleted = 0;
    int bDelete;
    int rc;

    ssl_mutex_on(s);
    if (table_first_r(mc->tSessionCacheDataTable, &iterator,
                      &vpKey, &nKey, &vpData, &nData) == TABLE_ERROR_NONE) {
        do {
            bDelete = FALSE;
            nElements++;
            if (nData < sizeof(time_t) || vpData == NULL)
                bDelete = TRUE;
            else {
                memcpy(&tExpiresAt, vpData, sizeof(time_t));
                if (tExpiresAt <= tNow)
                   bDelete = TRUE;
            }
            vpKeyThis = vpKey;
            nKeyThis  = nKey;
            rc = table_next_r(mc->tSessionCacheDataTable, &iterator,
                              &vpKey, &nKey, &vpData, &nData);
            if (bDelete) {
                table_delete(mc->tSessionCacheDataTable,
                             vpKeyThis, nKeyThis, NULL, NULL);
                nDeleted++;
            }
        } while (rc == TABLE_ERROR_NONE);
    }
    ssl_mutex_off(s);
    ssl_log(s, SSL_LOG_TRACE, "Inter-Process Session Cache (SHM) Expiry: "
            "old: %d, new: %d, removed: %d", nElements, nElements-nDeleted, nDeleted);
    return;
}
```

ssl_session_cache.c

The shared memory session caching code is extremely similar to the DBM-based code. The primary difference from a programming perspective is that it uses a hash table (`table_insert_kd()` and friends) instead of DBM. However, because DBM's API is essentially a hash table, the primary difference is which functions to call.

Session storage

752–769 As with the DBM code, we first lock the cache with `ssl_mutex_on()` and then store the data. The primary difference here is that we don't have to actually open the database, as we did with DBM. Clearly, this is a significant performance improvement.

Session retrival

769–898 The same comments apply here.

Session removal

799–808 Same here.

Cache expiry

809–852 Cache expiry is substantially simpler than with DBM. Because the hash table implementation correctly handles deletions in the middle of a scan, the expiry can be accomplished in one pass. Otherwise, the code is fairly similar to the DBM code.

Appendix B

SSLv2

B.1 Introduction

The focus of this book has been on SSLv3 and TLS. However, most SSLv3 implementations are also SSLv2 implementations and some significant Web sites still support only SSLv2. This appendix thus provides an introduction to SSLv2, described in [Hickman1995]. We start by providing an overview of how SSLv2 works. However, we won't be discussing SSLv2 at anywhere near the level of detail we used for SSLv3. Rather, we'll stay at more or less the same level of abstraction we used for discussing IPSEC, S-HTTP, and S/MIME.

By the time that SSLv3 was designed, Netscape employed security professionals including the noted cryptographer Taher Elgamal. SSLv3 itself was designed with the assistance of a security consultant (Paul Kocher). However, at the time that SSLv2 was designed, Netscape (then Mosaic Communications) didn't employ any security experts. In the absence of real expertise, Kipp Hickman, an employee without notable security credentials or experience, was tasked to do the design. Unsurprisingly then, SSLv2 contained a number of security flaws which ultimately led to the design of SSLv3.

Our next order of business will be to explore the ways in which SSLv2 was flawed. First, SSLv2 is missing several features. Second, it has a number of security flaws. Though none of these is crippling for simple credit card submission, they are nevertheless quite serious. We'll discuss these problems. We've seen how they were fixed in SSLv3, but it's instructive to look at how Microsoft fixed them in *Private Communications Technology* (PCT). Finally, we'll briefly discuss what happened to SSLv1.

B.2 SSLv2 Overview

SSLv2 uses the same model as SSLv3. There is a handshake phase to establish keys and negotiate algorithms. After the handshake is completed, the keys and algorithms are used to encrypt the application data traffic. In fact, many of the messages in the SSLv2 handshake have similar names and field values to the SSLv3 handshake. (No doubt by the intent of the SSLv3 designers.) Figure B.1 shows a sample SSLv2 handshake.

Ex.1024
Askeladden, L.L.C. / Page 525 of 788

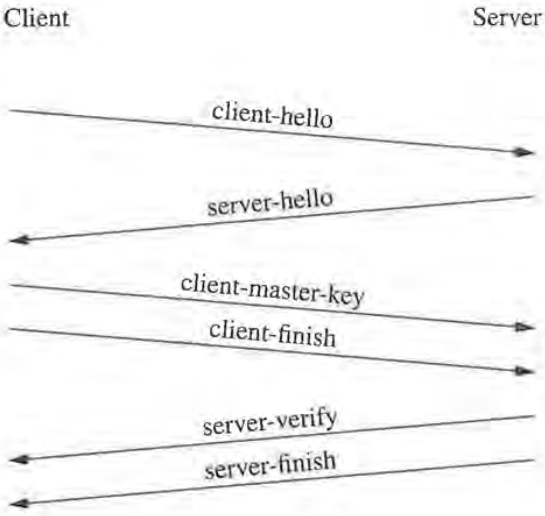


Figure B.1 SSLv2 handshake

CLIENT-HELLO

In the CLIENT-HELLO message, the client sends a list of the ciphers it supports, a random CHALLENGE, and possibly a session ID. The CHALLENGE plays much the same role as the random value in SSLv3: it provides freshness for the session keys and allows the client to detect replay attacks. As with SSLv3, the session ID is used for session resumption.

SERVER-HELLO

In the SERVER-HELLO message, the server provides its certificate as well as the subset of the client's ciphers that it supports. Unlike SSLv3, the client ultimately chooses which cipher to use. However, the server can delete any ciphers that it doesn't support. The server also provides a CONNECTION-ID which serves much the same purpose as the server random value in SSLv3.

CLIENT-MASTER-KEY

The CLIENT-MASTER-KEY provides the master key encrypted under the server's public key. In order to handle exportable ciphers, the master key is split into an encrypted section and a clear section. Thus, to do 40-bit RC4, you would encrypt 40 bits and have 88 bits in the clear. Digest functions are then used to convert the master key to encryption and MAC keys. This message also provides the IV if necessary for the cipher. The IV is transmitted in the clear.

CLIENT-FINISH

The CLIENT-FINISH message is the first encrypted message. It contains the CONNECTION-ID and proves that the client knows the encryption keys.

SERVER-VERIFY

The SERVER-VERIFY message contains the encrypted challenge. This proves that the server knows the encryption keys.

SERVER-FINISH

The SERVER-FINISH message contains the session ID. The client will use that if it wishes to resume the session.

Session Resumption

SSLv2 also has a session resumption mode. If the server recognizes the client's session ID, it sets a flag in its SERVER-HELLO. When resuming a session, the client simply omits the CLIENT-MASTER-KEY message and the client and server reuse the master key.

Client Authentication

SSLv2's client authentication is fairly simple. The server has an additional message called REQUEST-CERTIFICATE which it uses to request client authentication. The message contains a challenge that is used to generate the data signed by the client. The client responds with a CLIENT-CERTIFICATE message, which contains the client's certificate and a signature over the challenge, the client's encryption key, and the server's encryption key.

Data Transmission

SSLv2 data transmission is analogous to SSLv3 data transmission, although the details are different. The data are broken up into a series of records. Each record is invidually MACed and encrypted. The MAC is an ad hoc function rather than HMAC, but it doesn't have any known serious weaknesses.

B.3 Missing Features

SSLv2 was missing a number of features that seemed desirable. None of these features are absolutely necessary, but their absence was inconvenient and they were added when SSLv3 was designed.

Ex.1024
Askeladden, L.L.C. / Page 527 of 788

Certificate Chaining

SSLv2 permitted the client and server to send only one certificate each. Thus, this certificate had to be directly signed by the root. As we discussed in Chapter 5, it's quite convenient to have one root CA that signs a number of CAs who themselves certify users. This is not possible in SSLv2. SSLv3 allows clients and servers to have arbitrary-length certificate chains.

Same RSA Key for Export and Domestic Clients

At the time that SSLv2 was designed, U.S. export regulations forbade the use of RSA keys longer than 512 for key exchange in exportable software. However, 1024- and 512-bit keys were already standard for moderate- to high-security applications. Thus, if you wanted your server to handle strong (128-bit) ciphers, you clearly wanted to have a 1024-bit RSA key. However, because that key would also be used for key exchange, you couldn't serve export clients with the same key. Rather, your server would need two keys.

However, when Netscape first delivered SSLv2 in Navigator, they omitted the test for long keys from the export client. The *exportable* Navigator supported 1024-bit RSA. Nevertheless, they somehow got export approval. It's not clear (at least to the author) whether NSA didn't know about this problem when they reviewed Navigator, or if they just ignored it. In any case, this set a precedent and several other vendors got similar approval.

However, when SSLv3 was designed, there were indications that the NSA would be less forgiving. Thus, SSLv3 complied with the regulations by using the long RSA key to sign a 512-bit ephemeral RSA key. With this trick, it was possible both to have only one key and to comply with regulations.

B.4 Security Problems

Export Message Authentication

SSLv2 uses the same keys for message authentication as for encryption. Thus, if we're using RC4 in export mode, the encryption and MAC keys are based on only 40 bits of secret data and have 40 bits of entropy as well. Thus, in export mode, it's no more difficult to mount an integrity attack than to mount a confidentiality attack. The attacker simply exhaustively searches all 40 bits of secret master key. Exhaustively searching a 40-bit keyspace is within the means of fairly modest attackers.

By contrast, SSLv3 uses a large master key and does the entropy reduction in the key derivation stage. You can exhaustively search the encryption keys, but this doesn't allow you to work backwards to the master secret. Thus, the MAC keys are strong, even if encryption is weak—or nonexistent. In SSLv3, even if weak ciphers are used, mounting an integrity attack is intractable.

Weak MACs

Although no really good attacks are known on SSLv2's MAC, it certainly isn't as strong as HMAC. In particular, it relies solely on MD5. There's no way to use SHA-1. There's been some recent concern about the strength of MD5 [Dobbertin1996] and so this is rather undesirable, though not fatal. In this instance, at least, the designers can be forgiven somewhat. The attacks on MD5 and the development of HMAC happened after SSLv2 was published. However, not having the ability to negotiate to SHA is unnecessarily inflexible in any case.

Downgrade

SSLv2 doesn't have any protection for the handshake. Thus, it's possible for an attacker to force the parties to negotiate to a weak cipher even if they both support a stronger cipher. SSLv3 solves this problem with the Finished messages, which contain a message digest over the entire handshake.

Truncation Attacks

SSLv2 simply uses the TCP connection close to indicate end of data. This means that it's susceptible to truncation attacks. The attacker can simply forge TCP FINs and the recipient can't tell that it's not a legitimate end of data. SSLv3 fixes this problem by having an explicit closure alert.

Client Authentication Transfer

The PCT specification [Benaloh1995] describes a subtle but serious attack on SSLv2 client authentication. Consider the problem of securing a Web site where part of the content can be accessed by a normal SSL connection but part requires client authentication. Because SSLv2 message authentication is so weak when using export ciphers, the server will want to require a strong cipher for the protected sections of the Web site. Otherwise, the attacker can just crack the connection and pose as the client.

Imagine that we have a situation in which the client has already connected to the server using an export cipher. The attacker has cracked the connection by exhaustively searching the 40-bit keyspace and so knows the master key and the read/write keys. Eventually, the client is likely to reconnect to the server to make another request. The attack then proceeds as shown in Figure B.2.

The attacker intercepts the TCP connection between client and server and reads the client's CLIENT-HELLO. The attacker then opens up its own connection to the server, offering a 128-bit cipher but using the client's challenge. It sends its own SERVER-HELLO to the client agreeing to resume the connection. The attacker then sends his own CLIENT-MASTER-KEY message to the server using the same master key as in the original connection, but this time with the entire key encrypted under the server's RSA key. At this point, the client-attacker and server-attacker master keys are both the same as the master key in the original connection because the client has resumed that connection.

Ex.1024
Askeladden, L.L.C. / Page 529 of 788

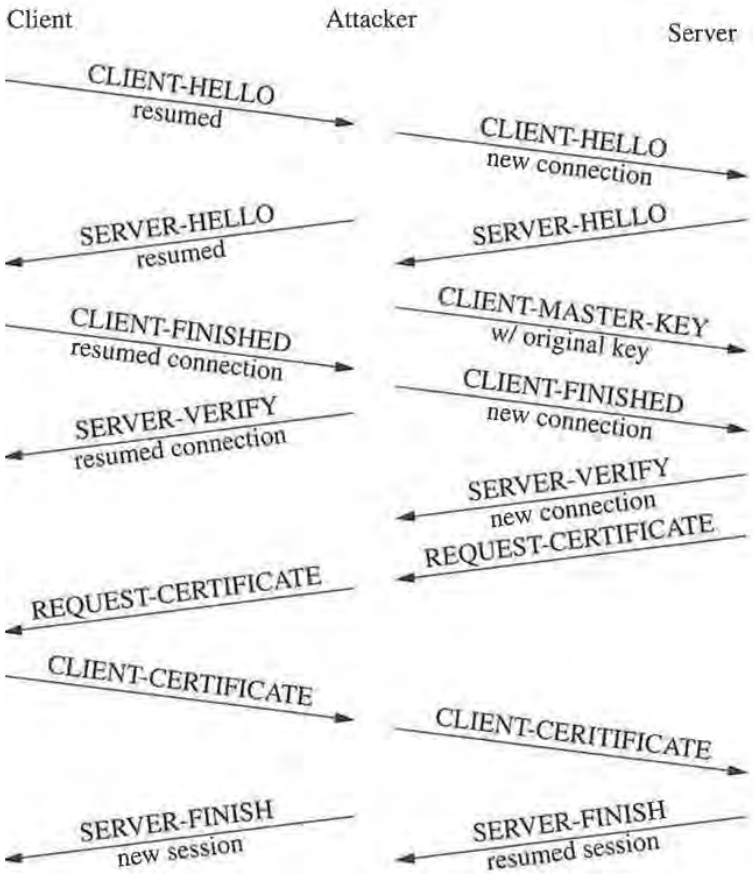


Figure B.2 SSLv2 authentication transfer attack

The server asks for client authentication using a REQUEST-CERTIFICATE message which the attacker forwards to the client. The client responds with a CLIENT-CERTIFICATE message. Recall that the data that is signed consists of the client write key, the server write key, and the challenge. All of these are the same in the client-attacker connection and the attacker-server connection. Thus, the attacker can simply forward the CLIENT-CERTIFICATE message to the server and the attacker has what appears to the server to be a client-authenticated connection with strong encryption—except that actually it's a connection to the attacker. This attack is impossible in SSLv3 and TLS because of the separation of authentication keys from encryption keys.

B.5 PCT

Microsoft's response to the security errors in SSLv2 was to design their own protocol, a variant of SSLv2 called *Private Communications Technology* (PCT). PCT fixed the most severe security flaws in SSLv2. In particular, it fixed the downgrade and client authentication vulnerabilities. It also simplified the handshake somewhat, thus reducing the number of round trips that were required in order to establish a connection. Finally, it incorporated more flexible cipher suite selection.

Verify Prelude

SSLv3 used the Finished messages to prevent a downgrade attack by an active attacker. PCT uses a verify-prelude field in the CLIENT-MASTER-KEY message to provide the same protection. The verify-prelude contains a MAC of the client and server hello messages. Thus, the server can detect if the connection has been downgraded. Because the verify-prelude is sent with the key exchange, the server can immediately reject the connection if any tampering has occurred—unlike with SSLv3, where the entire connection has to be completed before the attack can be detected.

Strengthened Message Integrity

PCT made message authentication orthogonal to security. As with SSLv3, a long master key was transmitted and this key was used to generate the MAC and encryption keys independently. Even if a 40-bit encryption key was being used, the MAC key was full strength. Thus, even if the encryption keys were broken, message integrity would be intact.

Improved Client Authentication

The authentication transfer attack on SSLv2 depended on the CLIENT-CERTIFICATE message being the same no matter what cipher suite was used. PCT fixed this attack by having the client's certificate sign the verify-prelude field. Because this field depends on the cipher suite, the transfer attack no longer works. Moreover, the transfer attack depends on being able to recover the master secret for a connection using a 40-bit cipher suite. This is no longer possible in PCT because the master key is always long even if export encryption is used.

Orthogonal Cipher Suite Negotiation

Whereas SSLv2, SSLv3, and TLS all have a single cipher suite value which defines all the cryptographic parameters in use, PCT specifies the cipher, hash, certificate, and key exchange algorithms separately. Moreover, the cipher specifications are themselves orthogonal so that an encryption algorithm can be specified independently of the key

length. The advantage of this approach is that it's not necessary to register a cipher suite for every desired combination of algorithms.

The argument used by the SSLv3 designers against this approach was that specifying algorithms orthogonally allowed inappropriate or insecure combinations to be constructed. I don't find this argument that compelling. Even when cipher suites are monolithic, it's possible to create inappropriate combinations (4096-bit RSA keys with DES, for instance). By contrast, we've seen in previous chapters that the failure to specify cipher suites orthogonally in SSLv3 led to some useful combinations not being registered (e.g., DH/DSS with RC4).

Backward Compatibility

As with SSLv3, it was desirable that PCT be backward-compatible with SSLv2. Thus, the PCT CLIENT-HELLO had to be representable as an SSLv2 CLIENT-HELLO. However, because Microsoft didn't have control of the version number space, they couldn't just use version 3 as Netscape did. Instead, PCT was forced to use a hack: the designers specified a new cipher suite that indicated that the client supported PCT. Thus, an SSLv2 server that saw this cipher suite wouldn't recognize it and would continue with SSLv2, but a PCT server would be able to do the PCT handshake.

The second backward-compatibility problem was support for PCT's orthogonal algorithm selection. The approach used by the designers was to represent the algorithms all as SSLv2 cipher suites and then to use bit masks to distinguish the various classes. Figure B.3 shows a dump of part of a PCT-compatible CLIENT-HELLO.

This CLIENT-HELLO is actually compatible with SSLv2, PCT, and SSLv3/TLS. Thus SSLv3/TLS cipher suites (represented as SSLv2 cipher suites) also appear in this message. Line 1 indicates that the client speaks PCT. Lines 2–3 indicate that it supports both X.509 certificate chains as well as bare X.509 certificates. Lines 4–5 indicate that both MD5 and SHA are supported. Line 6 indicates support for RSA key exchange.

Pay special attention to lines 10–11 and lines 21–22. The PCT specifications for the encryption and MAC algorithms are four bytes long and so must be represented as two SSLv2 cipher suites, with the first half appearing in one and the second in the next. Thus lines 10–11 specify 128-bit RC4 with a 128-bit MAC. This hack is required because the orthogonal PCT cipher suites can't fit into the 3-byte space provided by SSLv2.

Bottom Line

At the time that PCT was introduced, Netscape was by far the dominant Web browser and so SSLv3 became far more important than PCT. Thus, no major vendor other than Microsoft implemented PCT. Although Microsoft Internet Explorer eventually became the dominant Web browser, the defeat of Netscape happened far too late to save PCT. Although it's arguable that PCT embodies more elegant design choices than SSLv3/TLS, it doesn't contain any features sufficiently important to justify using it in face of the far wider deployment of SSLv3.

```
1    8f 80 01    PCT_SSL_COMPAT | PCT_VERSION_1
2    80 00 03    PCT_SSL_CERT_TYPE | PCT1_CERT_X509_CHAIN
3    80 00 01    PCT_SSL_CERT_TYPE | PCT1_CERT_X509
4    81 00 01    PCT_SSL_HASH_TYPE | PCT1_HASH_MD5
5    81 00 03    PCT_SSL_HASH_TYPE | PCT1_HASH_SHA
6    82 00 01    PCT_SSL_EXCH_TYPE | PCT1_EXCH_RSA_PKCS1
7    00 00 04    TLS_RSA_WITH_RC4_128_MD5
8    00 00 05    TLS_RSA_WITH_RC4_128_SHA
9    00 00 0a    TLS_RSA_WITH_3DES_EDE_CBC_SHA
10   83 00 04    PCT_SSL_CIPHER_TYPE_1ST_HALF | PCT1_CIPHER_RC4
11   84 80 40    PCT_SSL_CIPHER_TYPE_2ND_HALF | PCT1_ENC_BITS_128 | PCT1_MAC_BITS_128
12   01 00 80    SSL2_RC4_128_WITH_MD5
13   07 00 c0    SSL2_DES_192_EDE3_CBC_WITH_MD5
14   03 00 80    SSL2_RC2_128_CBC_WITH_MD5
15   00 00 09    TLS_RSA_WITH_DES_CBC_SHA
16   06 00 40    SSL2_DES_64_CBC_WITH_MD5
17   00 00 64    TLS_RSA_EXPORT1024_WITH_RC4_56_SHA
18   00 00 62    TLS_RSA_EXPORT1024_WITH_DES_CBC_SHA
19   00 00 03    TLS_RSA_EXPORT_WITH_RC4_40_MD5
20   00 00 06    TLS_RSA_EXPORT_WITH_RC2_CBC_40_MD5
21   83 00 04    PCT_SSL_CIPHER_TYPE_1ST_HALF | PCT1_CIPHER_RC4
22   84 28 40    PCT_SSL_CIPHER_TYPE_2ND_HALF | PCT1_ENC_BITS_40 | PCT1_MAC_BITS_128
23   02 00 80    SSL2_RC4_128_EXPORT40_WITH_MD5
24   04 00 80    SSL2_RC2_128_CBC_EXPORT40_WITH_MD5
25   00 00 13    TLS_DHE_DSS_WITH_3DES_EDE_CBC_SHA
26   00 00 12    TLS_DHE_DSS_WITH_DES_CBC_SHA
27   00 00 63    TLS_DHE_DSS_EXPORT1024_WITH_DES_CBC_SHA
```

Figure B.3 PCT CLIENT-HELLO

B.6 What about SSLv1?

At this point, you're probably thinking "I've heard about SSLv3 and SSLv2, but what about SSLv1?" Netscape never actually delivered a specification or a public implementation of SSL version 1. We've just seen that SSLv2 contains a number of modest security flaws that could have been avoided by careful engineering. The drafts of SSLv1 that circulated for review had all these security flaws but also contained a very serious security flaw: weak integrity protection.

The very first SSLv1 draft didn't have any message integrity protection at all. Almost unbelievably, it also used RC4, with the effect that an attacker could make predictable changes in the plaintext. Moreover, it didn't have any sequence numbers, so it was totally vulnerable to replay attack.

Later versions added a sequence number and a checksum, but used a *cyclic redundancy check* (CRC) as the checksum. Unfortunately, CRC isn't non-invertible or collision free the way that MD5 is, so it didn't prevent message integrity attacks when RC4 was used. Finally, right before releasing SSLv2, the designers figured out about integrity attacks and changed the MAC to MD5, which solved the problem. Credit for these changes largely goes to Martin Abadi, who pointed out the problems to Netscape.

Bibliography

All RFCs are freely downloadable via the Web as described in Section 2.2.

[Abbott1988] Abbott, S., and Keung, S., *CryptoSwift (ver.2) Performance on Netscape Enterprise Server* (April, 1988).
`http://isglabs.rainbow.com/isglabs/NS351-CSv2-NT-perf/NS351-CSv2.html`
Describes performance behavior of a commercial (though somewhat outdated) server with and without hardware acceleration.

[Anderson1999] Anderson, R., Biham, A., and Knudsen, L., *Serpent: A Proposal for the Advanced Encryption Standard* (1999).
`http://csrc.nist.gov/encryption/aes/round2/AESAlgs/Serpent/Serpent.pdf`

[ANSI1985] ANSI, "American National Standard for Financial Institution Key Management (wholesale)," ANSI X9.17 (1985).
Describes 3DES.

[ANSI1986] ANSI, "American National Standard for Financial Institution Message Authentication (Wholesale)," ANSI X9.9 (Revised) (1986).
One of many descriptions of DES-CBC MAC.

[ANSI1995] ANSI, "Public key cryptography for the financial services industry — Certificate management," ANSI X9.57 (1995).

[ANSI1998] ANSI, "Agreement of Symmetric Keys Using Diffie-Hellman and MQV Algorithms," ANSI X9.42 draft (1998).

[Atkins1996] Atkins, D., Stallings, W., and Zimmermann, P., "PGP Message Exchange Formats," RFC 1991 (August 1996).

[Balenson1993] Balenson, D., "Privacy Enhancement for Internet Electronic Mail: Part III: Algorithms, Modes, and Identifiers," RFC 1423 (February 1993).

[Banes1999] Banes, J., and Harrington, R., "56-bit Export Cipher Suites for TLS," draft-ietf-tls-56-bit-ciphersuites-00.txt (April 1999).

[Banes2000] Banes, J., *Personal communication.* (2000).

Ex.1024
Askeladden, L.L.C. / Page 535 of 788

[Bellare1995] Bellare, M., and Rogaway, P., "Optimal asymmetric encryption padding," *Advances in Cryptology — Eurocrypt 94 Proceedings*, Springer-Verlag, Berlin (1995).

[Bellovin1995] Bellovin, S.M., "Using the Domain Name System for System Break-ins" in *5th USENIX UNIX Security Symposium*, p. 199–208, USENIX, Salt Lake City, UT (June 5–7, 1995).

[Benaloh1995] Benaloh, J., Lampson, B., Simon, D., Spies, T., and Yee, B., "Private Communication Technology Protocol," draft-microsoft-PCT-01.txt (September 1995).
The second PCT draft

[Berners–Lee1994] Berners–Lee, T., Masinter, L., and McCahill, M., "Uniform Resource Locators," RFC 1738 (December 1994).

[Berners-Lee1998] Berners-Lee, T., Fielding, R., and Masinter, L., "Uniform Resource Identifiers (URI)," RFC 2396 (August 1998).

[Biham1991a] Biham, E., Shamir, A., and Differential Cryptanalyis of DES-like Cryptosystems, *Advances in Cryptology—CRYPTO '90 Proceedings*, p. 2–21, Springer-Verlag, Berlin (1991).

[Biham1991b] Biham, E., Shamir, A., and Differential Cryptanalyis of DES-like Cryptosystems, *Journal of Cryptology*, 4, 1, p. 3–72 (1991).

[Biham1993a] Biham, E., and Shamir, A., "Differential Analysis of the Full 16-Round DES," *Advances in Cryptology—CRYPTO '92 Proceedings*, p. 487–496, Springer-Verlag, Berlin (1993).

[Biham1993b] Biham, E., and Shamir, A., *Differential Analysis of the Data Encryption Standard*, Springer-Verlag, New York, N.Y. (1993).

[Blaze1996] Blaze, M., Diffie, W., Rivest, R., Schneier, B., Shimomura, T., Thompson, E., and Weiner, M., *Minimal Key Lengths for Symmetric Ciphers to Provide Adequate Commercial Security* (January 1996).

[Blaze1999] Blaze, M., Feigenbaum, J., Ionnidis, J., and Keromytis, A., "The Keynote Trust Management System, Version 2," RFC 2704 (September 1999).

[Bleichenbacher1998] Bleichenbacher, D., "Chosen Ciphertext Attacks against Protocols Based on RSA Encryption Standard PKCS #1," *Advances in Cryptology — CRYPTO 98*, p. 1–12 (1998).

[Bleichenbacher1999] Bleichenbacher, D., *Personal communication.* (1999).

[Boe1999] Boe, M., "TLS-based Telnet Security," draft-ietf-tn3270-telnet-tls-03.txt (October, 1999).
Telnet over TLS has never been standardized, but this document describes the best current practice.

[Burwick1999] Burwick, C., Coppersmith, D., D'Avignon, E., Genarro, R., Halevi, S., Jutla, C., Matyas, S.M. Jr., O'Connor, L., Peyravian, M., Safford, D., and Zunic, N., *MARS, a Candidate Cipher for AES*, IBM Corporation (September 1999).
`http://csrc.nist.gov/encryption/aes/round2/AESAlgs/MARS/mars.pdf`

[Cocks1973] Cocks, C., "A Note on Non-Secret Encryption," CESG Report, CESG (1973).
http://www.cesg.gov.uk/about/nsecret/notense.htm
Describes CESG's invention of what we call RSA.

[Crispin1996] Crispin, M., "Internet Message Access Protocol—Version 4rev1," RFC 2060 (December 1996).

[Crocker1982] Crocker, D., "Standard for the Format of ARPA Internet Text Messages," RFC 822 (August 1982).
The basic standard for internet email messages.

[Daemen1999] Daemen, J., and Rijmen, V., *AES Proposal: Rijndael* (March 1999).
http://csrc.nist.gov/encryption/aes/round2/AESAlgs/Rijndael/↵
Rijndael.pdf

[Dai2000] Dai, W., *Crypto++ 3.1 Benchmarks* (2000).
http://www.eskimo.com/~weidai/benchmarks.html
Wei Dai's benchmarks for his Crypto++ package.

[Dierks1999] Dierks, T., and Allen, C., "The TLS Protocol Version 1.0," RFC 2246 (January 1999).

[Diffie1976] Diffie, W., and Hellman, M.E., "New directions in cryptography," *IEEE Transactions on Information Theory*, 22, 6, p. 655–654 (1976).

[Diffie1992] Diffie, W., van Oorschot, P.C., and Wiener, M.J., "Authentication and Authenticated Key Exchanges," *Designs, Codes, and Cryptography*, 2, p. 102–125 (1992).
Describes the Station to Station protocol (STS), a form of authenticated DH upon which IKE is based.

[Dobbertin1996] Dobbertin, H., "The Status of MD5 After a Recent Attack," *CryptoBytes*, 2, 2, RSA Laboratories (Summer 1996).

[Dusse1998] Dusse, S., Hossman, P., Ramsdell, B., Lundblade, L., and Repka, L., "S/MIME Version 2 Message Specification," RFC 2311 (March 1998).

[Eastlake1994] Eastlake, D. 3rd., Crocker, S., and Schiller, J., "Randomness Recommendations for Security," RFC 1750 (December 1994).

[Eastlake1999] Eastlake, D., 3rd., "Domain Name System Security Extensions," RFC 2535 (March 1999).

[Ellis1970] Ellis, J.H., *The Possibility of Non-Secret Encryption*, CESG (1970).
http://www.cesg.gov.uk/about/nsecret/possnse.htm

[Ellis1987] Ellis, J.H., *The Story of Non-Secret Encryption*, CESG (1987).
http://www.cesg.gov.uk/about/nsecret/ellis.htm

[Ellison1999] Ellison, C., Frantz, B., Lampson, B., Rivest, R., Thomas, B., and Ylonen, T., "SPKI Certificate Theory," RFC 2693 (September 1999).

[Fielding1999] Fielding, R., Gettys, J., Mogul, J., Frystyk, H., Masinter, L., Leach, P., and Berners-Lee, T., "Hypertext Transfer Protocol," RFC 2616 (June 1999).

[Ford-Hutchinson2000] Ford-Hutchinson, Paul, Carpenter, M., Hudson, T., Murray, E., and Wiegand, V., "Securing FTP with TLS," draft-murray-auth-ftp-ssl-05.txt (January 2000).
FTP over TLS has never been standardized. This document describes the de facto standard.

Ex.1024
Askeladden, L.L.C. / Page 537 of 788

[Freed1996a] Freed, N., and Borenstein, N., "Multipurpose Internet Mail Extensions (MIME) Part One: Format of Internet Message Bodies," RFC 2045 (November 1996).

[Freed1996b] Freed, N., and Borenstein, N., "Multipurpose Internet Mail Extensions (MIME) Part Two: Media Types," RFC 2046 (November 1996).

[Freed1996c] Freed, N., Klensin, J., and Postel, J., "Multipurpose Internet Mail Extensions (MIME) Part Four: Registration Procedures," RFC 2048 (November, 1996).

[Freed1996d] Freed, N., and Borenstein, N., "Multipurpose Internet Mail Extensions (MIME) Part Five: Conformance Criteria and Examples," RFC 2049 (November 1996).

[Freier1996] Freier, A.O., Karlton, P., and Kocher, P.C., *The SSL Protocol Version 3.0* (November 1996).
`http://home.netscape.com/eng/ssl3/draft302.txt`
The last published draft of SSLv3. Note that SSLv3 was never published as an IETF RFC of any kind. This document combined with Netscape's implementation represent the de facto standard.

[Gilmore1998] Gilmore, J. (Ed.), *Cracking DES: Secrets of Encryption Research, Wiretap Politics & Chip Design,* O'Reilly & Associates (May 1998).
Describes how to build a dedicated hardware DES search engine.

[Goland1999] Goland, Y., Whitehead, E., Faizi, A., Carter, S., and Jensen, D., "HTTP Extensions for Distributed Authoring—WEBDAV," RFC 2518 (February 1999).

[Goldberg1996] Goldberg, I., and Wagner, D., "Randomness and the Netscape Browser," *Dr. Dobb's Journal* (January 1996).
The paper that described the weakness in Netscape PRNG seeding.

[Harkins1998] Harkins, D., Carrel, D., and The Internet Key Exchange (IKE), RFC 2409 (November 1998).

[Hennessey1996] Hennessey, J., Goldberg, D., and Patterson, D.A., *Computer Architecture: A Quantitative Approach, 2ed.*, Morgan Kaufmann (January 1996).
The standard book on processor design by some of the inventors of RISC.

[Hickman1995] Hickman, K., *The SSL Protocol* (February 1995).
`http://www.netscape.com/eng/security/SSL_2.html`
The specification for SSLv2.

[Hoffman1999a] Hoffman, P., "SMTP Service Extension for Secure SMTP over TLS," RFC 2487 (January 1999).
The standard for SMTP over TLS.

[Hoffman1999b] Hoffman, P., "Enhanced Security Services for S/MIME," RFC 2634 (June 1999).

[Housley1999a] Housley, R., Ford, W., Polk, W., and Solo, D., "Internet X.509 Public Key Infrastructure Certificate and CRL Profile," RFC 2459 (January 1999).

[Housley1999b] Housley, R., "Cryptographic Message Syntax," RFC 2630 (June 1999).
CMS is a variant of PKCS #7 that has been extended to support Diffie-Hellman. and DSA.

[IANA] IANA, *Well Known Ports.*
`http://www.iana.org/numbers.htm`
A list of all the assigned well known ports.

[ITU1988a] ITU, "The Directory—Authentication Framework," ITU Recommendation X.509 (1988).

[ITU1988b] ITU, "The Directory—Models," ITU Recommendation X.500 (1988).

[ITU1988c] ITU, "Specification of Abstract Syntax Notation One (ASN.1)," ITU Recommendation X.208 (1988).

[ITU1988d] ITU, "Specification of Basic Encoding Rules for Abstract Syntax Notation One (ASN.1)," ITU Recommendation X.209 (1988).

[Jablon1996] Jablon, D., "Strong Password-only Authenticate Key Exchange," *ACM Computer Communications Review,* 26, 5 (October 1996).

[Jacobsen1988] Jacobsen, V., "Congestion Avoidance and Control," *Computer Communication Review,* 18, 4, p. 314–329 (August 1988).

[JavaSoft1999] JavaSoft, *Java Secure Socket Extension (JSSE 1.0)* (1999).
`http://java.sun.com/products/jsse/`
Sun's free "non-commercial" quality implementation of SSL/TLS.

[Johnson1993] Johnson, D.B., Matyas, S.M., Le, A.V., and Wilkins, J.D., "Design of the Commercial Data Masking Facility Data Privacy Algorithm" in *1st ACM Conference on Computer and Communications Security,* p. 93–96, ACM Press (1993).

[Joncheray1995] Joncheray, L., "A Simple Active Attack Against TCP" in *5th USENIX UNIX Security Symposium,* p. 7–19, USENIX, Salt Lake City, UT (June 5-7, 1995).

[Kaliski1993] Kaliski, B., "Privacy Enhancement for Internet Electronic Mail: Part IV: Key Certification and Related Services," RFC 1424 (February 1993).

[Kaliski1998a] Kaliski, B., and Staddon, J., "PKCS #1: RSA Cryptography Specifications Version 2.0," RFC 2437 (October 1998).

[Kaliski1998b] Kaliski, B.S., Jr., "Compatible cofactor multiplication for Diffie-Hellman primitives," *Electronics Letters,* 34, 25, p. 2396–2397 (December 1998).

[Kaufman1995] Kaufman, C., Perlman, R., and Speciner, M., *Network Security: Private Communications in a Public World,* Prentice-Hall, Englewood Cliffs, NJ (1995).
An excellent introduction to cryptography and network security. Focuses on the application of cryptography to communications security.

[Kelsey1999] Kelsey, J., Schneier, B., and Ferguson, N., "Notes on the Design and Analysis of the Yarrow Cryptographic Pseudorandom Number Generator" in *Sixth Annual Workshop on Selected Areas in Cryptography,* Springer-Verlag, Berlin (August 1999).

[Kent1993] Kent, S., "Privacy Enhancement for Internet Electronic Mail: Part II: Certificate-Based Key Management," RFC 1422 (February 1993).

[Kent1998a] Kent, S., and Atkinson, R., "IP Authentication Header," RFC 2402 (November, 1998).

[Kent1998b] Kent, S., Atkinson, R., and RFC 2406, *IP Encapsulating Security Payload (ESP)* (November 1998).

[Kent1998c] Kent, S., and Atkinson, R., "Security Architecture for the Internet Protocol," RFC 2401 (November 1998).

[Keung] Keung, S., *Cryptoswift performance under SSL with file transfer* (19XX).
`http://isglabs.rainbow.com/isglabs/SSLperformance/SSL+file%20performance.html`

[Khare2000] Khare, R., and Lawrence, S., "Upgrading to TLS Within HTTP/1.1," RFC 2817 (May 2000).

[Klein1990] Klein, D.V., *"Foiling the Cracker": A Survey of and Improvements to Password Security* (1990).
A good description of how easy it is to crack passwords when users choose them.

[Klensin1995] Klensin, J., Freed, N., Rose, M., Stefferud, E., and Crocker, D., "SMTP Service Extensions," RFC 1869 (November 1995).

[Kocher1996a] Kocher, P., *A Quick Introduction to Revocation Trees* (1996).
`http://www.valicert.com/pdf/Certificate_revocation_trees.pdf`

[Kocher1996b] Kocher, P., *Timing Attacks on Implementation of Diffie-Hellman, RSA, DSS, and Other Systems* (1996).

[Kocher1999] Kocher, P., and Jun, B., *The Intel Random Number Generator* (April 1999).
This paper describes an analysis of the Intel hardware PRNG in the Pentium III.

[Krawczyk1995] Krawczyk, H., *SKEME: A Versatile Secure Key Exchange Mechanism for Internet* (August 1995).

[Krawczyk1996] Krawczyk, H., *Personal communication.* (1996).
HMAC is believed to be immune to Dobbertin's attacks on MD5.

[Krawczyk1997] Krawczyk, H., Bellare, M., and Canetti, R., "HMAC: Keyed-Hashing for Message Authentication," RFC 2104 (February 1997).

[Lear1994] Lear, E., Fair, E., and Kessler, T., "Network 10 Considered Harmful (Some Practices Shouldn't be Codified)," RFC 1627 (June 1994).
The RFC that launched an extremely acrimonious debate over NAT.

[Lim1997] Lim, C.H., and Lee, P.J., "A key recovery attack on discrete log-based schemes using a prime order subgroup" in *Advances in Cryptology—Crypto 97,* p. 249–263, Springer-Verlag, Berlin (1997).

[Linn1993] Linn, J., "Privacy Enhancement for Internet Electronic Mail: Part I: Message Encryption and Authentication Procedures," RFC 1421 (February 1993).

[Maughan1998] Maughan, D., Schertler, M., Schneider, M., and Turner, J., "Internet Security Association and Key Management Protocol (ISAKMP)," RFC 2408 (November 1998).

[Medvinsky1999] Medvinsky, A., and Hur, M., "Addition of Kerberos Cipher Suites to Transport Layer Security (TLS)," RFC 2712 (October 1999).

[Menezes1996] Menezes, A.J., van Oorschot, P.C., and Vanstone, S.A., *Handbook of Applied Cryptography,* CRC Press, Boca Raton, FL (1996).
Not a useful introduction but an extremely useful technical reference to cryptography.

[Microsoft2000] Microsoft, *Security Support Provider Interface* (2000).
`http://msdn.microsoft.com/library/default.asp?URL=/library/psdk/secspi/portalsspi_1545.htm`
Documents Microsoft's built-in security services including SChannel.

[Miller1987] Miller, S., Neumann, B., Schiller, J., and Saltzer, J., "Kerberos Authentication and Authorization System," *Project Athena Technical Plan,* MIT Project Athena (December 1987).

[Mockapetris1987a] Mockapetris, P.V., "Domain Names—Concepts and Facilities," RFC 1034 (November 1987).

[Mockapetris1987b] Mockapetris, P.V., "Domain Names—Implementation and Specification," RFC 1035 (November 1987).

[Moeller1998] Moeller, B., "Export-PKC attacks on SSL 3.0/TLS 1.0," *Message to IETF-TLS mailing list* (October 1998).
`http://www.imc.org/ietf-tls/mail-archive/msg01671.html`

[Mogul1995] Mogul, Jeffrey C., "The Case for Persistent-Connection HTTP," Research Report 95/4 (May 1995).
`http://www.research.digital.com/abstracts/95.4.html`
An early paper showing the benefits of HTTP retained connections.

[Moore1996] Moore, K., "MIME (Multipurpose Internet Mail Extensions) Part Three: Message Header Extensions for Non-ASCII," RFC 2047 (November 1996).

[Myers1996] Myers, J., and Rose, M., "Post Office Protocol—Version 3," RFC 1939 (May 1996).

[Myers1999] Myers, M., Ankney, R., Malpani, A., Galperin, S., and Adams, C., " X.509 Internet Public Key Infrastructure Online Certificate Status Protocol—OCSP," RFC 2560 (June 1999).

[Nagle1984] Nagle, J., "Congestion Control in IP/TCP Internetworks," RFC 0896 (Jan 1984).

[Needham1978] Needham, R.M., and Schroeder, M.D., "Using Encryption for Authentication in Large Networks of Computers," *Communications of the ACM,* 21, p. 993–999 (December 1978).

[Netcraft2000] Netcraft, *Netcraft Secure Web Server Survey* (January 2000).
Netcraft kindly provided me with a copy of this survey.

[Netscape1995a] Netscape Communications Corp, *SSL 2.0 Certificate Usage* (1995).
`http://www.netscape.com/eng/security/ssl_2.0_certificate.html`
Describes Netscape's wildcarding technique.

[Netscape1999a] Netscape Communications Corp., *Netscape Certificate Extensions, Communicator 4.0 Version* (1999).
`http://www.netscape.com/eng/security/comm4-cert-exts.html`

[Neumann1951] von Neumann, J., "Various Techniques Used in Connection with Random Digits," *Applied Mathematics Series*, 12, p. 36–38, U.S. National Bureau of Standards (1951).

[Newman1999] Newman, C., "Using TLS with IMAP, POP3 and ACAP," RFC 2595 (June 1999).
Provides upward negotiation mechanisms for IMAP, POP and ACAP.

[NIST1993a] National Institute of Standards and Technology (NIST), "Data Encryption Standard," FIPS PUB 46-2, U.S. Department of Commerce (December 1993).
This is the reissued DES document. It's essentially identitical to the document published in 1977.

[NIST1994a] National Institute of Standards and Technology (NIST), and Secure Hash Standard, FIPS PUB 180-1, U.S. Department of Commerce (May 1994).
The revised SHA draft that describes SHA-1.

[NIST1994b] National Institute of Standards and Technology, "Security Requirements for Cryptographic Modules," FIPS PUB 140-1, U.S. Department of Commerce (January 1994).
Describes four levels of secure modules, ranging from Level 1 (requiring approved algorithms but running on general purpose computers) to Level 4 (full tamperproofing.)

[Orman1998] Orman, H., "The OAKLEY Key Determination Protocol," RFC 2412 (November 1998).

[Padmanabhan1995] Padmanabhan, V.N., "Improving World Wide Web Latency," UCB/CSD 95-875, Computer Science Division, University of California, Berkeley (May 1995).

[Postel1982] Postel, J., "Simple Mail Transfer Protocol," RFC 821 (August 1982).
The base standard for Internet mail transport.

[Postel1985] Postel, J., and Reynolds, J.K., "File Transfer Protocol," RFC 959 (October 1985).

[Postel1991a] Postel, J., "Internet Protocol," RFC 791 (September 1991).
The IETF standard for IP.

[Postel1991b] Postel, J., "Internet Control Message Protocol," RFC 792 (September 1991).

[Postel1991c] Postel, J., "Transmission Control Protocol," RFC 793 (September 1991).
The IETF standard for TCP.

[Ramsdell1999] Ramsdell, B., "S/MIME Version 3 Message Specification," RFC 2633 (June 1999).

[Rekhter1994] Rekhter, Y., Moskowitz, B., Karrenberg, D., de Groot, G.J., and RFC 1597, *Address Allocation for Private Internets* (March 1994).
The original RFC describing NAT and Network 10.

[Rekhter1996] Rekhter, Y., Moskowitz, B., Karrenberg, D., de Groot, G.J., and Lear, E., RFC 1918 (February 1996).
This document represents a truce of sorts in the NAT debate.

[Relyea1996] Relyea, B., *Appendix A—SSL Protocol Version 3.0 Specification Errata for Fortezza Implementations* (November, 1996).
`http://www.armadillo.huntsville.al.us/Fortezza_docs/ssl_fortezza.pdf`
Describes how to make SSLv3 work with FORTEZZA.

[Rescorla1999a] Rescorla, E., and Schiffman, A., "The Secure HyperText Transfer Protocol," RFC 2660 (August 1999).

[Rescorla1999b] Rescorla, E., and Schiffman, A., "Security Extensions for HTML," RFC 2659 (August 1999).

[Rescorla2000] Rescorla, E., "HTTP over TLS," RFC 2818 (May 2000).

[Rivest1979] Rivest, R.L., Shamir, A., and Adelman, L.M., "On Digital Signatures and Public Key Cryptosystems," Technical Report, MIT/LCS/TR-212, MIT Laboratory for Computer Science (January 1979).

[Rivest1983] Rivest, R., Shamir, A., and Adleman, L.M., "Cryptographic communications system and method," US Patent 4405829 (September 1983).
The RSA Patent.

[Rivest1992] Rivest, R., "The MD5 Message-Digest Algorithm," RFC 1321 (April 1992).

[Rivest1995] Rivest, R., Robshaw, M.J.B., Sidney, R., and Yin, Y.L., *The RC6™ Block Cipher* (August 1995).
`http://csrc.nist.gov/encryption/aes/round2/AESAlgs/RC6/cipher.pdf`

[Rivest1998] Rivest, R., "A Description of the RC2(r) Encryption Algorithm," RFC 2268 (January 1998).

[RSA1993a] Kaliski, B.S., Jr., "A Layman's Guide to a Subset of ASN.1, BER, and DER," Technical Note, RSA Laboratories (November 1993).
Provides a readable introduction to ASN.1, BER, and DER.

[RSA1993b] RSA Laboratories, "RSA Encryption Standard," PKCS #1 (November 1993).

[RSA1993c] RSA Laboratories, "Cryptographic Message Syntax Version 1.5," PKCS #7 (November 1993).

[RSA1993d] RSA Laboratories, "Password Based Encryption Standard," PKCS #5 (November 1993).

[RSA1999a] RSA Laboratories, "Personal Information Exchange Syntax," PKCS #12 (June 1999).

[RSA1999b] RSA Laboratories, "Password Based Encryption Standard," PKCS #5v2.0 (March 1999).

[Saltzer1984] Saltzer, J.H., Reed, D.P., and Clark, D.D., "End-to-End Arguments in System Design," *ACM Transactions in Computer Systems*, 2, 4, p. 277–288 (November 1984).
The classic description of the end-to-end argument, a basic networking design principle.

[Schneier1996a] Schneier, B., *Applied Cryptography, 2ed.*, John Wiley & Sons, New York, N.Y. (1996).
The standard text on cryptography.

[Schneier1996b] Schneier, B., and Wagner, D., "Analysis of the SSL 3.0 Protocol," *The Second USENIX Workshop on Electronic Commerce Proceedings*, p. 29–40, USENIX Press (November 1996).

[Schneier1998] Schneier, B., Kelsey, J., Whiting, D., Wagner, D., Hall, C., and Ferguson, N., *Twofish: A 128-Bit Block Cipher* (June 1998).
http://csrc.nist.gov/encryption/aes/round2/AESAlgs/Twofish/Twofish.pdf

[Schnorr1991] Schnorr, K., "Method for Identifying Subscribers and for Generating and Verifying Electronic Signatures in a Data Exchange System," US Patent 4995082 (Feb 1991).

[Shamir1999] Shamir, A., "Factoring Large Numbers with the TWINKLE Device," *Eurocrypt '99 Rump Session* (1999).

[Spero1994] Spero, S., *Analysis of HTTP Performance Problems* (1994).
http://sunsite.unc.edu/mdma-release/http-prob.html

[Stevens1994] Stevens, W.R., *TCP/IP Illustrated, Volume 1: The Protocols*, Addison-Wesley, Reading, MA (1994).
The classic book on TCP/IP.

[Voydock1983] Voydock, V., and Kent, S.T., "Security mechanisms in high-level network protocols," *ACM Computing Surveys*, 15, p. 135–171 (1983).
An early survey of various mechanisms for providing cryptographic security in networks. Voydock and Kent discuss many of the same issues that we've seen in this chapter in the context of early network protocols. Kent has been a key figure in the design of a number of important public and classified security protocols, including IPsec and PEM.

[W3C2000] W3C, *Naming and Addressing: URIs, URLs, ...* (2000).
http://www.w3.org/Addressing/
A good guide to the relationship between URIs and URLs.

[WAP1999a] Wireless Application Protocol Forum, *WAP WTLS* (Nov 1999).

[Williamson1974] Williamson, M., *Non-Secret Encryption Using a Finite Field*, CESG (1974).
http://www.cesg.gov.uk/about/nsecret/secenc.htm
Describes CESG's invention of a system which is essentially Diffie-Hellman.

[Williamson1976] Williamson, M., *Thoughts on Cheaper Non Secret Encryption*, CESG (1976).
http://www.cesg.gov.uk/about/nsecret/cheapnse.htm

[Wu1998] Wu, T., "The Secure Remote Password Protocol," *Proceedings of the 1998 Internet Society Network and Distributed Systems Security Symposium*, p. 97–111 (March 1998).

Index

As is the case with many technical books, a large number of the terms we use are acronyms. In this index, the primary entry for each term is indexed under the acronym with the expansion in parentheses. The index entry for the expanded term refers the reader to the acronym. For convenient reference, a list of important acronyms with the page where they are first explained follows the index.

1024-bit
  export cipher suites, 73
  key exchange, export, 25
2DES, 30
3DES (Triple DES), 6, 30–31, 102, 393–394
  EDE mode, 30
  EEE mode, 30
  export, 50
  performance of, 30, 183
  required by S/MIMEv3, 404
  strength of, 30, 164
  and TLS, 51
40-bit encryption, export, 24
512-bit key exchange, export, 24
56-bit encryption, export, 25
90/10 Rule, 176–177

`/dev/random`, 288

Abstract Syntax Notation 1, *see* ASN.1
acceleration
  cryptographic, 152
  hardware, 204–209
  Java native code, 200–202
  of RSA, 205
  transparent, 206
accept
  OpenSSL server, 264–265
  PureTLS server, 264
`accept()`
  function, 263
  method, 264
access
  client mail, 357
  control with certificate, 236–237
Access Control List, *see* ACL
ACK
  piggybacking, 212
  TCP delayed, 212
ACL (Access Control List), 236
  use with username/password, 221
active
  attack, 2, 90, 114, 137, 141, 163–164, 221, 223, 229–232, 238–242, 247, 308–312, 372–373, 388, 461
  dictionary attack, 241
addresses, e-mail, 351–352
Adelman, L., 33
Advanced Encryption Standard, *see* AES
AES (Advanced Encryption Standard), 31
AH (Authentication Header), 380, 384
  MAC, 384
alert
  bad_certificate, 131–132, 311–312
  bad_record_mac, 130
  certificate_unknown, 131–132
  close_notify, 65, 93–94, 128, 244–247, 269, 282–285, 289, 307, 314–315, 363, 425
  decode_error, 133
  decompression_failure, 131
  decrypt_error, 130, 133
  decryption_failed, 130
  error, 128–134
  fatal, 92, 128–130, 132–134
  handshake_failure, 93, 112, 131, 133–134
  illegal_parameter, 132
  insufficient_security, 133–134
  internal_error, 134
  level, 128

Ex.1024
Askeladden, L.L.C. / Page 545 of 788

no_certificate, 131, 133
no_renegotiation, 100, 134
protocol_version, 133–134
table, 129
unexpected_message, 130
unknown_ca, 132
unsupported_certificate, 132
user_cancelled, 134
warning, 92–93, 128–129, 131–132, 134
alert
content type, 62, 231
trace, 93
algorithm
choice, bulk data, 183
choice, S/MIME, 403–405
digest, 32
encryption, 26–32
strength, 162
aliases, IP, 320
Alleged RC4, 27
Amdahl's Law, 176
American National Standards Institute, *see* ANSI
anchor
HTML, 297, 392
method type, 298
anonymous DH, 220, 238–242
ANSI (American National Standards Institute), 44
Apache, 54, 118, 295, 305, 319, 436–453
use with OpenSSL, 54
ApacheSSL, 54, 340
API
OpenSSL, 249
PureTLS, 249
SSL compared with sockets, 46
Apple patents, 103
Application Programming Interface, *see* API
`application/pkcs7-mime` content type, 401
application_data content type, 62
application-layer
protocol selection, 224–227
protocols, certificate chain and, 141–142
arc OID, 15
Arcfour, *see* Alleged RC4
ASN.1 (Abstract Syntax Notation 1), 14–15, 68
parsing, cost of, 186
asymmetric
algorithms, performance of, 184
cryptography, 9
endpoint authentication, 4
attack
active, 2, 90, 114, 137, 141, 163–164, 221, 223, 229–232, 238–242, 247, 308–312, 372–373, 388, 461
active dictionary, 241
brute force, 39
ciphertext only, 6
confidentiality, 144, 231
denial-of-service, 2, 230
DH man-in-the-middle, 223, 238–242
DH small-subgroup, 170–171
downgrade, 137, 171–173, 230–232, 308, 331, 379, 388
exhaustive search, 6
HTTPS and substitution, 327–328
integrity, 144, 231, 458, 464
known plaintext, 6
man-in-the-middle, 9, 141, 147–148, 223
million-message, 166, 168–170
passive, 2, 140, 165
on password, man-in-the-middle, 235
password guessing, 235
reordering, 90
replay, 59, 73, 90
SSLv2 client authentication transfer, 459–461
SSLv2 downgrade, 459
SSLv2 truncation, 459
`STARTTLS` downgrade, 369
`STARTTLS` useful against passive, 372
timing cryptanalysis, 167–168
truncation, 22, 93, 244, 315–316
and upward negotiation, downgrade, 229
Attribute-Value Assertion, *see* AVA
authenticated relaying, `STARTTLS`, 366–367
authentication
biometric, 153
client, 227, 242–243
e-mail client, 357
e-mail originator, 357
endpoint, 3
HTTP/SSL client, 303
HTTPS client, 322–327
HTTPS server, 308–309
mutual, 19, 238–242
one-way, 19
recipient, 4
S/MIME sender, 405
S-HTTP client, 395–396
sender, 4
and separate port, client, 242
SSL client, 108–112, 222, 236–238
`STARTTLS` lack of sender, 373
`STARTTLS` originator, 367
username/password, 235–236
Authentication Header, *see* AH
AVA (Attribute-Value Assertion), 12, 258

backward-compatible
ClientHello, 135
handshake, SSLv2, 135–137
PCT, 47, 49
SSL, 49, 73
SSLv3, 49
TLS, 50, 73
bad_certificate alert, 131–132, 311–312
bad_record_mac alert, 130
Baltimore, 54

bang path, 351
Basic Encoding Rules, *see* BER
basicConstraints
  certificate extension, 159, 161
  not checked by PureTLS, 260
Benaloh, J., 47
BER (Basic Encoding Rules), 14–15
`berr_exit()` function, 268
big-endian byte order, 34
Biham, E., 30
`BIO` object, 261, 264, 374
biometric authentication, 153
Birthday Paradox, collisions and, 32
Blaze, M., 40
Bleichenbacher, D., 168
block
  cipher modes, 28–29
  cipher padding check, 130
  ciphers, 27–32, 91
blocking
  I/O, 269
  and threads, 271
bottleneck, defined, 177
buffer size, OpenSSL, 214, 266
buffering
  and latency, 186
  latency, SSL handshake, 214–216
  network, 373
  performance cost, SSL handshake, 217
  SSL, 185, 277
`BUFSIZZ` macro, 266, 279
bulk data
  algorithm choice, 183
  transfer, performance of, 183
Bureau of Export Administration, *see* BXA
BXA (Bureau of Export Administration), 24
byte order, network, 114

C
  examples, 249–289, 331–341, 411–424
  HTTPS examples, 431–435
  `switch` statement, 70
C2 Net, 319
CA (Certificate Authority), 9–14, 52, 110, 141–142, 158–162
  initializing client, 253
  root, 158, 187, 253, 458
  SGC and, 100
cache
  certificate verification, 187
  proxy, 301–302
  SSL session, 181, 339–341, 397
`cancel.interrupt()` method, 285
CBC
  mode, 28
  rollover, 29, 165
  SSL encryption, 91
CDMF (Commercial Data Masking Facility), 25
Certicom, 53–54
  patents, 103
certificate, 9–13, 58, 60, 95–97, 108–112, 131–133, 222–223, 303, 456–457
  access control with, 236–237
  chain, 458
  chain and application-layer protocols, 141–142
  chain verification, 141, 156–162, 260, 262–263
  cost of, 52
  defined, 10
  DH, 102, 126
  discovery, 18, 407
  domain names and, 233
  extensions, 13
  extensions, Netscape, 161–162
  fingerprint, 158
  issuance, 236
  KEA, 105
  matching, IE, 321
  matching, Netscape, 321
  mismatch, HTTPS, 309–312
  revocation, 13–14, 238
  S/MIME, 161, 406–407
  SGC, 100–101, 119–124
  signature, 10
  Step-Up, 100–101, 115–119
  used with password, 237
  verification cache, 187
  X.509, 10–14
Certificate Authority, *see* CA
Certificate Revocation List, *see* CRL
Certificate
  message, 60, 64, 77–78, 94, 97, 108, 110–112, 115, 126, 185–186, 188, 194–195, 211, 214–216, 310
  message, cost of, 186–188, 193
  trace, 77
certificate_authorities value, CertificateRequest message, 110
certificate_types value, CertificateRequest message, 110
certificate_unknown alert, 131–132
CertificateRequest
  message, 97, 105, 108, 110, 126, 326
  message, certificate_authorities value, 110
  message, certificate_types value, 110
  trace, 109
CertificateVerify
  computation, SSLv3, 111
  computation, TLS, 112
  message, 97, 105, 108, 110–111, 114, 126, 163, 194–195
  message, cost of, 188
  trace, 111
CESG (Communications-Electronics Security Group), 9

CFB mode, 28
CHALLENGE value
  CLIENT-HELLO message, 136
  SSLv2, 136
change_cipher_spec content type, 62
ChangeCipherSpec
  message, 60, 64–65, 80, 106, 167, 185, 211
  Netscape use of, 80
check_cert_chain() function, 262–263
cipher
  suite recommendations, 142
  suite setting, 253
  suite table, 74
  suite table, SSLv2-compatible, 137
  suite translation, HTTPS, 319
  suites, 1024-bit export, 73
  suites, elliptic curve, 103
  suites, FORTEZZA, 104
  suites, Kerberos, 104
Cipher Block Chaining, *see* CBC
Cipher Block Feedback, *see* CFB
cipher_suite value, ServerHello message, 75
cipher_suites value, ClientHello message, 73
CIPHER-SPECS value, CLIENT-HELLO message, 136
CIPHER-SPECS-DATA value, CLIENT-HELLO message, 135
ciphers
  block, 27–32
  stream, 26–27
CipherSuite type, 73, 136
ciphertext
  defined, 6
  only attack, 6
Clark, D., 378
client
  authentication, 227, 242–243
  authentication, e-mail, 357
  authentication, HTTP/SSL, 303
  authentication, HTTPS, 322–327
  authentication, S-HTTP, 395–396
  authentication, SSL, 45, 96–97, 108–112, 222, 236–238
  authentication options, 221
  authentication and separate port, 242
  authentication time allocation, DSS/DH, 194–195
  authentication time allocation, RSA, 188–189
  authentication timeline, 98
  authentication transfer attack, SSLv2, 459–461
  authentication username/password, 221
  connect, 258–263
  identity, 158
  identity and, HTTPS, 314
  initialization, SSL, 253
  mail access, 346, 357
  private key, 96–97
  public key, 125–126
client.c file, 260, 263, 412, 420–421
client.h file, 420
Client.java file, 259
Client class, 425
client_random, 86–87, 143
client_version value, ClientHello message, 73
client_write_iv, 127
client_write_key, 164–165
CLIENT-CERTIFICATE message, 457, 460–461
CLIENT-FINISH message, 457
CLIENT-HELLO message, 135–138, 456, 459, 462
  CHALLENGE value, 136
  CIPHER-SPECS value, 136
  CIPHER-SPECS-DATA value, 135
CLIENT-MASTER-KEY message, 456–457, 459, 461
ClientHello
  backward-compatible, 135
  message, 60, 63, 67, 73–74, 99, 118, 124, 134–136, 138, 167, 171–172, 211, 225–226, 230–231, 316, 321, 326, 329, 362
  message, cipher_suites value, 73
  message, client_version value, 73
  message, ClientRandom value, 136
  message, compression_methods value, 73
  message, cost of, 186
  message, session_id value, 74–75, 96, 106
  message and rehandshake, 100
  message and session resumption, 96, 106
  message and SGC, 115
  message and SGC/Step-Up, 100
  random value, 73
  trace, 73
ClientKeyExchange
  message, 60, 64, 70, 78–79, 102, 126–127, 133, 167–168, 171, 173, 185, 193
  message, cost of, 186–188, 193–194
  message, DH, 125–126
  message, FortezzaKeys value, 127
  message, Kerberos, 104
  trace, 78
ClientRandom value, ClientHello message, 136
close() function, 245–246, 283
close_notify
  alert, 65, 93–94, 128, 244–247, 269, 282–285, 289, 307, 314–315, 363, 425
  RST during, 283, 285
  unnecessary with STARTTLS, 363–364
closure
  errors, 244–247
  HTTPS, 314–316
  incomplete, 244–246
  OpenSSL, 269, 283
  premature, 246–247
  PureTLS, 283–285
  SSL, 93, 244–247, 282–285
  STARTTLS connection, 363–364

Ex.1024
Askeladden, L.L.C. / Page 548 of 788

CMS (Cryptographic Messaging Syntax), 390–391
  detached signature, 390
  use in S-HTTP, 389
CN (Common Name), 13, 318
  extraction, OpenSSL, 263
  extraction, PureTLS, 258
  use of domain name, 156
CNAME record, 234, 320
collision-resistance, message digest, 7
collisions and Birthday Paradox, 32
COM.claymoresystems.ptls package, 254
COM.claymoresystems.sslg package, 254
Commercial Data Masking Facility, *see* CDMF
Common Name, *see* CN
common.c file, 255–256, 283, 411, 413–414
common.h file, 255, 257, 412–413
Communications-Electronics Security Group, *see* CESG
compatible cofactor exponentiation, 171
compression
  method, NULL, 73, 76
  record, 88
compression_method value, ServerHello message, 75
compression_methods value, ClientHello message, 73
compromise
  of digest algorithms, 165–166
  of digital signature algorithms, 163
  of encryption algorithms, 164–166
  of key establishment algorithm, 163–164
  of master_secret, 142–145, 163
  of private key, 140, 146–149, 163
  SSL partial, 162–166
confidentiality
  advised, 221
  attack, 144, 231
  defined, 3
  requirement for e-mail sender, 343
  SSL design goal, 45
congestion control and, SSL, 212
connect, client, 258–263
connect() function, 260
CONNECT
  and firewall, HTTP, 371–372
  method, HTTPS, 316
  method implementation, 331–336
  STARTTLS and, 371–372
connection
  behavior, HTTP, 293
  closure, STARTTLS, 363–364
  hijacking TCP, 144
  HTTP, 300
  security properties, 223
  semantics, HTTP/SSL, 303–304
  semantics optimizing, 228
  vs. session, SSL, 96
Connection header, 294
CONNECTION-ID, 456–457
content type
  alert, 62, 231
  application/pkcs7-mime, 401
  application_data, 62
  change_cipher_spec, 62
  handshake, 62
  multipart/signed, 402
  SSL record, 61
Content-Length header, 295, 300, 315
Content-Type header, 295, 401
context
  initialization, SSL, 252–258
  object, SSL, 252
core dump, 146
cost of
  ASN.1 parsing, 186
  Certificate message, 186–188, 193
  CertificateVerify message, 188
  ClientHello message, 186
  ClientKeyExchange message, 186–188, 193–194
  cryptography, 179–180
  ephemeral key generation, 195–197
  I/O, 177
  prime generation, 192
  record encryption, 197–199
  record MAC, 197–199
  ServerKeyExchange message, 190–193, 195
  session resumption, 181–182
countermeasures
  downgrade attack, 232
  million-message attack, 170
  small-subgroup attack, 171
  timing cryptanalysis, 168
CRC in SSLv2, 464
CreateThread() method, 337–338
CRL (Certificate Revocation List), 13, 164, 238
cRLSign keyUsage value, 160
Cryptographic Messaging Syntax, *see* CMS
cryptography
  cost of, 179–180
  defined, 5
cryptology, defined, 5
Cyclic Redundancy Check, *see* CRC

Dai, W., 197
Danvers Doctrine, 50
data records, TSP, 20–24
Data Encryption Standard, *see* DES
Data CMS content type, 390
DATA command, SMTP, 349
dataEncipherment keyUsage value, 160
DBM, 340
deadlock, 269
decipherOnly keyUsage value, 160
decode_error alert, 133
decompression_failure alert, 131

decrypt_error alert, 130, 133
decryption_failed alert, 130
Deep Crack, 30
delayed ACK, TCP, 212
Demo.java file, 254, 428
Demo class, 254–255, 425
denial-of-service attack, 2, 230
DER (Distinguished Encoding Rules), 14–15
DES (Data Encryption Standard), 6, 29–30, 102, 393
  strength of, 29–30
DES-MAC, 38
design goals, SSL, 44–45
designing with SSL, 219–247
destroySession() method, 340
DESX, 393
detached signature, CMS, 390
DH
  anonymous, 220, 238–242
  certificate, 102, 126
  ClientKeyExchange message, 125–126
  ClientKeyExchange trace, 126
  ephemeral, 102, 125–126
  group, setting, 253
  handshake, parallelism in, 193
  IKE use of, 381
  key establishment, 34–36
  key generation on the fly, 192
  man-in-the-middle attack, 223, 238–242
  modulus, 35, 125
  parameters, 126
  patent, 102
  performance improvements with, 195–197
  performance of, 37, 191–197
  public key, 381
  reuse of ephemeral key, 196–197
  ServerKeyExchange trace, 125
  small-subgroup attack, 170–171
  and SSL, 49, 102, 124–126
  static, 126, 146
  strength of, 35, 40
  and TLS, 51
  use of smaller keys, 195–196
DHCP (Dynamic Host Configuration Protocol), 108
dictionary attack, active, 241
differential cryptanalysis, 30
Diffie, W., 9, 40
Diffie-Hellman, *see* DH
digest, 15
  algorithm, 32, 156
  algorithms, 155
  algorithms, compromise of, 165–166
  algorithms, performance of, 184
DigestedData CMS content type, 390
DigestInfo structure, 36, 112
Digital Signature Algorithm, *see* DSS
digital signature, 9, 15, 36–38
  RSA, 36–37
  algorithms, 163
  algorithms, compromise of, 163
  DSS, 37
  performance, 182
  see also DSS; RSA
Digital Signature Standard, *see* DSS
digitally-signed operator, 111
digitalSignature keyUsage value, 160–161
discovery, certificate, 407
Distinguished Encoding Rules, *see* DER
Distinguished Name, *see* DN
DN (Distinguished Name), 12
  wildcarding, 308
DNS
  domain name, 156
  spoofing, 369
DNS Security, *see* DNSSEC
dNSName certificate extension, 13, 157, 308
DNSSEC (DNS Security), 409
dnToCommonName() method, 258
domain
  name, Common Name use of, 156
  name, DNS, 156
  name in RCPT command, 355
  names and certificate, 233
Domain Name System, *see* DNS
downgrade attack, 137, 171–173, 230–232, 308, 331, 379, 388
  countermeasures, 232
  SSLv2, 459
  STARTTLS, 369
  trace, 172
  and upward negotiation, 229
DSS (Digital Signature Standard), 37
  patent, 37, 102
  performance of, 37, 182
  RSA vs., 182
  signature encoding, 125
  and SSL, 49, 102, 124–126
  strength of, 40
  and TLS, 51
  use with SHA-1, 37
DSS/DH
  client authentication time allocation, 194–195
  expanded buffer timeline, 215
  normal buffer timeline, 214
  time allocation, 191–194
  timeline, 103
  with RC4, 73
dss_fixed_dh certificate type, 110, 126
dss_sign certificate type, 110
dual homing, 210
Dynamic Host Configuration Protocol, *see* DHCP

e-mail, 234, 343–376, 400–408
  addresses, 351–352
  client authentication, 357

and firewall, 352
message identity, 351
originator authentication, 357
reference integrity, 357–358
relaying, 344, 352–355, 357
security indicators, 365–366
sender identity, 351
transport identity, 351
virtual host, 355
EC cipher suites, 103
ECB mode, 28
echo server, 251
`echo()` function, 266–267
`echo.c` file, 267, 412, 415–416
`echo.h` file, 415
EDE mode, 3DES, 30
EEE mode, 3DES, 30
EHLO command, SMTP, 349
Electronic Code Book, *see* ECB
Elgamal, T., 455
Elliptic Curve, *see* EC
emailAddress certificate extension, 13, 405
Encapsulating Security Payload, *see* ESP
encipherOnly keyUsage value, 160
Encrypt-Decrypt-Encrypt, *see* EDE
Encrypt-Encrypt-Encrypt, *see* EEE
encrypted master secret, 18
EncryptedData CMS content type, 390
EncryptedPreMasterSecret, 79
encryption, 6, 15
algorithm, 26–32
algorithms, 155
algorithms, compromise of, 164–166
export 40-bit, 24
export 56-bit, 25
non-secret, 9
password-based, 149–150
of private key, 149
see also DES; RC2; RC4; 3DES
end-of-data markers, 246
end-to-end
argument, 378–379
security, 224
endpoint authentication, 3
asymmetric, 4
entropy of text, 149
enum
maximum size, 69
select and, 70
type, 69–70
EnvelopedData CMS content type, 390–391
ephemeral
DH, 102, 125–126
key, DH reuse of, 196–197
key generation, cost of, 195–197
keying, 167
keying performance cost, 216
RSA, 97–99, 112–115, 183, 253
RSA time allocation, 189–191
RSA timeline, 99
`errno` variable, 268
error
alert, 128–134
alerts, SSL, 93
would block, 277
errors, closure, 244–247
ESP (Encapsulating Security Payload), 380, 384–385
Ethernet maximum segment size, 215
examples
C, 249–289, 331–341
Java, 249–289
Exclusive Or, *see* XOR
exhaustive search attack, 6
export, 24–25, 100–101
40-bit encryption, 24
56-bit encryption, 25
1024-bit key exchange, 25
3DES, 50
512-bit key exchange, 24
ciphers, performance of, 198
key derivation, 86–87
RC2, 25
RC4, 25
SSL, 25
EXPORT1024 cipher suite, 119
exportability, SSLv2, 49
extendedKeyUsage certificate extension, 13, 115
extensions
certificate, 13
key usage, 160–161
Netscape certificate, 161–162
SMTP, 357–358

F5, 206
fatal alert, 92, 128–130, 132–134
file locking, 340
File Transfer Protocol, *see* FTP
FIN, TCP, 65, 93, 244–246, 283, 285, 307, 459
final_client_write_iv computation, 88
final_client_write_key computation, 87–88
final_server_write_iv computation, 88
final_server_write_key computation, 87–88
`findSession()` method, 340
fingerprint, 153
certificate, 158
Finished, 94
computation, SSLv3, 81
computation, TLS, 82
message, 60, 64–65, 80–82, 108, 130, 133, 147, 167, 170–171, 211–213, 230, 240–242, 459, 461
message, protection against tampering, 81
message, Sender value in, 81
message, sha_hash value, 125

processing, 82–83
trace, 81
when to transmit, 65
FIPS-186, 37
firewall, 383
e-mail and, 352
HTTP CONNECT and, 371–372
proxy, 230
and separate port, 229
and SSL, 224
and upward negotiation, 230
flock() function, 340
fork() function, 265, 337–338, 341
forms, HTML, 298
FORTEZZA
cipher suite table, 128
cipher suite translation, 319
cipher suites, 104
implementation, Netscape, 106
key escrow, 105
LEAF, 126
SSL, 49, 104–106, 126–128
timeline, 105
FortezzaKeys value, ClientKeyExchange message, 127
Fortify, 100, 319
fragments, 88
definition of, 61
Freier, A., 49
From header, 350
FTP (File Transfer Protocol), 244
over SSL, 52, 244

GET method, 294, 298, 327
gethostbyname() function, 260
getpeername() function, 286
Goldberg, I., 79
GoNative Provider, 200–202
gprof program, 179
Group of Seven, 40

half-close, TCP, 245
handshake
buffering latency, SSL, 214–216
buffering performance cost, SSL, 217
message, SSL, 59–60, 70
over high-latency network timeline, SSL, 211
overhead reducing, 227
parallelism, SSL, 216
serialization, 185–186
SSL, 58–60, 95–101, 171–173, 258–265
SSLv2, 455–457
time allocation, SSL, 184–197
TSP, 18–19
Handshake message, msg_type value, 70
handshake content type, 62
handshake_failure alert, 93, 112, 131, 133–134
hardware
acceleration, 204–209
key storage, 152
RNG, 155–156
hash algorithms, *see* message digest also
HEAD method, 294
Hellman, M., 9
HelloRequest message, 99–100, 134, 243, 322, 326
HELO command, SMTP, 349
Hickman, K., 47, 455
high-latency network timeline, SSL handshake over, 211
hijacking TCP connection, 144
HMAC, 7, 81–82, 85, 89, 180, 457
construction of, 38
immune to known attacks on MD5, 165
keyed digest, 38
overhead of, 198
pad1, 81
pad2, 81
in TLS PRF, 85
XOR in, 39
hop-by-hop
headers, HTTP, 330
security, STARTTLS, 371
host
e-mail virtual, 355
HTTP virtual, 293, 302
smart, 354–355
SMTP virtual, 344
STARTTLS and virtual, 364–365
Host header, 302, 320, 328
HTML (Hypertext Markup Language), 292, 295–298
anchor, 297, 392
dynamic content in, 298
forms, 298
links, 297
tags, 295
HTTP (Hypertext Transfer Protocol), 43, 251, 291–295, 389–400
CONNECT and firewall, 371–372
connection, 300
connection behavior, 293
hop-by-hop headers, 330
over SSL, 44, 291–341
performance of, 300
proxy, 293, 301–302
Request-URI, 294, 302
response, 296
security considerations, 293
Upgrade, 328–331
upgrade and reference integrity, 330–331
Upgrade header, 328–331
upward negotiation, 328–331
virtual host, 293, 302
HTTP/SSL
client authentication, 303

connection semantics, 303–304
programming issues, 331
reference integrity, 303–304
HTTPS (traditional HTTP over SSL), 51, 304–328
certificate checking algorithm, 309
certificate mismatch, 309–312
cipher suite translation, 319
client authentication, 322–327
client identity and, 314
closure, 314–316
CONNECT method, 316
CONNECT trace, 317
error handling, 310
and man-in-the-middle proxy, 316–319
man-in-the-middle-attack, 308
and multiple clients, 336–341
and multiple names, 321
port, 305
problems with proxy, 378
and proxy, 316–319
reference sources, 312
and Referrer header, 327
S-HTTP vs., 397–400
and separate port, 305
server authentication, 308–309
and substitution attack, 327–328
trace, 305–307
and virtual hosts, 320–321
https: URL scheme, 223, 304
Hypertext Markup Language, *see* HTML
Hypertext Transfer Protocol, *see* HTTP

I/O
blocking, 269
cost of, 177
handling, OpenSSL, 265–269, 274–282
handling, PureTLS, 269–274
multiplexed with select() function, 274–282
multiplexed with threads, 269–274
nonblocking, 276
IANA (Internet Assigned Numbers Authority), 225
IBM, 25, 29
IDEA, 102, 393
identity
checking, 157
client, 158
e-mail message, 351
e-mail sender, 351
e-mail transport, 351
protection, IKE, 382–383
server, 156–157, 222–223
IE (Internet Explorer), xxi, 54, 119, 200, 310–312
accepting bad certificate timeline, 313
certificate matching, 321
Java performance on, 200
UI for SSL, 52
IESG and TLS working group, 50
IETF (Internet Engineering Task Force), 43
obtaining documents, 43
and separate port, 229
Standards Track, 43
TLS and, 44, 49, 51
IETF Public Key Infrastructure WG, *see* PKIX
IIS (Internet Information Server), 54, 119
IKE (Internet Key Exchange), 380–384
exchanges, 382
identity protection, 382–383
initiator, 382
proposal, 382
responder, 382
use of DH, 381
illegal_parameter alert, 132
IMAP (Internet Mail Access Protocol), 251, 346
IMG tag, 297
implementation
barriers, S/MIME, 406–407
CONNECT method, 331–336
of PRNG, 154–155
incomplete
close, 245
closure, 244–246
inetd, 374
initialization
OpenSSL, 255–257
PureTLS, 254–255
SSL client, 253
SSL context, 252–258
SSL server, 253, 257–258
Initialization Vector, *see* IV
initialize_ctx() function, 255–256, 261–262
initiator, IKE, 382
inline images, 297–298
Input/Output, *see* I/O
InputStream
class, 425
object, 272–273
insufficient_security alert, 133–134
INTEGER type, 14
integrity
attack, 144, 231, 458, 464
reference, 227, 233–235
S-HTTP reference, 396
SSL reference, 222–223
Intel, 206–207
internal_error alert, 134
InternalError error, 274
International Standards Organization, *see* ISO
International Telecommunications Union, *see* ITU
Internet
mail security, 343–345
threat model, 1–2
Internet Assigned Numbers Authority, *see* IANA
Internet Engineering Steering Group, *see* IESG
Internet Engineering Task Force, *see* IETF

Ex.1024
Askeladden, L.L.C. / Page 553 of 788

Internet Explorer, *see* IE
Internet Information Server, *see* IIS
Internet Key Exchange, *see* IKE
Internet Mail Access Protocol, *see* IMAP
Internet Protocol, *see* IP
Internet Protocol Security, *see* IPsec
Internet Security Association and Key Management Protocol, *see* ISAKMP
Internet Service Provider, *see* ISP
interprocess communication, 340
`interrupted()` method, 285
`IOException` exception, 274, 285
IP (Internet Protocol), 46, 320
iPivot, 206–207
IPsec (Internet Protocol Security), 379–388
  AH, 380
  ESP, 380
  IKE, 380
  ISAKMP, 380
  and NAT, 388
  policy, 386
  and proxy, 387
  transport mode, 385
  tunnel mode, 385
  vs. SSL, 386–388
  Windows 2000 support for, 388
ISAKMP (Internet Security Association and Key Management Protocol), 380–384
  transport, 383–384
ISO (International Standards Organization), 15
ISP (Internet Service Provider), 302
ISPs virtual host and, 344
issuer name, 10
ITU (International Telecommunications Union), 44
IV (Initialization Vector), 28
  export, 87–88
iv_block, 86
  computation for TLS, 87

JCA (Java Cryptography Architecture) 200
Java
  examples, 249–289, 425–430
  native code acceleration, 200–202
  Native Interface, 200
  performance, 199–202
  in Web pages, 298
Java Cryptographic Architecture, *see* JCA
Java Development Kit, *see* JDK
Java Secure Sockets Extension, *see* JSSE
`java.net.Socket` class, 258
`java.net` package, 264
JavaScript, 298
JCA (Java Cryptographic Architecture), 200
JDK (Java Development Kit), 200, 250
JNI, performance of, 200–202
`join()` method, 272
JSSE (Java Secure Sockets Extension), 258

Karlton, P., 49
KDC (Key Distribution Center), 8
  drawbacks of, 8
KDF
  message digest, 20
  TSP, 20
KEA (Key Exchange Algorithm), 105
  certificate, 105
`Keep-Alive` header, 295
Kerberos, 8, 104
  cipher suites, 104
  ClientKeyExchange message, 104
  ticket, 104
key
  compromise of private, 163
  defined, 6
  derivation, SSL, 82–88, 143
  derivation, TSP, 20
  DH reuse of ephemeral, 196–197
  escrow FORTEZZA, 105
  establishment, 32–36
  establishment, DH, 34–36
  establishment algorithm, compromise of, 163–164
  exchange, 32
  exchange, export 1024-bit, 25
  exchange, export 512-bit, 24
  exchange, RSA, 33–34, 79
  generation on the fly, DH, 192
  length, 39–40
  loading at startup, 375
  MAC, 58, 73
  mailing list, 406
  management, 8
  private, 9–12, 16–20, 32–38
  public, 9–12, 16–20, 32–38, 102
  recovery, storageless, 150–151
  reuse, 20
  server public, 58, 64
  size and performance, 182–183
  SSL MAC, 144
  storage, hardware, 152
  storage of, private, 149
  transport, 32
  usage extensions, 160–161
  see also DH; RSA
Key Derivation Function, *see* KDF
Key Distribution Center, *see* KDC
Key Exchange Algorithm, *see* KEA
key_block computation, TLS PRF, 86
`Key-Assign` header, S-HTTP, 396–397
keyAgreement keyUsage value, 160
keyCertSign keyUsage value, 160
keyed digest, HMAC, 38
keyEncipherment keyUsage value, 160–161
keying material, loading, 253
keystream, defined, 26
keyUsage certificate extension, 13, 160–161

Klein, D., 327
known plaintext attack, 6
Kocher, P., 49, 156, 167, 455

Lampson, B., 47
last-hop delivery, SMTP/TLS and, 378–379
latency
  buffering and, 186
  SSL handshake buffering, 214–216
  SSL and network, 209–212
  vs. throughput, 178
LDAP (Lightweight Directory Access Protocol), 299
ldap: URL scheme, 299
LEAF (Law Enforcement Access Field), FORTEZZA, 126
Lightweight Directory Access Protocol, *see* LDAP
link-level security, SMTP/TLS and, 379
load
  generator, 202
  SSL servers under, 202–204
loading keying material, 253
locking memory, 145–146
LUCIFER, 40

MAC (Message Authentication Code), 7, 20–23, 38–39, 59, 238–242, 391
  AH, 384
  of handshake messages, 58
  key, 58, 73
  key, SSL, 144
  of type field, 22
  PCT, 461
  SSL, 108, 130, 180, 274
  SSLv2, 457, 459
  TLS, 89–90
  TSP, 20–21
MAC-Info header, 391
mail
  access, client, 346, 357
  access, remote, 345
  security, Internet, 343–345
  server, 343
  servers on, UNIX, 374
Mail API, *see* MAPI
Mail Transport Agent, *see* MTA
Mail User Agent, *see* MUA
MAIL command, SMTP, 349, 351
mailing list key, 406
main()
  function, 255, 261–262
  method, 271
man-in-the-middle
  attack, 9, 141, 147–148, 223
  attack, DH, 223, 238–242
  attack on password, 235
  proxy, HTTPS and, 316–319
  timeline, 239
man-in-the-middle-attack, HTTPS, 308
MAPI (Mail API), 345
markers, end-of-data, 246
MARS, 31
masks, select(), 282
master secret, TSP, 18
master_secret, 58, 83–84, 96, 106, 108, 112, 119, 140, 142–147, 165–166, 171, 173, 181, 239
  compromise of, 142–145, 163
  computation, PRF TLS, 85
  protecting, 140
master_secret_iv, 127
maximum segment size, Ethernet, 215
md5_hash
  computation, 114
  SSLv3 Finished, 81
MD2 (Message Digest 2), 32
MD4 (Message Digest 4), 32
MD5 (Message Digest 5), 7, 32, 81–88, 111–112, 165–166
  deprecated, 142
  performance of, 183, 198
memory
  locking, 145–146
  scrubbing, 145
  zeroing, 145
Merkle-Hellman patent, 50
message
  authentication requirement for e-mail sender, 343
  digest, 6–7
  digest, defined, 6
  digest, key derivation function, 20
  digest collision-resistance, 7
  format, S-HTTP, 389, 391
  identity, e-mail, 351
  integrity, defined, 3
  integrity required by SSL, 221
  integrity requirement for e-mail sender, 343
Message Authentication Code, *see* MAC
Message Digest 2, *see* MD2
Message Digest 4, *see* MD4
Message Digest 5, *see* MD5
messaging
  secure, 16–17
  store-and-forward, 346
method type, anchor, 298
Microsoft, 47, 49, 54, 119, 455
  Exchange, 345
  and SGC, 100
million-message attack, 166, 168–170
  countermeasures, 170
MIME (Multipurpose Internet Mail Extensions), 344, 349–351
MIME Object Security Services, *see* MOSS
Mockapetris, P., 156

mod_ssl, xxiii, 340
  session cache, 436–453
  use of Step-Up, 118
modulus
  DH, 35, 125
  RSA, 40, 114
Moeller, B., 171, 240
Moore, G., 39
Moore's law, 39
MOSS (MIME Object Security Services), 391
mserver.c file, 338, 434–435
mserver program, 431
msg_type value, Handshake message, 70
MTA (Mail Transport Agent), 345–346
MUA (Mail User Agent), 345–346
multipart/signed content type, 402
multiple
  client handling, 288
  clients, HTTPS and, 336–341
  names, HTTPS and, 321
  recipients and, S/MIME, 406
  signers and, S/MIME, 406
multiprocess servers, 337–339
Multipurpose Internet Mail Extensions, *see* MIME
mutual authentication, 19, 238–242, 387
MX record, 234, 345, 356

Nagle Algorithm, performance and, 212–213
name
  DNS domain, 156
  issuer, 10
  resolution, weaknesses of, 157
  subject, 10
NAT (Network Address Translation), 388
National Bureau of Standards, *see* NBS
National Institute of Standards and Technology, *see* NIST
National Security Agency, *see* NSA
native code acceleration, Java, 200–202
NBS (National Bureau of Standards), 29
nCipher, 204
Needham, R.M., 8
negotiation
  S-HTTP, 389, 392–393
  SSL, 59, 72–76
  TSP lack of, 24
  upward, 225–227
Netscape, 49, 53, 119, 125, 319
  accepting bad certificate timeline, 311
  certificate extensions, 161–162
  certificate matching, 321
  Commerce Server, 52
  de facto standard for SSL, 50
  design of SSL, 44, 455
  FORTEZZA implementation, 106
  Navigator, xxi, 200
  Navigator, Java performance on, 200
  Navigator 1.1, 47
  Navigator 2, 304
  PRNG, 164
  rejecting bad certificate timeline, 312
  SSLREF, 137
  UI for SSL, 52
  use of ChangeCipherSpec, 80
  use of Step-Up, 100, 118
  weaknesses of, 47, 79
  wildcarding, 321
netscape-cert-type certificate extension, 161
netscape-ssl-server-name certificate extension, 161–162
Network Alchemy, 206, 208–209
network
  buffering, 373
  byte order, 114
  error handling, PureTLS, 274
  latency, SSL and, 209–212
Network Address Translation, *see* NAT
Network News Transfer Protocol, *see* NNTP
New Directions in Cryptography, 9
NIST (National Institute of Standards and Technology), 29, 102
NNTP
  over SSL, 52
  SSL over, 45
no_certificate alert, 131, 133
no_renegotiation alert, 100, 134
Nokia, 206, 208–209
Non-Secret Encryption, *see* NSE
nonblocking I/O, 276
nonrepudiation, 224
  defined, 9
  S/MIME, 408
  S-HTTP, 398
nonRepudiation keyUsage value, 160
NSA (National Security Agency), 24
NSE (Non-Secret Encryption), 9
null cipher, 91
NULL, 67
  compression method, 73, 76

OAEP (Optimal Asymmetric Encryption Padding), 170
OAKLEY, 381
object
  security protocols, 379
  signing, 161
Object Identifier, *see* OID
OCSP (Online Certificate Status Protocol), 13
OFB mode, 28
OID (Object Identifier), 15
  arc, 15
one-way authentication, 19
onEOD() method, 284–285
online certificate status, 13
Online Certificate Status Protocol, *see* OCSP

opaque type, 68
open SMTP relays, 354–355
Open Systems Interconnect, *see* OSI
OpenSSL, xix, xxi, 53–54, 249–289
  Apache use with, 54
  API, 249
  buffer size, 214, 266
  closure, 269, 283
  Common Name extraction, 263
  I/O handling, 265–269, 274–282
  initialization, 255–257
  password callback, 255
  read handling, 266–268
  server accept, 264–265
  support for SGC, 124
  symmetric, performance of, 197
  write handling, 268–269, 278–279
opportunistic cryptography, 231
Optimal Asymmetric Encryption Padding, *see* OAEP
`OPTIONS` method, 294, 329, 331
oracle, victim's server as, 169
organizational servers, 353–354
originator authentication
  e-mail, 357
  `STARTTLS`, 367
OSI (Open Systems Interconnect), 14
Output Feedback Mode, *see* OFB
`OutputStream`
  class, 425
  object, 272–273

pad1, HMAC, 81
pad2, HMAC, 81
pad_1, SSLv3 MAC, 89
pad_2, SSLv3 MAC, 89
padding
  check, block cipher, 130
  PKCS #1, 34
  SSL record, 91
parallel operation, 228
parallelism
  in DH handshake, 193
  SSL handshake, 216
parameters, DH, 126
parsing, cost of ASN.1, 186
partial compromise, SSL, 162–166
passive attack, 2, 140, 165
  `STARTTLS` useful against, 372
  on password, 221
passphrase, 150
  entry at reboot, 153
  iteration, 150
  salting, 150
password
  callback OpenSSL, 255
  certificate used with, 237
  guessing attack, 235
  man-in-the-middle attack on, 235
  one-time, 222
  passive attack on, 221
  transferable, 235
password-based
  encryption, 149–150
  mutual authentication timeline, 241
patent
  DH, 102
  DSS, 37, 102
  Merkle-Hellman, 50
  RSA, 34
patents
  Apple, 103
  Certicom, 103
PCI bus, 204
`pclient.c` file, 332, 335, 431, 434
`pclient` program, 431
PCMCIA card, 152
PCT (Private Communications Technology), 47–49, 461–462
  backward-compatible, 47, 49
  cipher suite table, 463
  MAC, 461
  version number hack and, 47
PCT_SSL_COMPAT cipher suite, 47
PEM (Privacy Enhanced Mail), 16
Pentium III RNG, 156
Perfect Forward Secrecy, *see* PFS
performance, 175–217
  of 3DES, 30, 183
  of asymmetric algorithms, 184
  of bulk data transfer, 183
  cost, ephemeral keying, 216
  cost, SSL handshake buffering, 217
  of DH, 37, 191–197
  of digest algorithms, 184
  digital signature, 182
  of DSS, 37, 182
  of export ciphers, 198
  of HTTP, 300
  improvements with, DH, 195–197
  Java, 199–202
  of JNI, 200–202
  key size and, 182–183
  of MD5, 183, 198
  and Nagle Algorithm, 212–213
  of OpenSSL symmetric, 197
  principles, 176–179
  public key, 103
  of public key algorithms, 37
  of RC4, 183
  of rehandshake, 326
  of RSA, 37, 182, 186
  rules, SSL, 184, 216–217
  of SHA-1, 183, 198
  of SSL record processing, 197–199
  of symmetric algorithms, 31, 184

PFS (Perfect Forward Secrecy), 35, 147, 195, 381
PFX, 150
PGP (Pretty Good Privacy), 16
Phaos Technology Corporation, 54
PIN (Personal Identification Number), use with hardware, 152
PKC (Public Key Cryptography), 9
PKCS
#1, 33
#1, weaknesses of, 169
#1 padding, 34
#5, 150–151
#7, 400
#12, 150
PKIX (IETF Public Key Infrastructure WG), 50, 159
plaintext, defined, 6
policy, IPsec, 386
POP (Post Office Protocol), 346
port
client authentication and separate, 242
HTTPS, 305
separate, 228–229
well-known, 52, 225
Post Office Protocol, *see* POP
POST method, 294, 298, 327
pre_master_secret, 58, 63, 73, 76, 79, 82–83, 94, 96, 98, 102, 104, 110, 126–127, 141–143, 146, 167–170, 173, 180, 188–189, 239
encryption of, 77
pre-forking, 341
preferSignedData S/MIME extension, 403
PreMasterSecret, 239–240
structure, 79
premature closure, 246–247
Pretty Good Privacy, *see* PGP
PRF (Pseudo-Random function), 82–87
key_block computation, TLS, 86
TLS master_secret computation, 85
prime
generation, cost of, 192
strong, 171
primitives, security, 15
Privacy Enhanced Mail, *see* PEM
private key, 9–12, 16–20, 32–38
client, 96–97
compromise of, 140, 146–149, 163
encryption of, 149
server, 58, 97–103, 119, 375
compromise, summary of effects of, 164
securing 146–154
size, 184
storage of, 149
Private Communications Technology, *see* PCT
PRNG (Pseudo-Random Number Generator), 140–141
implementation of, 154–155
Netscape, 164
requirements, 154
seeding, 155, 253
programming issues, HTTP/SSL, 331
programs, sample, 249–289, 331–341, 411–436
proposal, IKE, 382
protecting, master_secret, 140
protection, IKE identity, 382–383
protocol selection, 227
application-layer, 224–227
protocol_version alert, 133–134
protocols, object security, 379
proxy
cache, 301–302
firewall, 230
HTTP, 293, 301–302
HTTPS and, 316–319
HTTPS and man-in-the-middle, 316–319
HTTPS problems with, 378
IPsec and, 387
PureTLS and, 336
S-HTTP and, 398
SMTP, 352
SMTP/TLS problems with, 378
`proxy_connect()` function, 335–336
Pseudo-Random function, *see* PRF
Pseudo-Random Number Generator, *see* PRNG
public key, 9–12, 16–20, 32–38, 102, 127
client, 125–126
DH, 381
server, 58, 64, 77–79, 125, 239
algorithms, performance of, 37
cryptography, defined, 6
encryption, 15
performance, 103
Public Key Cryptography, *see* PKC
Public Key Cryptography Standards, *see* PKCS
public-key-encrypted operator, 79
PureTLS, xix, xxi, 54, 249–289
API, 249
basicConstraints not checked by, 260
closure, 283–285
Common Name extraction, 258
I/O handling, 269–274
initialization, 254–255
lack of direct network access, 374
network error handling, 274
and proxy, 336
read handling, 269–274
server accept, 264
session cache, 340
write handling, 269–274
PUT method, 294

qmail, xxi, 362
query, URL, 299
queue size, 202
QUIT command, SMTP, 349, 363

Ex.1024
Askeladden, L.L.C. / Page 558 of 788

r_c FortezzaKeys structure, 127
r_s ServerFortezzaParams structure, 127
Rainbow, 204–206
random
  number generation, 79, 154–156
  numbers, SSL use of, 140
Random Number Generator, *see* RNG
random value
  ClientHello, 73
  ServerHello message, 75
Raven, 54
RC2, 31, 102
  export, 25
  strength of, 31
  variable effective key length, 31
RC4, 6, 27, 102
  DSS/DH with, 73
  export, 25
  performance of, 183
  published, 27
  speed, 27
  strength of, 27
  weakness of 40-bit, 164
RC6, 31
`rclient.c` file, 287, 412, 423–424
`RClient.java` file, 285, 430
`RClient` class, 426
`RCPT` command
  domain name in, 355
  SMTP, 349
RDN (Relative Distinguished Name), 12, 258
read
  handling, OpenSSL, 266–268
  handling, PureTLS, 269–274
  and `select()` function, SSL, 274–276
`read()`
  function, 266, 268
  method, 273–275
`read_write()`
  function, 279–282
  method, 279
`read_write.c` file, 276, 278, 280, 282, 412, 416, 418
`read_write.h` file, 416
`readline()` function, 335
`ReadWrite.java` file, 272–273, 426–427
`ReadWrite.onEOD()` method, 284
`ReadWrite.run()` method, 284
`ReadWrite`
  class, 265, 272–273, 285, 425
  file, 284
`ReadWriteWithCancel.java` file, 284, 427–428
`ReadWriteWithCancel` class, 272, 284–285, 425
receipts, S/MIME, 406
`Received` headers, 350–351
recipient authentication, 4
RecipientInfo, 390, 406
record
  compression, 88
  encryption, cost of, 197–199
  length, TSP, 21
  MAC, cost of, 197–199
  MX, 356
  padding, SSL, 91
  processing, performance of SSL, 197–199
  protocol, SSL, 61–65, 88–91
  size, SSL optimal, 183–184, 198–199
  SSL, 61
record_overflow alert, 131
`recvClose()` method, 283
Reed, D., 378
reference
  integrity, 227, 233–235
  integrity, e-mail, 357–358
  integrity, HTTP upgrade and, 330–331
  integrity, HTTP/SSL, 303–304
  integrity, S-HTTP, 396
  integrity, SSL, 222–223
  integrity, `STARTTLS`, 367–371
  sources, HTTPS, 312
`Referrer` header, 327
  HTTPS and, 327
rehandshake
  performance of, 326
  security value, 165
  SGC, 100
  SSL, 99–100, 242–244
  Step-Up, 119
  with client authentication trace, 322–325
relative URL, 299
Relative Distinguished Name, *see* RDN
relay, S/MIME interaction with, 408
relaying
  e-mail, 344, 352–355, 357
  SMTP, 369–371
  SMTP/TLS and, 379
  `STARTTLS` authenticated, 366–367
  vs. security, `STARTTLS`, 369–371
reliable
  transport, SSL and, 47, 90
  transport and SSL, 409
remote mail access, 345
renegotiation failure, 134
reordering attack, 90
replay attack, 22, 59, 73, 90
Request For Comments, *see* RFC
REQUEST-CERTIFICATE message, 457, 460
Request-URI, 316, 327, 391
  HTTP, 294, 302
  S-HTTP use of, 391
responder, IKE, 382
response, HTTP, 296
resumption, SSL session, 181–182

retinal scan, 153
revocation, certificate, 13–14, 238
RFC (Request For Comments), 43
  AH, 380
  CMS, 400
  DNS Security, 409
  ESP, 380
  HTTP, 293
  HTTP over TLS, 245, 308, 319
  HTTP Upgrade, 316, 329
  IKE, 380
  Internet Mail, 344, 349–351, 380
  ISAKMP, 380
  PKIX, 10
  S/MIMEv3, 400, 404
  Security Architecture for Internet Protocol, 380
  SMIMEv3 Enhanced Security Services, 406
  SMTP, 343
  SMTP Extensions, 358
  SMTP over TLS, 343, 358–373
  TLS, 43, 102, 358
Rijndael, 31
Rivest, R., 33, 40
RNG (Random Number Generator), 140
  hardware, 155–156
  Pentium III, 156
rollback protection, SSLv2, 137
rollover, CBC, 29, 165
root CA, 158, 187, 253, 458
RSA
  acceleration of, 205
  client authentication time allocation, 188–189
  digital signature, 36–37
  ephemeral, 97–99, 112–115, 183, 253
  handshake timeline, 60
  handshake trace, 67
  key exchange, 33–34, 79
  mode time allocation, 186–188
  modulus, 40, 114
  patent, 34
  performance of, 37, 182, 186
  Security, 53
  signature SSL client authentication, 112
  strength of, 40
  time allocation, ephemeral, 189–191
  vs. DSS, 182
RSA_EXPORT cipher suite, 114, 133
rsa_fixed_dh certificate type, 110, 126
rsa_sign certificate type, 110
RST
  during close_notify, 283, 285
  segment, TCP, 245–246
run() method, 273

S/Key, 222
S/MIME (Secure MIME), 16, 379, 400–408
  algorithm choice, 403–405
  certificate, 161, 406–407
  implementation barriers, 406–407
  interaction with relay, 408
  multiple recipients and, 406
  multiple signers and, 406
  nonrepudiation, 408
  receipts, 406
  sender authentication, 405
  signing only, 401–402
  and virtual hosts, 408
  vs. SMTP/TLS, 407–408
S/MIMEv3, 3DES required by, 404
s_client program, 184
s_server program, 184
S-HTTP (Secure HTTP), 379, 389–400
  automatic options generation, 396
  client authentication, 395–396
  CMS use in, 389
  cryptographic options, 389, 392–393
  Key-Assign header, 396–397
  message format, 389, 391
  negotiation, 389, 392–393
  nonrepudiation, 398
  and proxy, 398
  reference integrity, 396
  use of, Request-URI, 391
  vs. HTTPS, 397–400
SafePassage, 319
salting, passphrase, 150
Saltzer, J., 378
sample programs, 249–289, 331–341, 411–436
  shortcomings, 286–288
Schneier, B., 40, 50
Schnorr, C., 37
Schroeder, M.D., 8
sclient.c file, 261–262, 336, 374, 412, 422–423
SClient.java file, 271, 429–430
SClient class, 286, 425
scrubbing, memory, 145
SCSI bus, 204
secondary channels, 244
secret key cryptography, defined, 6
secrets, protecting in memory, 145–146
Secure DNS, 233
secure
  channel, 17–24
  messaging, 16–17
Secure Hash Algorithm 1, *see* SHA-1
Secure HTTP, *see* S-HTTP
Secure MIME, *see* S/MIME
Secure News, *see* SNEWS
Secure Sockets Layer, *see* SSL
Secure Transport Layer Protocol, *see* STLP
SecureID, 222
security
  associations, 380–381
  considerations, HTTP, 293

goals, 3–4
indicators, e-mail, 365–366
Internet mail, 343–345
primitives, 15
properties, connection, 223
protocols, object, 379
SMTP/TLS and link-level, 379
STARTTLS hop-by-hop, 371
STARTTLS relaying vs., 369–371
Security Parameters Index, *see* SPI
Security-Scheme header, 396
seeding PRNG, 155, 253
select()
function, 271, 274–275, 277, 282, 289, 336
function, I/O multiplexed with, 274–282
function, SSL read and, 274–276
function, SSL write and, 276–279
masks, 282
select type, 69–70
sendClose() method, 283–285
sender
authentication, 4
authentication, S/MIME, 405
authentication, STARTTLS lack of, 373
identity, e-mail, 351
Sender, 81
value in Finished message, 81
Sendmail program, 345
separate port, 225, 228–229
client authentication and, 242
defined, 52
firewall and, 229
HTTPS and, 305
IETF and, 229
problems with, 225
strategy for SMTP, 358
TCP port consumption, 229
when to use, 227
seq_num, 89
sequence number
SSL, 89
TSP, 22
serialization, handshake, 185–186
Serpent, 31
server
accept, OpenSSL, 264–265
accept, PureTLS, 264
authentication, HTTPS, 308–309
authentication, SSL, 45
echo, 251
identity, 156–157, 222–223
initialization, SSL, 253, 257–258
mail, 343
private key, 58, 97–103, 119, 375
private key securing, 146–154
public key, 58, 64, 77–79, 125, 239
startup, STARTTLS, 374–375
Server Gated Cryptography, *see* SGC
server.c file, 412, 419–420
server.h file, 418–419
Server.java file, 264, 429
Server class, 265, 425
server_random, 86–87, 143
server_version value, ServerHello message, 75
server_write_iv, 127–128
server_write_key, 165
SERVER-FINISH message, 457
SERVER-HELLO message, 456–457, 459
SERVER-VERIFY message, 457
ServerDHParams value ServerKeyExchange message, 114
ServerFortezzaParams value, ServerKeyExchange message, 127
ServerHello
message, 60, 63, 75, 88, 96, 106, 112, 134, 173, 185, 211, 214–216, 335
message, cipher_suite value, 75
message, compression_method value, 75
message, random value, 75
message, server_version value, 75
message, session_id value, 76, 96, 106
trace, 75
ServerHelloDone message, 60, 64, 78, 185, 211, 214–215
ServerKeyExchange
computation, 114
message, 77, 98, 102, 105, 114, 125–126, 147, 165, 167, 169, 171–173, 192, 196–197, 214–216, 238–240
message, cost of, 190–193, 195
message, ServerDHParams value, 114
message, ServerFortezzaParams value, 127
message, ServerRSAParams value, 114
trace, 113
ServerRSAParams value ServerKeyExchange message, 114
servers
multiprocess, 337–339
UNIX mail, 374
organizational, 353–354
under load, SSL, 202–204
ServerSocket.accept() method, 264
ServerSocket class, 263–264
session
cache, mod_ssl, 436–453
cache, PureTLS, 340
cache, SSL, 181, 339–341, 397
ID lookup, 108
key, 15
resumption, 128, 184
resumption, cost of, 181–182
resumption, SSL, 96–97, 106–108, 181–182, 285–286
resumption, SSLv2, 136
resumption, trace, Nagle Algorithm disabled and, 213

resumption timeline, 97
resumption trace, 107, 213
SSL connection vs., 96
SESSION object, 286
session_id
messageClientHello, 106
messageServerHello, 96, 106
value, ClientHello message, 74–75, 96, 106
value, ServerHello message, 76, 96, 106
SESSION-ID message, 136
SGC (Server Gated Cryptography), 100–101, 119–124
and CA, 100
certificate, 100–101, 119–124
Microsoft and, 100
OpenSSL support for, 124
rehandshake, 100
trace, 119–123
see also Step-Up
SHA-1 (Secure Hash Algorithm 1), 7, 32, 81–88, 111–112, 165–166
DSS use with, 37
performance of, 183, 198
and TLS, 51
sha_hash
computation, 114
SSLv3 Finished, 81
value, Finished message, 125
Shamir, A., 30, 33, 40
Shared Memory, *see* SHM
Shimomura, T., 40
SHM (Shared Memory), 340, 453
shttp:// URL scheme, 304, 392
shutdown_wait variable, 283
signal() function, 283
signature
certificate, 10
CMS detached, 390
digital, 9
SignedData CMS content type, 390–391, 401
signing
object, 161
only, S/MIME, 401–402
SIGPIPE UNIX signal, 283
Simon, D., 47
Simple Mail Transfer Protocol, *see* SMTP
single point of failure, defined, 2
size, private key, 184
SKEME, 381
SKIPJACK, 105
small-subgroup
attack, DH, 170–171
attack countermeasures, 171
man-in-the-middle timeline, 240
smart host, 354–355
smartcards, 152
SMIMECapabilities S/MIME extension, 403
SMTP (Simple Mail Transfer Protocol), 225, 251, 343–344, 346–349
DATA command, 349
EHLO command, 349
extensions, 357–358
HELO command, 349
MAIL command, 349, 351
over SSL, 52
over TLS, 343–376
proxy, 352
QUIT command, 349, 363
RCPT command, 349
relaying, 369–371
relays, open, 354–355
STARTTLS command, 358–363
trace, 346–349
virtual host, 344
SMTP/TLS
and last-hop delivery, 378–379
and link-level security, 379
problems with proxy, 378
and relaying, 379
S/MIME vs., 407–408
upward negotiation, 357
SNEWS (Secure News), 52
socket() function, 260
Socket class, 258, 264
spam, 354–355
SPI (Security Parameters Index), 381
Spies, T., 47
SPYRUS, 319, 340
TLSGold, 253
SPYRUS/Terisa Systems, 53
sserver.c file, 258, 265, 412, 421–422
sserver program, 265
SSL (Secure Sockets Layer), xvii–464
backward-compatible, 49, 73
buffering, 185, 277
client authentication, 45, 96–97, 108–112, 222, 236–238
client authentication, RSA signature, 112
client initialization, 253
closure, 93, 244–247, 282–285
congestion control and, 212
connection vs. session, 96
context initialization, 252–258
context object, 252
deployment, 54
design goal confidentiality, 45
design goals, 44–45
designed for WWW, 44
designing with, 219–247
DH and, 49, 102, 124–126
DSS and, 49, 102, 124–126
encryption, CBC, 91
error alerts, 93
export, 25
firewall and, 224

FORTEZZA, 49, 104–106, 126–128
FTP over, 52, 244
handshake, 58–60, 95–101, 171–173, 258–265
handshake buffering latency, 214–216
handshake buffering performance cost, 217
handshake message, 59–60, 70
handshake over high-latency network timeline, 211
handshake parallelism, 216
handshake time allocation, 184–197
handshake timeline, 64
HTTP over, 44, 291–341
inappropriate tasks for, 224
incompatible with UDP, 90
IPsec vs., 386–388
key derivation, 82–88, 143
MAC, 108, 130, 180, 274
MAC key, 144
message integrity required by, 221
and NAT, 388
negotiation, 59, 72–76
Netscape de facto standard for, 50
and network latency, 209–212
NNTP over, 52
obtaining, 53–54
optimal record size, 183–184, 198–199
over NNTP, 45
partial compromise, 162–166
performance rules, 184, 216–217
place in protocol stack, 46
read and `select()` function, 274–276
record, 61
record, overhead of, 198–202
record content type, 61
record encryption, 90–91
record processing, performance of, 197–199
record protocol, 61–65, 88–91
reference integrity, 222–223
rehandshake, 99–100, 242–244
and reliable transport, 47, 90
sequence number, 89
server authentication, 45
server initialization, 253, 257–258
servers under load, 202–204
session cache, 181, 339–341, 397
session resumption, 96–97, 106–108, 181–182, 285–286
SMTP over, 52
specification language, 68–70
TCP interaction with, 46–47
Telnet over, 52
transparency, 44
use of random numbers, 140
use with any TCP-based protocol, 45
write and `select()` function, 276–279
`SSL` object, 261, 264, 266, 269, 283, 286, 328
`SSL_accept()` function, 263, 265
`SSL_connect()` function, 265, 286
`SSL_CTRL_SET_READ_AHEAD` flag, 268
`SSL_CTX` object, 255, 261
`SSL_CTX_new()` function, 255
`SSL_CTX_set_verify_depth()` function, 262
`ssl_engine_kernel.c` file, 436, 439
`ssl_engine_scache.c` file, 440, 443–444, 449
`SSL_ERROR_WANT_READ` error, 275–276
`SSL_ERROR_WANT_WRITE` error, 279
`SSL_get_error()` function, 268–269
`SSL_get_session()` function, 286
`SSL_handshake()` function, 338
`SSL_IsMoreWriteData()` function, 279
`SSL_library_init()` function, 255
`SSL_MODE_ENABLE_PARTIAL_WRITE` flag, 269
`SSL_pending()` function, 275–276, 279
`SSL_read()` function, 266, 269, 271, 275, 283
`ssl_session_cache.c` file, 450, 453
`SSL_set_session()` function, 286
`SSL_shutdown()` function, 269, 283
`SSL_write()` function, 266, 268–269, 277–279
`SSLContext.java` file, 339
`SSLContext` object, 255, 340
`ssldump` program, xxi, 65
SSLeay, 53, 249
SSLREF, 137
`SSLServerSocket` class, 264, 425
`SSLSocket` class, 258, 264
SSLv2-compatible cipher suite table, 137
`SSLv3_method()` function, 255
`SSLv23_method()` function, 255
SSLv1, 47, 49
integrity attack, 458
SSLv2, 44, 47, 49, 455–464
backward-compatible handshake, 135–137
CHALLENGE value, 136
client authentication transfer attack, 459–461
CRC in, 464
downgrade attack, 459
exportability, 49
handshake, 455–457
MAC, 457, 459
rollback protection, 137
session resumption, 136
timeline, 456
truncation attack, 459
weaknesses of, 49, 308
SSLv3, 47, 49
backward-compatible, 49
CertificateVerify computation, 111
export key derivation, 87–88
Finished, md5_hash, 81
Finished, sha_hash, 81
Finished computation, 81
incompatibility with TLS, 50
key derivation, 87–88

MAC pad_1, 89
MAC pad_2, 89
record format, 49
specification language, 49
support for, 54
`start()` method, 265, 272–273
starting threads, 272–274
STARTTLS
message trace, 365
trace, 359–362
STARTTLS
authenticated relaying, 366–367
close_notify unnecessary with, 363–364
command, SMTP, 358–363
and CONNECT, 371–372
connection closure, 363–364
downgrade attack, 369
enforcing security, 369
hop-by-hop security, 371
implementation, 373–374
implementing, 373–374
lack of sender authentication, 373
need for transitive security, 369
originator authentication, 367
reference integrity, 367–371
relaying vs. security, 369–371
server startup, 374–375
use of, 372–373
useful against passive attack, 372
and virtual host, 364–365
startup, STARTTLS server, 374–375
static DH, 126, 146
Station-To-Station protocol, *see* STS
`stdio`, 373
Step-Up, 100–101, 115–119
certificate, 100–101, 115–119
mod_ssl use of, 118
Netscape use of, 100, 118
rehandshake, 119
timeline, 101
trace, 115–118
see also SGC
STLP (Secure Transport Layer Protocol), 47, 49–50
storageless key recovery, 150–151
store-and-forward messaging, 346
`storeSession()` method, 340
stream ciphers, 26–27, 91
XOR in, 26–27
strength
of 3DES, 30, 164
algorithm, 162
of DES, 29–30
of DH, 35, 40
of DSS, 40
of RC2, 31
of RC4, 27
of RSA, 40
`Strings` class, 258
strong, prime, 171
Stronghold, 54
struct type, 69
STS (Station-To-Station protocol), 240, 381
subject
name, 10
public key, 10
subjectAltName certificate extension, 13, 157, 161–162, 234, 286, 308, 405
substitution attack, HTTPS and, 327–328
summary of effects of private key compromise, 164
Sun, 54
superencryption, 30
superserver, 374
symmetric
algorithms, performance of, 31, 184
cryptography, 9
performance of OpenSSL, 197

table
alert, 129
cipher suite, 74
FORTEZZA cipher suite, 128
PCT cipher suite, 463
SSLv2-compatible cipher suite, 137
tags, HTML, 295
tamper-evident, defined, 5
tamper-proof, defined, 5
tamper-resistant, defined, 5
TCP
connection, hijacking, 144
delayed ACK, 212
FIN, 65, 93, 244–246, 283, 285, 307, 459
half-close, 245
interaction with, SSL, 46–47
port consumption separate port, 229
RST segment, 245–246
`tcp_connect()` function, 260–261, 335
`tcpdump` program, xxi
TEK (Token Encryption Key), 127
Telnet, 225
over SSL, 52
`telnet` program, 251
Terisa Systems, 54
text, entropy of, 149
`text/html` MIME content type, 295
TGS (Ticket Granting Server), 104
Thawte, 100
third party, trusted, 12
Thompson, E., 40
`Thread` class, 265, 272–273
threads
blocking and, 271
I/O multiplexed with, 269–274
starting, 272–274
support, UNIX, 274

threat model, 154, 220
  defined, 1
  identification of, 227
throughput, latency vs., 178
Ticket Granting Server, *see* TGS
time allocation
  DSS/DH, 191–194
  DSS/DH client authentication, 194–195
  ephemeral RSA, 189–191
  RSA client authentication, 188–189
  RSA mode, 186–188
  SSL handshake, 184–197
timeline
  client authentication, 98
  DSS/DH, 103
  DSS/DH expanded buffer, 215
  DSS/DH normal buffer, 214
  ephemeral RSA, 99
  FORTEZZA, 105
  HTTPS man-in-the-middle proxy, 318
  incomplete close, 245
  Internet Explorer accepting bad certificate, 313
  man-in-the-middle, 239
  Netscape accepting bad certificate, 311
  Netscape rejecting bad certificate, 312
  password-based mutual authentication, 241
  RSA handshake, 60
  session resumption, 97
  small-subgroup man-in-the-middle, 240
  SSL handshake, 64
  SSL handshake over high-latency network, 211
  SSLv2, 456
  Step-Up, 101
  upward negotiation, 226
  upward negotiation downgrade attack, 231
timing cryptanalysis, 166
  attack, 167–168
  countermeasures, 168
tinygrams, 212, 214
TLS (Transport Layer Security), xvii–464
  3DES and, 51
  backward-compatible, 50, 73
  CertificateVerify computation, 112
  DH and, 51
  DSS and, 51
  export key derivation, 86–87
  Finished computation, 82
  and IETF, 44, 49, 51
  incompatibility with SSLv3, 50
  key derivation, 82–87
  MAC, 89–90
  mandatory algorithms, 51
  master_secret computation, PRF, 85
  PRF key_block computation, 86
  required algorithms, 50
  SHA-1 and, 51
  SMTP over, 343–376
  support for, 51, 54
  working group, IESG and, 50
TLS_NULL_WITH_NULL_NULL cipher suite, 91
TLS_RSA_WITH_3DES_EDE_CBC_SHA cipher suite, 179, 202
TLS_RSA_WITH_DES_CBC_SHA cipher suite, 75, 88
TLSGold, 340
  write handling, 279
Token Encryption Key, *see* TEK
Toy Security Protocol, *see* TSP
trace
  alert, 93
  Certificate, 77
  CertificateRequest, 109
  CertificateVerify, 111
  ClientHello, 73
  DH ClientKeyExchange, 126
  DH ServerKeyExchange, 125
  downgrade attack, 172
  Finished, 81
  HTTPS, 305–307
  HTTPS CONNECT, 317
  Nagle Algorithm disabled and session resumption, 213
  rehandshake with client authentication, 322–325
  RSA handshake, 67
  ServerHello, 75
  ServerKeyExchange, 113
  session resumption, 107, 213
  SGC, 119–123
  SMTP, 346–349
  STARTTLS, 359–362
  STARTTLS message, 365
  Step-Up, 115–118
transferable, password, 235
transitive security, STARTTLS need for, 369
transparent acceleration, 206
transport
  identity, e-mail, 351
  ISAKMP, 383–384
  mode, IPsec, 385
Transport Control Protocol, *see* TCP
Transport Layer Security, *see* TLS
Triple DES, *see* 3DES
Triple-DES, 6
truncation attack, 22, 93, 244, 315–316
  SSLv2, 459
trusted third party, 8, 12
TSP (Toy Security Protocol), 17–24
  data records, 20–24
  handshake, 18–19
  key derivation, 20
  key derivation function, 20
  lack of, negotiation, 24
  MAC, 20–21
  master secret, 18
  record length, 21

sequence number, 22
stages, 18
type field, 22
TSW_SSL_IsMoreWriteData() function, 279
TSW_SSL_Write() function, 279
tunnel
mode, IPsec, 385
mode used in VPN, 385
Twinkle, 40
Twofish, 31
type
anchor method, 298
field, TSP, 22
field MAC of, 22

UDP, SSL incompatible with, 90
uint8 type, 68–69
uint16 type, 68
uint24 type, 68
uint32 type, 68
unexpected_message alert, 130
Uniform Resource Identifier, *see* URI
Uniform Resource Locator, *see* URL
Uniform Resource Name, *see* URN
UNIX, 250–271
local mail access, 345
mail servers on, 374
pre-forking, 341
threads support, 274
unknown_ca alert, 132
unsupported_certificate alert, 132
Upgrade, HTTP, 328–331
upgrade
mandatory, 329
and reference integrity, HTTP, 330–331
Upgrade
header, HTTP, 328–331
HTTP request, 328–330
upward negotiation, 225–227, 229–230
defined, 52
downgrade attack and, 229
firewall and, 230
HTTP, 328–331
SMTP/TLS, 357
downgrade attack timeline, 231
timeline, 226
when to use, 227
URI (Uniform Resource Identifier), 300
vs. URL, 300
URL (Uniform Resource Locator), 156, 292, 297–300
query, 299
reference integrity and, 307–314
relative, 299
scheme, https:, 223, 304
scheme, shttp://, 392
URI vs., 300

URN (Uniform Resource Name), 300
user_cancelled alert, 134
username/password
ACL use with, 221
authentication, 235–236
client authentication, 221
variants, 222

VBScript, 298
vector type, 68–69
Vector class, 258
verification, certificate chain, 141, 156–162
verify-prelude message, 461
Verisign, 52, 100
version number hack and PCT, 47
victim's server as oracle, 169
virtual
host, 302
host, e-mail, 355
host, HTTP, 293, 302
host, SMTP, 344
host, STARTTLS and, 364–365
host and, ISPs, 344
hosts, HTTPS and, 320–321
hosts, S/MIME and, 408
von Neumann, J., 156
VPN (Virtual Private Network), tunnel mode used in, 385

Wagner, D., 79
warning alert, 92–93, 128–129, 131–132, 134
weakness of 40-bit RC4, 164
weaknesses of
name resolution, 157
Netscape, 47, 79
PKCS #1, 169
SSLv2, 49, 308
Web pages, Java in, 298
Web Distributed Authoring and Versioning, *see* WEBDAV
WEBDAV (Web Distributed Authoring and Versioning), 294
Weiner, E., 40
well-known port, 52, 225
WG (Working Group), 49
wildcarding
DN, 308
Netscape, 321
Windows 2000 support for, IPsec, 388
Wireless Transport Layer Security, *see* WTLS
Working Group, *see* WG
World Wide Web, *see* WWW
wrapped_client_write_key FortezzaKeys structure, 127
wrapped_server_write_key FortezzaKeys structure, 127

write
  handling, OpenSSL, 268–269, 278–279
  handling, PureTLS, 269–274
  handling, TLSGold, 279
  and `select()` function, SSL, 276–279
`write()`
  function, 266
  method, 273, 277
`writestr()` function, 335
WTLS (Wireless Transport Layer Security), 47, 49
WWW (World Wide Web), 291–341
  SSL designed for, 44

X.509 certificate, 10–14
XOR in
  HMAC, 39
  stream ciphers, 26–27
  TLS PRF, 85

y_c FortezzaKeys structure, 127
y_signature FortezzaKeys structure, 127
YACC (Yet Another Compiler Compiler), 68
Yee, B., 47
Yet Another Compiler Compiler, *see* YACC
Young, E., 53, 249

zeroing memory, 145
zeroizing after bad PIN, 152

ACRONYMS

3DES	Triple DES, p. 6
ACL	Access Control List, p. 236
AES	Advanced Encryption Standard, p. 31
AH	Authentication Header, p. 380
ANSI	American National Standards Institute, p. 44
API	Application Programming Interface, p. 46
ASN.1	Abstract Syntax Notation 1, p. 14
AVA	Attribute-Value Assertion, p. 258
BER	Basic Encoding Rules, p. 14
BXA	Bureau of Export Administration, p. 24
CA	Certificate Authority, p. 9
CBC	Cipher Block Chaining, p. 28
CDMF	Commercial Data Masking Facility, p. 25
CESG	Communications-Electronics Security Group, p. 9
CFB	Cipher Block Feedback, p. 28
CMS	Cryptographic Messaging Syntax, p. 390
CN	Common Name, p. 13
CRC	Cyclic Redundancy Check, p. 464
CRL	Certificate Revocation List, p. 13
DER	Distinguished Encoding Rules, p. 14
DES	Data Encryption Standard, p. 6
DH	Diffie-Hellman, p. 34
DHCP	Dynamic Host Configuration Protocol, p. 108
DN	Distinguished Name, p. 12
DNS	Domain Name System, p. 156
DNSSEC	DNS Security, p. 409
DSA	Digital Signature Algorithm, p. 37
DSS	Digital Signature Standard, p. 37
EC	Elliptic Curve, p. 103
ECB	Electronic Code Book, p. 28
EDE	Encrypt-Decrypt-Encrypt, p. 30
EEE	Encrypt-Encrypt-Encrypt, p. 30
ESP	Encapsulating Security Payload, p. 380
FIPS	Federal Information Processing Standard, p. 37
FTP	File Transfer Protocol, p. 244
HTML	Hypertext Markup Language, p. 295
HTTP	Hypertext Transfer Protocol, p. 293
HTTPS	traditional HTTP over SSL, p. 304
I/O	Input/Output, p. 177
IANA	Internet Assigned Numbers Authority, p. 225
IE	Internet Explorer, p. 54
IESG	Internet Engineering Steering Group, p. 50
IETF	Internet Engineering Task Force, p. 43
IIS	Internet Information Server, p. 54
IKE	Internet Key Exchange, p. 380
IMAP	Internet Mail Access Protocol, p. 346
IP	Internet Protocol, p. 46
IPsec	Internet Protocol Security, p. 380
ISAKMP	Internet Security Association and Key Management Protocol, p. 380
ISO	International Standards Organization, p. 15
ISP	Internet Service Provider, p. 302
ITU	International Telecommunications Union, p. 44
IV	Initialization Vector, p. 28
JCA	Java Cryptographic Architecture, p. 200
JDK	Java Development Kit, p. 200
JSSE	Java Secure Sockets Extension, p. 258
KDC	Key Distribution Center, p. 8
KDF	Key Derivation Function, p. 20
KEA	Key Exchange Algorithm, p. 105

LDAP	Lightweight Directory Access Protocol, p. 299
LEAF	Law Enforcement Access Field, p. 126
MAC	Message Authentication Code, p. 7
MD5	Message Digest 5, p. 7
MIME	Multipurpose Internet Mail Extensions, p. 344
MOSS	MIME Object Security Services, p. 391
MSS	Maximum Segment Size, p. 215
MTA	Mail Transport Agent, p. 345
MUA	Mail User Agent, p. 345
MX	Mail Exchanger, p. 356
NAT	Network Address Translation, p. 388
NBS	National Bureau of Standards, p. 29
NIST	National Institute of Standards and Technology, p. 29
NNTP	Network News Transfer Protocol, p. 45
NSA	National Security Agency, p. 24
NSE	Non-Secret Encryption, p. 9
OAEP	Optimal Asymmetric Encryption Padding, p. 170
OCSP	Online Certificate Status Protocol, p. 13
OFB	Output Feedback Mode, p. 28
OID	Object Identifier, p. 15
OSI	Open Systems Interconnect, p. 14
PCT	Private Communications Technology, p. 461
PEM	Privacy Enhanced Mail, p. 16
PFS	Perfect Forward Secrecy, p. 35
PGP	Pretty Good Privacy, p. 16
PIN	Personal Identification Number, p. 152
PKC	Public Key Cryptography, p. 9
PKCS	Public Key Cryptography Standards, p. 33
PKIX	IETF Public Key Infrastructure WG, p. 50
POP	Post Office Protocol, p. 346
PRF	Pseudo-Random function, p. 82
PRNG	Pseudo-Random Number Generator, p. 140
RDN	Relative Distinguished Name, p. 258
RFC	Request For Comments, p. 43
RNG	Random Number Generator, p. 140
RSA	Rivest Shamir Adelman, p. 33
RSADSI	RSA Data Security, Inc., p. 40
S-HTTP	Secure HTTP, p. 389
S/MIME	Secure MIME, p. 400
SGC	Server Gated Cryptography, p. 115
SHA-1	Secure Hash Algorithm 1, p. 7
SHM	Shared Memory, p. 453
SMTP	Simple Mail Transfer Protocol, p. 346
SMTPS	SMTP Secure, p. 358
SNEWS	Secure News, p. 52
SPI	Security Parameters Index, p. 381
SSL	Secure Sockets Layer, p. xvii
STLP	Secure Transport Layer Protocol, p. 47
STS	Station-To-Station protocol, p. 240
TCP	Transport Control Protocol, p. 46
TEK	Token Encryption Key, p. 127
TGS	Ticket Granting Server, p. 104
TLS	Transport Layer Security, p. xvii
TSP	Toy Security Protocol, p. 17
URI	Uniform Resource Identifier, p. 300
URL	Uniform Resource Locator, p. 298
URN	Uniform Resource Name, p. 300
WEBDAV	Web Distributed Authoring and Versioning, p. 294
WG	Working Group, p. 49
WTLS	Wireless Transport Layer Security, p. 47
WWW	World Wide Web, p. 44
XOR	Exclusive Or, p. 26

Register Your Book

at www.aw.com/cseng/register



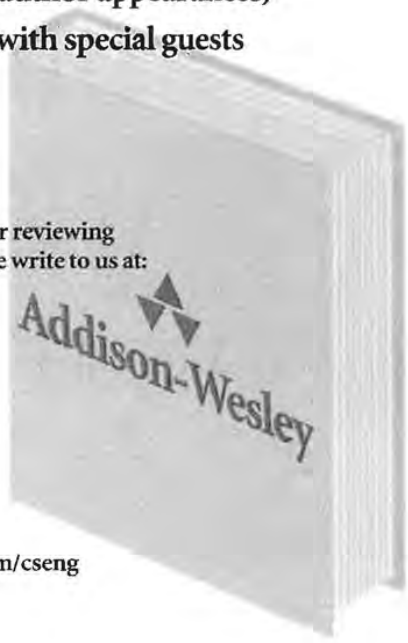
You may be eligible to receive:

- Advance notice of forthcoming editions of the book
- Related book recommendations
- Chapter excerpts and supplements of forthcoming titles
- Information about special contests and promotions throughout the year
- Notices and reminders about author appearances, tradeshows, and online chats with special guests

Contact us

If you are interested in writing a book or reviewing manuscripts prior to publication, please write to us at:

Editorial Department
Addison-Wesley Professional
75 Arlington Street, Suite 300
Boston, MA 02116 USA
Email: AWPro@aw.com



Visit us on the Web: http://www.aw.com/cseng

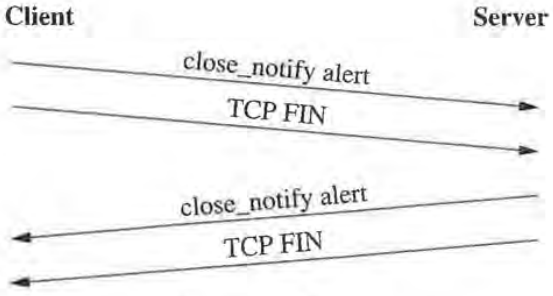
SSL/TLS cipher suites

Cipher Suite	Auth	Key Exchange	Encryption	Digest	Number
TLS_RSA_WITH_NULL_MD5	RSA	RSA	NULL	MD5	0x0001
TLS_RSA_WITH_NULL_SHA	RSA	RSA	NULL	SHA	0x0002
TLS_RSA_EXPORT_WITH_RC4_40_MD5	RSA	RSA_EXPORT	RC4_40	MD5	0x0003
TLS_RSA_WITH_RC4_128_MD5	RSA	RSA	RC4_128	MD5	0x0004
TLS_RSA_WITH_RC4_128_SHA	RSA	RSA	RC4_128	SHA	0x0005
TLS_RSA_EXPORT_WITH_RC2_CBC_40_MD5	RSA	RSA_EXPORT	RC2_40_CBC	MD5	0x0006
TLS_RSA_WITH_IDEA_CBC_SHA	RSA	RSA	IDEA_CBC	SHA	0x0007
TLS_RSA_EXPORT_WITH_DES40_CBC_SHA	RSA	RSA_EXPORT	DES40_CBC	SHA	0x0008
TLS_RSA_WITH_DES_CBC_SHA	RSA	RSA	DES_CBC	SHA	0x0009
TLS_RSA_WITH_3DES_EDE_CBC_SHA	RSA	RSA	3DES_EDE_CBC	SHA	0x000A
TLS_DH_DSS_EXPORT_WITH_DES40_CBC_SHA	RSA	DH_DSS_EXPORT	DES_40_CBC	SHA	0x000B
TLS_DH_DSS_WITH_DES_CBC_SHA	DSS	DH	DES_CBC	SHA	0x000C
TLS_DH_DSS_WITH_3DES_EDE_CBC_SHA	DSS	DH	3DES_EDE_CBC	SHA	0x000D
TLS_DH_RSA_EXPORT_WITH_DES40_CBC_SHA	RSA	DH_EXPORT	DES_40_CBC	SHA	0x000E
TLS_DH_RSA_WITH_DES_CBC_SHA	RSA	DH	DES_CBC	SHA	0x000F
TLS_DH_RSA_WITH_3DES_EDE_CBC_SHA	RSA	DH	3DES_EDE_CBC	SHA	0x0010
TLS_DHE_DSS_EXPORT_WITH_DES40_CBC_SHA	DSS	DHE_EXPORT	DES_40_CBC	SHA	0x0011
TLS_DHE_DSS_WITH_DES_CBC_SHA	DSS	DHE	DES_CBC	SHA	0x0012
TLS_DHE_DSS_WITH_3DES_EDE_CBC_SHA	DSS	DHE	3DES_EDE_CBC	SHA	0x0013
TLS_DHE_RSA_EXPORT_WITH_DES40_CBC_SHA	RSA	DHE_EXPORT	DES_40_CBC	SHA	0x0014
TLS_DHE_RSA_WITH_DES_CBC_SHA	RSA	DHE	DES_CBC	SHA	0x0015
TLS_DHE_RSA_WITH_3DES_EDE_CBC_SHA	RSA	DHE	3DES_EDE_CBC	SHA	0x0016
TLS_DH_anon_EXPORT_WITH_RC4_40_MD5	-	DH_EXPORT	RC4_40	MD5	0x0017
TLS_DH_anon_WITH_RC4_128_MD5	-	DH	RC4_128	MD5	0x0018
TLS_DH_anon_EXPORT_WITH_DES40_CBC_SHA	-	DH	DES_40_CBC	SHA	0x0019
TLS_DH_anon_WITH_DES_CBC_SHA	-	DH	DES_CBC	SHA	0x001A
TLS_DH_anon_WITH_3DES_EDE_CBC_SHA	-	DH	3DES_EDE_CBC	SHA	0x001B
TLS_RSA_EXPORT1024_WITH_DES_CBC_SHA †	RSA	RSA	DES_CBC	SHA	0x0062
TLS_DHE_DSS_EXPORT1024_WITH_DES_CBC_SHA †	RSA	RSA	DES_CBC	SHA	0x0063
TLS_RSA_EXPORT1024_WITH_RC4_56_SHA †	RSA	RSA	RC4_56	SHA	0x0064
TLS_DHE_DSS_EXPORT1024_WITH_RC4_56_SHA †	RSA	RSA	RC4_56	SHA	0x0065
TLS_DHE_DSS_WITH_RC4_128_SHA †	RSA	RSA	RC4_56	SHA	0x0066

Cipher suites marked with a † are 56-bit export ciphers defined in [Banes1999] not in RFC 2246.

SSL/TLS alerts

Alert Name	Level	SSL Version	Number
close_notify	either	either	0
unexpected_message	fatal	either	10
bad_record_mac	fatal	either	20
decryption_failed	fatal	TLS	21
record_overflow	fatal	TLS	22
decompression_failure	fatal	either	30
handshake_failure	fatal	either	40
no_certificate	either	SSLv3	41
bad_certificate	either	either	42
unsupported_certificate	either	either	43
certificate_revoked	either	either	44
certificate_expired	either	either	45
certificate_unknown	either	either	46
illegal_parameter	fatal	either	47
unknown_ca	fatal	TLS	48
access_denied	fatal	TLS	49
decode_error	fatal	TLS	50
decrypt_error	either	TLS	51
export_restriction	fatal	TLS	60
protocol_version	fatal	TLS	70
insufficient_security	fatal	TLS	71
internal_error	fatal	TLS	80
user_cancelled	fatal	TLS	90
no_renegotiation	warning	TLS	100



Closure exchange

Security/Networking

SSL and TLS

Praise for *SSL and TLS: Designing and Building Secure Systems*

"This is the best book on SSL/TLS. Rescorla knows SSL/TLS as well as anyone and presents it both clearly and completely.... At times, I felt like he'd been looking over my shoulder when I designed SSL v3. If network security matters to you, buy this book."
—Paul Kocher, Cryptography Research, Inc., co-designer of SSL v3

"Having the right crypto is necessary but not sufficient to having secure communications. If you're using SSL/TLS, you should have SSL and TLS *sitting on your shelf right next to* Applied Cryptography."
—Bruce Schneier, CTO of Counterpane Internet Security, Inc.

"Everything you wanted to know about SSL/TLS in one place. It covers the protocols down to the level of packet traces. It covers how to write software that uses SSL/TLS. And it contrasts SSL with other approaches. All this while being technically sound and readable!"
—Radia Perlman, Sun Microsystems, Inc., author of *Interconnections*

Secure Sockets Layer (SSL) and its IETF successor, Transport Layer Security (TLS), are the leading Internet security protocols, providing security for e-commerce, web services, and many other network functions. Using SSL/TLS effectively requires a firm grasp of its role in network communications, its security properties, and its performance characteristics. ***SSL and TLS*** provides total coverage of the protocols from the bits on the wire up to application programming.

This comprehensive book not only describes how SSL/TLS is supposed to behave but also uses the author's free ssldump diagnostic tool to show the protocols in action. The author covers each protocol feature, first explaining how it works and then illustrating it in a live implementation. This unique presentation bridges the difficult gap between specification and implementation that is a common source of confusion and incompatibility.

In addition to describing the protocols, ***SSL and TLS*** delivers the essential details required by security architects, application designers, and software engineers. Use the practical design rules in this book to quickly design fast and secure systems using SSL/TLS. These design rules are illustrated with chapters covering the new IETF standards for HTTP and SMTP over TLS.

Written by an experienced SSL implementor, ***SSL and TLS*** contains detailed information on programming SSL applications. The author discusses the common problems faced by implementors and provides complete sample programs illustrating the solutions in both C and Java. The sample programs use the free OpenSSL and PureTLS toolkits so the reader can immediately run the examples.

Eric Rescorla is an Internet security consultant and author of several commercial SSL implementations, including the freely available Java PureTLS toolkit. He is also the author of *HTTP over TLS* and *Secure HTTP IETF* RFCs.

http://www.awl.com/cseng/

Cover design by Karin Hansen
Cover illustration by Lois and Bob Schlowsky, Stone

Text printed on recycled paper

ADDISON-WESLEY
Pearson Education

53995
9 780201 615982
ISBN 0-201-61598-3
$39.95 US $59.95 CANADA

ATTACHMENT A-1

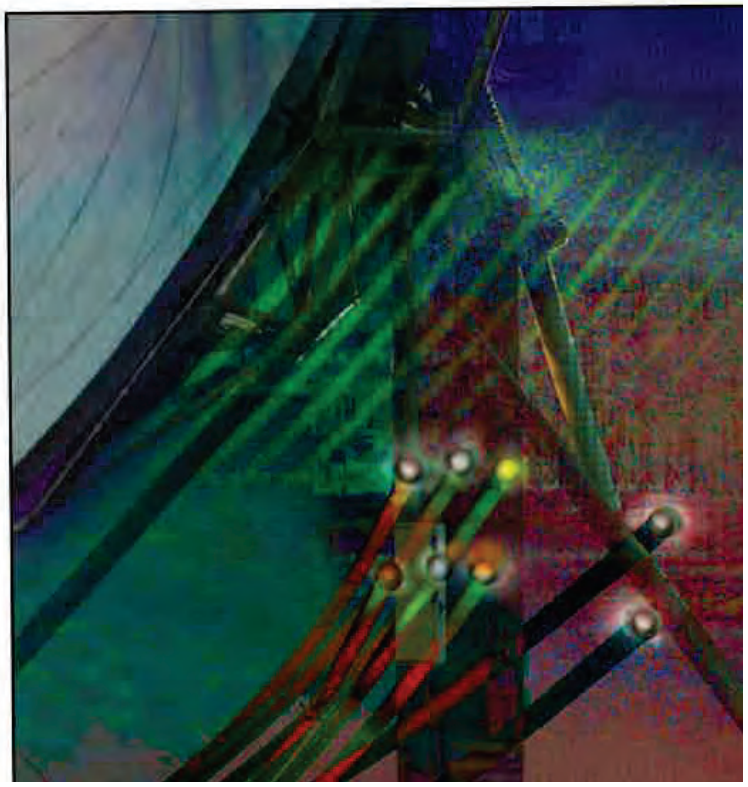
RESCORLA

SSL AND
TLS

TK 5105
.59
.R47
2001
Copy 1

SSL and TLS

Designing and Building Secure Systems



Eric Rescorla

Ex.1024
Askeladden, L.L.C. / Page 577 of 788

SSL and TLS

Designing and Building Secure Systems

Eric Rescorla



Addison-Wesley

Boston • San Francisco • New York • Toronto • Montreal
London • Munich • Paris • Madrid
Capetown • Sydney • Tokyo • Singapore • Mexico City

LIBRARY OF CONGRESS
NOV - 7 2000

Many of the designations used by manufacturers and sellers to distinguish their products are claimed as trademarks. Where these designations appear in this book and we were aware of a trademark claim, the designations have been printed in initial capital letters or all capitals.

The author and publisher have taken care in the preparation of this book, but make no expressed or implied warranty of any kind and assume no responsibility for errors or omissions. No liability is assumed for incidental or consequential damages in connection with or arising out of the use of the information contained herein.

Copyright © 2001 by Addison-Wesley

All rights reserved. No part of this publication may be reproduced, stored in a retrieval system, or transmitted, in any form or by any means, electronic, mechanical, photocopying, recording, or otherwise, without the prior consent of the publisher. Printed in the United States of America. Published simultaneously in Canada.

The publisher offers discounts on this book when ordered in quantity for special sales. For more information please contact:

Pearson Education Corporate Sales Division
One Lake Street
Upper Saddle River, NJ 07458
(800) 382-3419
corpsales@pearsontechgroup.com

Visit us on the Web at www.awl.com/cseng

TK5105.59
.R47
2001
Copy 1

Library of Congress Cataloging-in-Publication Data.

Rescorla, Eric, 1972-
SSL and TLS: designing and building secure systems / Eric Rescorla
p. cm.
Includes bibliographical references and index.
ISBN 0-201-61598-3 (alk. paper)
1. Computer networks—Security measures. 2. World Wide Web—Security measures
3. Computer network protocols. I. Title.
TK5105.59.R47 2000
05.8—dc21 00-057593

ISBN 0-201-61598-3

Text printed on recycled paper.
1 2 3 4 5 6 7 8 9 10-CRS-04 03 02 01 00
First printing, October 2000

Ex.1024
Askeladden, L.L.C. / Page 579 of 788

Contents

Preface		**xvii**
Chapter 1.	**Security Concepts**	**1**
1.1	Introduction	1
1.2	The Internet Threat Model	1
1.3	The Players	2
1.4	The Goals of Security	3
1.5	Tools of the Trade	5
1.6	Putting It All Together	15
1.7	A Simple Secure Messaging System	16
1.8	A Simple Secure Channel	17
1.9	The Export Situation	24
1.10	Real Cryptographic Algorithms	25
1.11	Symmetric Encryption: Stream Ciphers	26
1.12	Symmetric Encryption: Block Ciphers	27
1.13	Digest Algorithms	32
1.14	Key Establishment	32
1.15	Digital Signature	36
1.16	MACs	38
1.17	Key Length	39
1.18	Summary	40

Ex.1024
Askeladden, L.L.C. / Page 580 of 788

Chapter 2.	**Introduction to SSL**	**43**
2.1	Introduction	43
2.2	Standards and Standards Bodies	43
2.3	SSL Overview	44
2.4	SSL/TLS Design Goals	44
2.5	SSL and the TCP/IP Suite	46
2.6	SSL History	47
2.7	SSL for the Web	51
2.8	Everything over SSL	52
2.9	Getting SSL	53
2.10	Summary	55
Chapter 3.	**Basic SSL**	**57**
3.1	Introduction	57
3.2	SSL Overview	57
3.3	Handshake	58
3.4	SSL Record Protocol	61
3.5	Putting the Pieces Together	62
3.6	A Real Connection	63
3.7	Some More Connection Details	65
3.8	SSL Specification Language	68
3.9	Handshake Message Structure	70
3.10	Handshake Messages	71
3.11	Key Derivation	82
3.12	Record Protocol	88
3.13	Alerts and Closure	91
3.14	Summary	94
Chapter 4.	**Advanced SSL**	**95**
4.1	Introduction	95
4.2	Session Resumption	96
4.3	Client Authentication	96
4.4	Ephemeral RSA	97
4.5	Rehandshake	99
4.6	Server Gated Cryptography	100
4.7	DSS and DH	102
4.8	Elliptic Curve Cipher Suites	103
4.9	Kerberos	104
4.10	FORTEZZA	104
4.11	The Story So Far	106

4.12	Session Resumption Details	106
4.13	Client Authentication Details	108
4.14	Ephemeral RSA Details	112
4.15	SGC Details	115
4.16	DH/DSS Details	124
4.17	FORTEZZA Details	126
4.18	Error Alerts	128
4.19	SSLv2 Backward Compatibility	135
4.20	Summary	137
Chapter 5.	**SSL Security**	**139**
5.1	Introduction	139
5.2	What SSL Provides	139
5.3	Protect the master_secret	140
5.4	Protect the Server's Private Key	140
5.5	Use Good Randomness	140
5.6	Check the Certificate Chain	141
5.7	Algorithm Selection	142
5.8	The Story So Far	142
5.9	Compromise of the master_secret	142
5.10	Protecting Secrets in Memory	145
5.11	Securing the Server's Private Key	146
5.12	Random Number Generation	154
5.13	Certificate Chain Verification	156
5.14	Partial Compromise	162
5.15	Known Attacks	166
5.16	Timing Cryptanalysis	167
5.17	Million Message Attack	168
5.18	Small-Subgroup Attack	170
5.19	Downgrade to Export	171
5.20	Summary	173
Chapter 6.	**SSL Performance**	**175**
6.1	Introduction	175
6.2	SSL Is Slow	175
6.3	Performance Principles	176
6.4	Cryptography Is Expensive	179
6.5	Session Resumption	181
6.6	Handshake Algorithm and Key Choice	182
6.7	Bulk Data Transfer	183
6.8	Basic SSL Performance Rules	184

6.9	The Story So Far	184
6.10	Handshake Time Allocation	184
6.11	Normal RSA Mode	186
6.12	RSA with Client Authentication	188
6.13	Ephemeral RSA	189
6.14	DSS/DHE	191
6.15	DSS/DHE with Client Authentication	194
6.16	Performance Improvements with DH	195
6.17	Record Processing	197
6.18	Java	199
6.19	SSL Servers under Load	202
6.20	Hardware Acceleration	204
6.21	Inline Hardware Accelerators	206
6.22	Network Latency	209
6.23	The Nagle Algorithm	212
6.24	Handshake Buffering	214
6.25	Advanced SSL Performance Rules	216
6.26	Summary	217
Chapter 7.	**Designing with SSL**	**219**
7.1	Introduction	219
7.2	Know What You Want to Secure	220
7.3	Client Authentication Options	221
7.4	Reference Integrity	222
7.5	Inappropriate Tasks	224
7.6	Protocol Selection	224
7.7	Reducing Handshake Overhead	227
7.8	Design Strategy	227
7.9	The Story So Far	228
7.10	Separate Ports	228
7.11	Upward Negotiation	229
7.12	Downgrade Attacks	230
7.13	Reference Integrity	233
7.14	Username/Password Authentication	235
7.15	SSL Client Authentication	236
7.16	Mutual Username/Password Authentication	238
7.17	Rehandshake	242
7.18	Secondary Channels	244
7.19	Closure	244
7.20	Summary	247

Chapter 8. Coding with SSL **249**

8.1 Introduction 249
8.2 SSL Implementations 249
8.3 Sample Programs 250
8.4 Context Initialization 252
8.5 Client Connect 258
8.6 Server Accept 263
8.7 Simple I/O Handling 265
8.8 Multiplexed I/O Using Threads 269
8.9 Multiplexed I/O with `select()` 274
8.10 Closure 282
8.11 Session Resumption 285
8.12 What's Missing? 286
8.13 Summary 288

Chapter 9. HTTP over SSL **291**

9.1 Introduction 291
9.2 Securing the Web 291
9.3 HTTP 293
9.4 HTML 295
9.5 URLs 298
9.6 HTTP Connection Behavior 300
9.7 Proxies 301
9.8 Virtual Hosts 302
9.9 Protocol Selection 302
9.10 Client Authentication 303
9.11 Reference Integrity 303
9.12 HTTPS 304
9.13 HTTPS Overview 304
9.14 URLs and Reference Integrity 307
9.15 Connection Closure 314
9.16 Proxies 316
9.17 Virtual Hosts 320
9.18 Client Authentication 322
9.19 `Referrer` 327
9.20 Substitution Attacks 327
9.21 Upgrade 328
9.22 Programming Issues 331
9.23 Proxy `CONNECT` 331
9.24 Handling Multiple Clients 336
9.25 Summary 341

Chapter 10. SMTP over TLS **343**

10.1 Introduction 343
10.2 Internet Mail Security 343
10.3 Internet Messaging Overview 345
10.4 SMTP 346
10.5 RFC 822 and MIME 349
10.6 E-Mail Addresses 351
10.7 Mail Relaying 352
10.8 Virtual Hosts 355
10.9 MX Records 356
10.10 Client Mail Access 357
10.11 Protocol Selection 357
10.12 Client Authentication 357
10.13 Reference Integrity 357
10.14 Connection Semantics 358
10.15 STARTTLS 358
10.16 STARTTLS Overview 358
10.17 Connection Closure 363
10.18 Requiring TLS 364
10.19 Virtual Hosts 364
10.20 Security Indicators 365
10.21 Authenticated Relaying 366
10.22 Originator Authentication 367
10.23 Reference Integrity Details 367
10.24 Why Not CONNECT? 371
10.25 What's STARTTLS Good For? 372
10.26 Programming Issues 373
10.27 Implementing STARTTLS 373
10.28 Server Startup 374
10.29 Summary 375

Chapter 11. Contrasting Approaches **377**

11.1 Introduction 377
11.2 The End-to-End Argument 378
11.3 The End-to-End Argument and SMTP 378
11.4 Other Protocols 379
11.5 IPsec 380
11.6 Security Associations 381
11.7 ISAKMP and IKE 381
11.8 AH and ESP 384
11.9 Putting It All Together: IPsec 386
11.10 IPsec versus SSL 386

11.11	Secure HTTP	389
11.12	CMS	390
11.13	Message Format	391
11.14	Cryptographic Options	392
11.15	Putting It All Together: S-HTTP	393
11.16	S-HTTP versus HTTPS	397
11.17	S/MIME	400
11.18	Basic S/MIME Formatting	401
11.19	Signing Only	401
11.20	Algorithm Choice	403
11.21	Putting It All Together: S/MIME	405
11.22	Implementation Barriers	406
11.23	S/MIME versus SMTP/TLS	407
11.24	Choosing the Appropriate Solution	408
11.25	Summary	410
Appendix A.	**Example Code**	**411**
A.1	Chapter 8	411
A.1.1	C Examples	411
A.1.2	Java Examples	425
A.2	Chapter 9	431
A.2.1	HTTPS Examples	431
A.2.2	mod_ssl Session Caching	436
Appendix B.	**SSLv2**	**455**
B.1	Introduction	455
B.2	SSLv2 Overview	455
B.3	Missing Features	457
B.4	Security Problems	458
B.5	PCT	461
B.6	What about SSLv1?	463
Bibliography		**465**
Index		**475**

This material may be protected by Copyright law (Title 17 U.S. Code)

3

Basic SSL

3.1 Introduction

This chapter provides a detailed account of SSL operation for the most common case, server-only authentication using RSA. By titling this chapter Basic SSL, we mean that this is the basic mode of SSL, not that we're not going to go into depth about the protocol (we are). This chapter covers only the simplest mode of SSL. The next chapter will discuss several more-complicated modes.

This chapter is organized into two primary sections. The first is an overview of how SSL works and how the various pieces fit together. We'll broadly describe each protocol message and its general purpose. The second section dissects a sample SSL connection in detail. We use an automated tool to generate a session trace and then examine each protocol message in detail, seeing how the individual fields fit together to accomplish the job of the protocol. If you're reading this as an implementation guide or want a really detailed understanding of what's happening, you should read both sections. If your primary interest is simply in using SSL, you can probably safely skip the second half.

3.2 SSL Overview

The basic design of SSL should look fairly familiar from the Toy Security Protocol we designed in Chapter 1. A connection is divided into two phases, the *handshake* and *data transfer* phases. The handshake phase authenticates the server and establishes the cryptographic keys which are used to protect the data to be transmitted. The handshake must be completed before any application data can be transmitted. Once this has been done, the data is broken up and transmitted as a series of protected records.

Ex.1024
Askeladden, L.L.C. / Page 587 of 788

3.3 Handshake

The purpose of the SSL handshake is threefold. First, the client and the server need to agree on a set of algorithms which will be used to protect the data. Second, they need to establish a set of cryptographic keys which will be used by those algorithms. Third, the handshake may optionally authenticate the client, as we'll see in Chapter 4. First, we'll discuss the handshake process in general terms without discussing specific protocol messages and then we'll show how it breaks down into messages.

The overall process works something like this (see Figure 3.1 for a pictorial representation).

1. The client sends the server a list of the algorithms it's willing to support, along with a random number used as input to the key generation process.
2. The server chooses a cipher out of that list and sends it back along with a certificate containing the server's public key. The certificate also provides the server's identity for authentication purposes and the server supplies a random number which is used as part of the key generation process.
3. The client verifies the server's certificate and extracts the server's public key. The client then generates a random secret string called the pre_master_secret and encrypts it using the server's public key. It sends the encrypted public key to the server.
4. The client and server independently compute the encryption and MAC keys from the pre_master_secret and the client and server's random values.
5. The client sends a MAC of all the handshake messages to the server.
6. The server sends a MAC of all the handshake messages to the client.

So, what has this process accomplished? Remember that our two goals were, first, to agree on a set of algorithms and, second, to establish a set of cryptographic keys. Steps 1 and 2 accomplish the first goal. The client gets an opportunity to tell the server which algorithms it supports and the server chooses an algorithm. When the client receives the message the server sent in step 2, it also knows the algorithm, so both sides now know what algorithm to use.

The second goal, establishing a set of cryptographic keys, is accomplished by steps 2 and 3. In step 2 the server provides the client with its certificate, which allows the client to transmit a secret to the server. After step 3, the client and the server both share the pre_master_secret. The client has the pre_master_secret because it generated it, and the server has the pre_master_secret because it decrypted it.

Note that step 3 is the key step in the whole handshake. All the data that is going to be protected depends on the security of the pre_master_secret. What's going on is very simple: the client uses the server's public key (extracted from the certificate) to encrypt a shared key, and the server uses its private key to decrypt the shared key. The rest of the handshake is mainly devoted to ensuring that this exchange can happen safely. In step 4 the client and the server then separately use the same *key derivation function* (KDF) (see Section 3.11) to generate the master_secret. The master_secret is used to generate the cryptographic keys, again using the KDF.

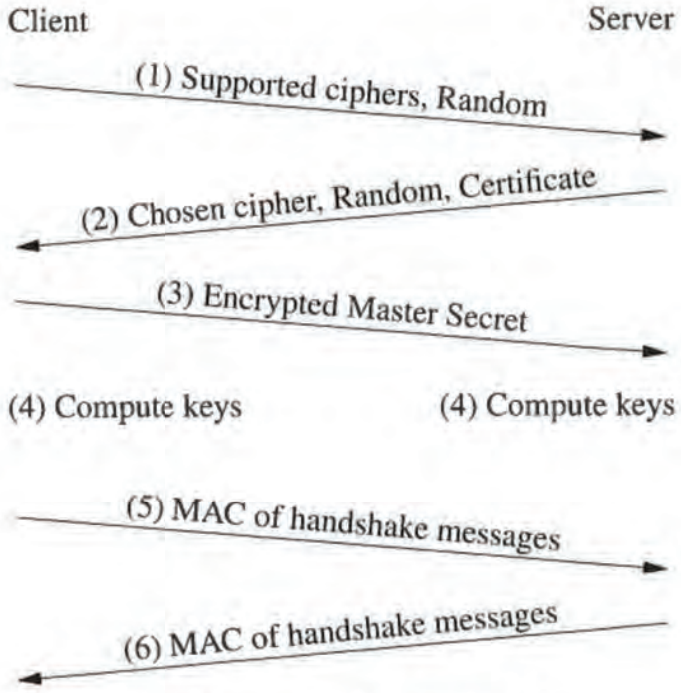


Figure 3.1 Overview of the SSL handshake

Steps 5 and 6 serve to protect the handshake itself from tampering. Imagine an attacker who wished to control the algorithms used by the client and the server. It's quite common for the client to offer a range of algorithms, some weak and some strong, so that it can communicate with servers that only support weak algorithms. The attacker could delete all the strong algorithms from the client's offer in step 1 and thus force the server to choose a weak algorithm. The MAC exchange in steps 5 and 6 stops this, because the client's MAC will be computed over the original messages and the server's MAC will be computed over the messages the attacker modified, and they won't match when checked. Because the client- and server-provided random numbers are inputs to the key generation process, the handshake is secure from replay attack. These messages are the first messages encrypted under the newly encrypted algorithms and keys.

So, at the end of this process, the client and the server have agreed on the cryptographic algorithms to use and have a set of keys to use with those algorithms. More important, they can be sure that an attacker hasn't interfered with the handshake and that therefore the negotiation reflects their true preferences.

Handshake Messages

Each step of the description we've just given is accomplished by one or more handshake messages. We'll briefly describe here which messages map to which steps and then

cover each individual message in detail later in this chapter. Figure 3.2 shows the individual messages.

- Step 1 corresponds to a single SSL handshake message, ClientHello.
- Step 2 corresponds to a series of SSL handshake messages. The first message the server sends is the ServerHello, which contains its algorithm preferences. Next it sends its certificate in the Certificate message. Finally, it sends the ServerHelloDone message, which indicates that this phase of the handshake is done. The reason that the ServerHelloDone is needed is that some of the more complicated handshake variants involve other messages being sent after the Certificate message. When the client receives the ServerHelloDone message, it knows that no such other messages will be arriving and so it can proceed with its part of the handshake.
- Step 3 corresponds to the ClientKeyExchange message.
- Steps 5 and 6 correspond to the Finished message. The Finished message is the first message that's protected using the just-negotiated algorithms. To protect the handshake from tampering, the content of the message is a MAC of *all* the previous handshake messages. However, since the Finished message is protected using the negotiated algorithms, the message itself is also MACed with the newly negotiated MAC keys.

  Note: This diagram omits two messages, the ChangeCipherSpecs. We'll cover them later, but they're not important at this stage in the discussion.

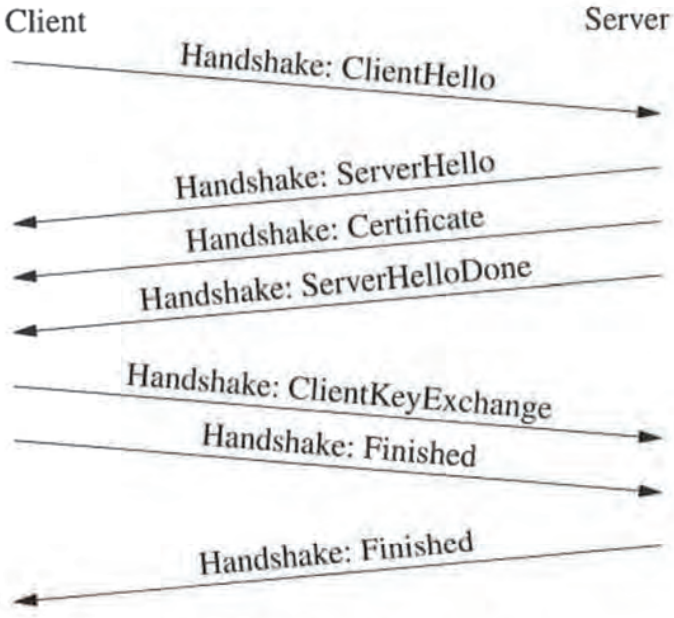


Figure 3.2 SSL handshake messages

3.4 SSL Record Protocol

All we've managed to do so far is to authenticate the server and share some keying material. Remember that the point of having SSL in the first place is to be able to exchange encrypted and authenticated *data*. The purpose of the handshake is to set up the shared state required to make sending and receiving protected data possible. In SSL the actual data transfer is accomplished using the *SSL Record Protocol*.

The SSL Record Protocol works by breaking up the data stream to be transmitted into a series of *fragments*, each of which is independently protected and transmitted. On the receiving end, each record may be independently decrypted and verified. This approach allows data to be transferred from one side of the connection to the other as soon as it is ready and processed as soon as it is received.

Before a fragment can be transmitted, it must be protected against attack. To provide integrity protection, a MAC is computed over the data. The MAC will be transmitted along with the fragment and must be verified by the receiving implementation. The MAC is then appended to the fragment and the concatenated data and MAC are encrypted to form the *encrypted payload*. Finally, a header is attached to the payload. The concatenated header and encrypted payload are referred to as a *record*. Records are what is actually transmitted. Figure 3.3 shows the transmission process.

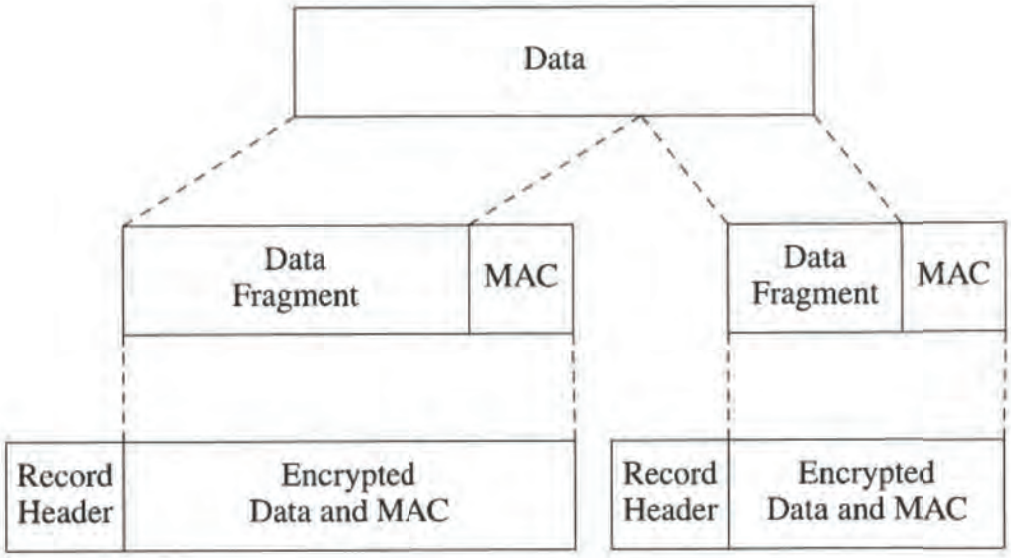


Figure 3.3 SSL data fragmentation and protection

Record Header

The job of the record header is to provide information that is necessary for the receiving implementation to interpret the record. In practice, this means three pieces of information: the *content type*, the length, and the SSL version. The length field allows the receiver to know how many more bytes to read off the wire before it can process the

message. The version number is simply a redundant check to ensure that each side agrees on the version.

The content type field identifies the type of the message. As we discussed in Chapter 1, it's convenient to be able to send other kinds of traffic over the same protected channel. In particular, we'd like to be able to send protected messages to indicate errors and connection closure termination. The content type field allows implementations to distinguish this sort of management traffic from data that is intended for the higher-level application.

Content Types

SSL supports four content types: application_data, alert, handshake, and change_cipher_spec. All data sent and received by software that uses SSL is sent as application_data. The other three content types are used for management traffic, such as performing the handshake and signaling errors.

The alert content type is primarily used for signalling various types of error. Most alerts signal things that went wrong in the handshake, but some indicate errors that occurred when trying to decrypt or authenticate records. The other use of alert messages is to signal that the connection is about to close. This is necessary to prevent the truncation attack discussed in Section 1.8.

The handshake content type is used (unsurprisingly) to carry handshake messages. Even the initial handshake messages that form the connection are carried by the record layer as records of type handshake. Because no cryptographic keys have been established, these initial messages aren't encrypted or authenticated, but otherwise the processing is the same. It is possible to initiate a new handshake over a preexisting connection (see Section 4.5) in which case the new handshake records are encrypted and authenticated just as any other data would be.

The change_cipher_spec message has a special purpose. It indicates a change in the encryption and authentication of records. Once a handshake has negotiated a new set of keys, the change_cipher_spec record is sent to indicate that those new keys will now be used. This is covered in more detail in Section 3.10.

3.5 Putting the Pieces Together

As we've seen, SSL is a *layered* protocol consisting of a record layer and a number of different message types which are carried on top of the record layer. In turn, the record layer is carried on top of some reliable transport protocol such as TCP. Figure 3.4 shows the structure of the protocol.

The rest of this chapter is a detailed look at a particular SSL connection. It covers similar territory to the first part of the chapter, but at a much deeper level of detail. The material covered here is primarily the contents of the specific handshake messages. Thus, if you're interested in implementing SSL, analyzing the protocol, or designing an SSL variant, you should read on. Otherwise, you can safely skip to Chapter 4 where we cover the rest of the handshake modes. First, we discuss the general features of the

Ex.1024
Askeladden, L.L.C. / Page 592 of 788

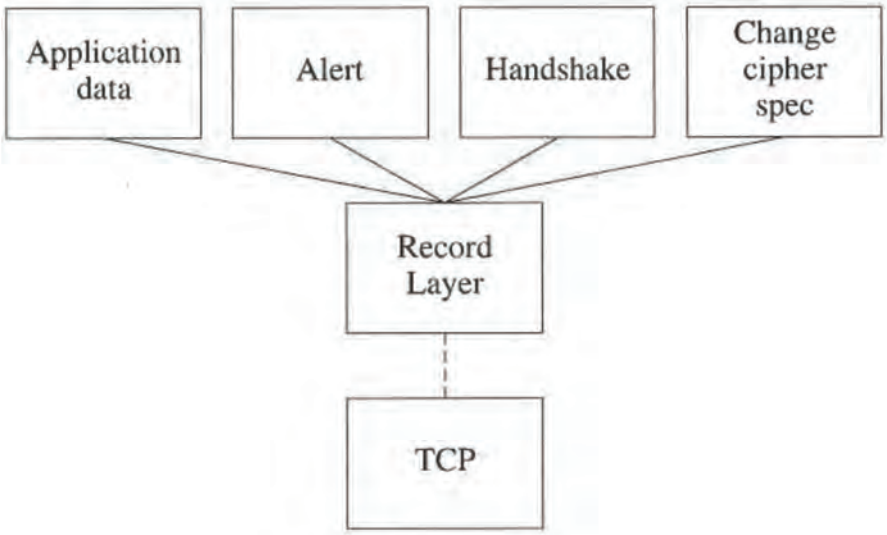


Figure 3.4 SSL protocol structure

connection and then move on to discussing the individual messages in detail.

As before, we'll cover the handshake first, describing how algorithms are negotiated by means of cipher suites and how the server's key is used to establish a shared pre_master_secret which can be used to create the cryptographic keys for the connection. Then we'll cover the cryptographic transformations that the SSL Record Protocol uses to protect the data being communicated.

3.6 A Real Connection

Figure 3.5 shows a ladder diagram for the connection we'll be studying. As with previous such diagrams, messages from the client (`speedy`) to the server (`romeo`) are shown as going from left to right and messages from server to client are shown as going from right to left.

The first message sent is the ClientHello, which contains the client's proposed cryptographic parameters, including the ciphers it's prepared to use. It also contains a random value to be used in key generation. The server responds with three messages: first, it sends the ServerHello which selects a cipher and a compression algorithm. This message contains a random value from the server.

The reason why SSL includes compression is that encrypted data is random for all practical purposes and thus incompressible. Thus, link-level compression such as is found in most modems doesn't work on SSL data. If you want to compress the data before transmission you must do so before encryption. Unfortunately, due to intellectual property considerations, neither SSLv3 nor TLS defined any compression algorithms.

Ex.1024
Askeladden, L.L.C. / Page 593 of 788

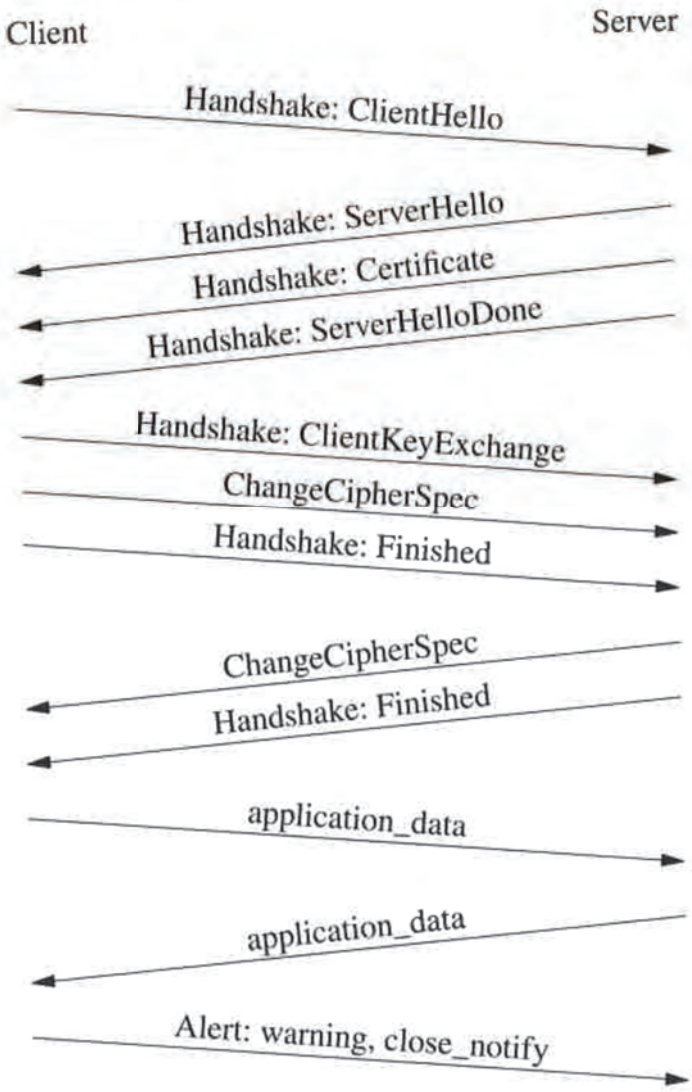


Figure 3.5 Time sequence of SSL messages

Thus, compression is almost never used with SSL. As an exception, OpenSSL does support compression, so if both sides are based on OpenSSL and suitably configured, compression can be used.

Next, the server sends the Certificate message, which contains the server's public key, in this case an RSA key. Finally, it sends the ServerHelloDone, which indicates that no more messages will be sent in this phase of the handshake.

The client now sends a ClientKeyExchange message, which contains a randomly generated key encrypted under the server's RSA key. This is followed by a ChangeCipherSpec message, which indicates that all messages that the client sends afterwards will be encrypted using the just-negotiated cipher. The Finished message contains a

check value for the entire connection so that the server can tell that the ciphers were securely negotiated.

Once the server has received the client's Finished message, it sends its own ChangeCipherSpec and Finished messages and the connection is ready to begin sending application data.

> The SSLv3 specification allows a party to start sending application data as soon as that party's Finished message is *sent*. However, this creates a security risk for client data if the connection has been attacked. If the attacker has changed the handshake to negotiate a weaker cipher suite, the client can't detect this until it receives the server's Finished message. To fix this, in TLS the client must wait until it has received the server's Finished message before it can send its first byte of data. Note that this isn't a security problem if the server chose the most secure of the client's ciphers, but rather than trying to detect this case, it's safest to simply wait for the other server's Finished message.

The next two messages are application data sent by the client and server, respectively. In this case, these were the result of typing data at the client and server programs, but in an actual transaction, this would be when the application protocol data was sent.

Finally, the client shuts down the connection, but first it sends a close_notify alert to indicate that the connection is about to close. The close_notify is followed by a TCP FIN (not shown here). The server responds with its own TCP FIN. The FINs indicate that the host's TCP implementation has closed its side of the connection and will not send any more data.

> Note that the server did not send a close_notify. The SSL specification is a little unclear on exactly whether close_notifys are required, but under perfect circumstances, both the server and client would send close_notifys.

3.7 Some More Connection Details

In order to understand the protocol messages we'll be examining, it's convenient to have a piece of software that can break apart the various messages and display their contents. We've used the `ssldump` program for this purpose.

Figure 3.6 shows an `ssldump` trace of all the messages in a simple SSL connection. We'll be talking about each message in detail in a little bit, but here we present a trace of the whole connection so you can get a feel for the flavor.

Since this is the first `ssldump` trace presented in the book, we'll take a moment to describe the output. The first line simply indicates the setup of the underlying TCP connection that the SSL session will use. In this case it's a connection between `romeo` and `speedy`, two machines on the test network. The name of the host (`romeo`) that initiated the TCP connection (i.e., the host which called `connect()`) is printed first, and the host (`speedy`) that accepted the TCP connection (i.e., the host which called `accept()`) is printed second. In normal SSL operation, the initiating host takes the client role and the accepting host takes the server role.

Each numbered line represents an SSL record. Thus, this connection consists of 12 different records. The first number is the number of the record in the sequence that was processed. This is followed by two timestamps, the first being the time since the creation of the TCP connection and the second being the time since the last record. Both times are represented in seconds and to four decimal places. Next is an indication of which direction the record was sent: `C>S` indicates client to server (in this case `romeo` to `speedy`) and `S>C` indicates server to client (`speedy` to `romeo`). Next is the record type, which corresponds to one of the layers described in the previous section.

```
New TCP connection: speedy(3266) <-> romeo(4433)
1 0.0456 (0.0456) C>S   Handshake
      ClientHello
        Version 3.1
        cipher suites
                  TLS_DHE_RSA_WITH_3DES_EDE_CBC_SHA
                  TLS_DHE_DSS_WITH_3DES_EDE_CBC_SHA
                  TLS_RSA_WITH_3DES_EDE_CBC_SHA
                  TLS_RSA_WITH_IDEA_CBC_SHA
                  TLS_RSA_WITH_RC4_128_SHA
                  TLS_RSA_WITH_RC4_128_MD5
                  TLS_DHE_RSA_WITH_DES_CBC_SHA
                  TLS_DHE_DSS_WITH_DES_CBC_SHA
                  TLS_RSA_WITH_DES_CBC_SHA
                  TLS_DHE_RSA_EXPORT_WITH_DES40_CBC_SHA
                  TLS_DHE_DSS_EXPORT_WITH_DES40_CBC_SHA
                  TLS_RSA_EXPORT_WITH_DES40_CBC_SHA
                  TLS_RSA_EXPORT_WITH_RC2_CBC_40_MD5
                  TLS_RSA_EXPORT_WITH_RC4_40_MD5
        compression methods
                  NULL
2 0.0461 (0.0004) S>C   Handshake
      ServerHello
        session_id[32]=
          74 6d 09 76 1d 2a c9 02 4a a1 a3 4e
          27 5c 18 63 8a d9 4a 59 f9 c3 14 a5
          c4 b3 a4 f6 61 ef f5 cd
        cipherSuite             TLS_RSA_WITH_DES_CBC_SHA
        compressionMethod               NULL
3 0.0461 (0.0000) S>C   Handshake
      Certificate
4 0.0461 (0.0000) S>C   Handshake
      ServerHelloDone
```

```
5 0.2766 (0.2304) C>S   Handshake
      ClientKeyExchange
        EncryptedPreMasterSecret[64]=
          17 4d 00 32 bc b2 af 95 09 0a 45 24
          97 d8 34 dc 73 20 4d 00 91 a5 0d ed
          c3 f0 b4 f5 32 6f 13 cc ea 41 00 5e
          bb 05 f1 b7 e2 c4 fa 1b 40 c1 2a f3
          00 20 83 43 2d 8c 2a 53 3b 33 cc 5f
          0b bc db a2
6 0.2766 (0.0000) C>S   ChangeCipherSpec
7 0.2766 (0.0000) C>S   Handshake
      Finished
8 0.2810 (0.0044) S>C   ChangeCipherSpec
9 0.2810 (0.0000) S>C   Handshake
      Finished
10 1.0560 (0.7749) C>S   application_data
11 6.3681 (5.3121) S>C   application_data
12 7.3495 (0.9813) C>S   Alert
        level         warning
        value         close_notify
Client FIN
Server FIN
```

Figure 3.6 A simple SSL connection

For some record types, `ssldump` performs further decoding, and this is shown on subsequent indented lines. In particular, if the record type is handshake, `ssldump` does a fair amount of decoding. The next line is the handshake type, and subsequent lines contain decodings of some relevant fields. Exactly how much information is conveyed is controlled by command line flags. This particular trace was generated with settings that show only a moderate amount of detail. The intent in this session is to show message flow, so some fields are elided from the trace. Later in the chapter we discuss each message in detail and show all fields in each message.

For example, the first record is a handshake message. The next line shows that it's a ClientHello, sent from `romeo` to `speedy`. It offers a number of cipher suites, ranging from ephemeral Diffie-Hellman with 3DES to RSA with RC4-40, but only one compression method, NULL.

Because `ssldump` was written partially in order to generate traces for this book, the traces are more or less direct program output, with the occasional addition of comments in *italics*. The message trace displayed here was generated by using OpenSSL's `s_client` and `s_server` programs, which are simple command-line client and server programs. Encrypted data is rendered in *`fixed-width italic`*.

3.8 SSL Specification Language

It's fairly common practice for protocols to be described in terms of a specification language which describes the various fields of the protocol messages and how they appear on the wire. This is intended to provide an unambiguous and concise description of the protocol. One example of such a language is ASN.1.

Both TLS and SSLv3 use the same specification language to describe their messages. Although the syntax is vaguely reminiscent of C type definitions, the language was invented for SSLv3 and the only definition is in the specifications.

> The TLS specification describes the syntax as "somewhat casually defined," and although most of the specification is obvious, new constructs are introduced several times without much introduction. At one point, the author attempted to write a grammar-driven parser (using YACC) for the language, with the aim of mechanically generating a decoder, but abandoned the project in frustration.

Basic types

The specification uses five basic types, opaque, uint8, uint16, uint24, and uint32. opaque is simply a single uninterpreted byte. uint8, uint16, uint24, and uint32 are unsigned 8-, 16-, 24-, and 32-bit integers respectively and are represented on the wire by sequences of 1, 2, 3, or 4 bytes. All numbers are represented in "network byte order"—high byte first—so the number 1 is represented as a uint32 by the hex bytes 00 00 00 01. The constructs /* and */ bracket comments just as in C.

Vectors

A vector is simply a sequence of elements of a given type. Vectors come in two flavors, fixed-length or variable-length. Fixed-length vectors are represented using [] and variable-length vectors are represented using <>. It is important to note that all lengths are in bytes, not in number of elements. As a consequence, the declaration uint16 foo[4] refers to two 16-bit integers, not four. This allows the decoder to be layered; it can treat structures as opaque strings (since it knows their length) and pass them on to another layer for parsing.

Variable-length vectors can be represented either by specifying both an upper and lower length bound or just an upper length bound. For example:

```
opaque stuff<1..20>  /* A string of between 1 and 20 bytes */
uint32 numbers<16>   /* Up to 4 32 bit integers */
```

Fixed-length vectors are encoded on the wire simply by concatenating their elements end-to-end with the first element coming first. Variable-length vectors are encoded in the same way, except that a length field is prepended. A length field is an integer type just large enough to encode the upper bound length of the vector. For instance, opaque vec<300> would require a uint16 length field.

Enumerateds

An enumerated type is simply a field that can assume only a specific series of values. Each value is named. For example:

```
enum { red(1), blue(2), green(3) } colors;
```

specifies a type called colors which can take on the values red, blue, and green, which are represented on the wire by the integers 1, 2, and 3. When an enum is encoded on the wire, it is represented by an integral type just large enough to hold its largest value. So, colors would be represented by a uint8, one byte.

It is possible to specify an explicit maximum size for an enum, by including an unnamed value:

```
enum { warning(1), fatal(2), (255) } AlertLevel;
```

As a special case, it is possible to define an enum with no values at all. This is used occasionally in the specification to refer to state values that are internal to an implementation. Obviously, such an enum can never be encoded on the wire, because there are no definitions for how to encode the values.

```
enum { high, low } security;
```

Structures

Constructed types may be made by using the struct definition, which is very similar to C `struct` syntax:

```
struct {
  type1 field1,
  type2 field2,
  type3 field3,
  ...
} name;
```

structs are encoded on the wire simply by concatenating the encodings of the various fields, in the order that they appear in the struct definition. struct definitions can also be nested. The name field of a struct may be omitted if it is directly included in another struct.

Variants

Structures may be defined with variant contents which change depending upon some external information. This is done with a construct that looks superficially like C's

`switch` statement, but is really more like a C `union` or an ASN.1 `CHOICE`.

```
select (type) {
    case value1: Type1
    case value2: Type2
    ...
} name;
```

The type that controls the contents of a select is always an enum. Here's an example:

```
struct {
   select (KeyExchangeAlgorithm) {
     case rsa: EncryptedPreMasterSecret;
     case diffie_hellman: DiffieHellmanClientPublicValue;
   } exchange_keys;
} ClientKeyExchange;

enum { rsa, diffie_hellman } KeyExchangeAlgorithm;
```

This switches the definition of the ClientKeyExchange struct based on the value of KeyExchangeAlgorithm. If KeyExchangeAlgorithm is rsa, then the value is of type EncryptedPreMasterSecret. If it is of diffie_hellman then the value is of type DiffieHellmanClientPublicValue. Note that the value of KeyExchangeAlgorithm is not represented anywhere in the message, but is merely part of the system context. Some variants, however, refer to other fields in the message.

3.9 Handshake Message Structure

Each SSL handshake message consists of a simple header followed by a body which is dependent on the message type. Figure 3.7 shows the definition for the message:

The header is 4 bytes long and consists of a 1-byte type field followed by 3 length bytes. The length field refers to the length of the rest of the handshake message (not including the type and length fields). The contents of the rest of the message are completely dependent on the msg_type field. The length field is not strictly necessary because the end of the handshake message can be determined from the components. However, having an explicit length field makes it much easier to implement the protocol. The implementation can break apart sequential handshake messages in one layer and then process them in another. Note that the mapping of handshake messages to SSL records is not fixed. Multiple handshake messages may appear in a single record. Theoretically, a single handshake message might span multiple records, but in practice this does not occur.

```
struct {
   HandshakeType msg_type;
   uint24 length;
   select (HandshakeType) {
      case hello_request:      HelloRequest;
      case client_hello:       ClientHello;
      case server_hello:       ServerHello;
      case certificate:        Certificate;
      case server_key_exchange: ServerKeyExchange;
      case certificate_request: CertificateRequest;
      case server_hello_done:   ServerHelloDone;
      case certificate_verify:  CertificateVerify;
      case client_key_exchange: ClientKeyExchange;
      case finished:           Finished;
   } body;
} Handshake;

enum {
   hello_request(0), client_hello(1), server_hello(2),
   certificate(11), server_key_exchange (12),
   certificate_request(13), server_hello_done(14),
   certificate_verify(15), client_key_exchange(16),
   finished(20), (255)
} HandshakeType;
```

Figure 3.7 SSL Handshake Message

3.10 Handshake Messages

The first step in creating an SSL connection is the handshake, which establishes the algorithms and keying material for the rest of the connection. This section covers the SSL handshake we showed earlier in detail. For each handshake message, we show the relevant piece of specification language and the expanded trace of that handshake message. Then, we explain the meaning and values of each field.

There are two ways to read this section. If you want to really understand what's going on, then read each piece of specification and match it up with the `ssldump` trace, using the accompanying text as a guide. If all you want to do is get the general idea, then just read the text, referring to the structure definitions and the protocol traces as a guide.

ClientHello

```
struct {
   ProtocolVersion client_version;
   Random random;
   SessionID session_id;
   CipherSuite cipher_suites<2..2^16-1>;
   CompressionMethod compression_methods<1..2^8-1>;
} ClientHello;

struct {
   uint8 major;
   uint8 minor;
} ProtocolVersion;

struct {
   uint32 gmt_unix_time;
   opaque random_bytes[28];
} Random;

opaque SessionID<0..32>;
uint8 CipherSuite[2];
enum { null(0), (255) } CompressionMethod;
```

```
ClientHello
  Version 3.1
  random[32]=
    38 f3 cb de 80 4c b4 79 0a 07 9f b3
    51 ba b8 62 69 e3 8f bf ce c7 ff 25
    3c 3b 84 16 38 b2 5e f7
  cipher suites
            TLS_DHE_RSA_WITH_3DES_EDE_CBC_SHA
            TLS_DHE_DSS_WITH_3DES_EDE_CBC_SHA
            TLS_RSA_WITH_3DES_EDE_CBC_SHA
            TLS_RSA_WITH_IDEA_CBC_SHA
            TLS_RSA_WITH_RC4_128_SHA
            TLS_RSA_WITH_RC4_128_MD5
            TLS_DHE_RSA_WITH_DES_CBC_SHA
            TLS_DHE_DSS_WITH_DES_CBC_SHA
            TLS_RSA_WITH_DES_CBC_SHA
            TLS_DHE_RSA_EXPORT_WITH_DES40_CBC_SHA
            TLS_DHE_DSS_EXPORT_WITH_DES40_CBC_SHA
            TLS_RSA_EXPORT_WITH_DES40_CBC_SHA
            TLS_RSA_EXPORT_WITH_RC2_CBC_40_MD5
            TLS_RSA_EXPORT_WITH_RC4_40_MD5
  compression methods
            NULL
```

The first handshake message sent is always the ClientHello message. The main purpose of this message is for the client to communicate its preferences for connection parameters. After sending the ClientHello, the client waits for the server to send its hello message. In SSLv3 and TLS, the client presents its acceptable parameters and the server chooses the final set. The three negotiable parameters are the version, represented by the client_version field, the cryptographic algorithms, represented by cipher_suites, and the compression algorithm, represented by compression_methods. Figure 3.8 lists the main TLS cipher suites.

The client_version field should contain the highest SSL version number that the client is prepared to speak. For SSLv3, this means major=3 and minor=0. For TLS, major=3 and minor=1. In this trace, the client has indicated that it speaks TLS. In general, it is expected that an implementation speaks all lesser versions. SSL provides no facility for indicating that you do not speak previous versions. This means that if a client knows that (say) version 2.0 is insecure, there's no way to tell the server: use version 3.0 or above. You simply have to try out the server and terminate the connection if the server chooses version 2.

All cryptographic selections for a connection are bundled together into CipherSuites, which are represented by arbitrarily chosen 2-byte constants. The CipherSuite specifies the server authentication algorithm, the key exchange algorithm, the bulk encryption algorithm, and the digest (message integrity) algorithm. The suites are listed in the order of descending client preference. Figure 3.8 shows a list of the TLS cipher suites.

The cipher suites marked with a † are not part of the TLS standard. They were introduced in [Banes1999] in response to the liberalization of U.S. export policy that allowed 1024-bit key agreement and 56-bit symmetric encryption. These cipher suites allow 1024-bit RSA and DH combined with either DES or 56-bit RC4. [Banes1999] also adds support for DSS/DH with RC4-128, which had been omitted in the TLS standard. Although these algorithms are widely implemented, the full liberalization of U.S. export policy has rendered them unnecessary, so it's unclear if they will ever be standardized. Note that because these algorithms are labelled export, they go through the export key derivation procedure described in Section 3.11.

The final negotiable parameter is the compression algorithm. Compression algorithms are represented by 1-byte constants. In actual practice, due mainly to patent considerations, no compression algorithms have ever been defined for SSL, and so the only supported algorithm is the mandatory NULL algorithm, which does not change the data. Some proprietary implementations implement private compression algorithms, as does OpenSSL.

The second parameter in the ClientHello is a 32-byte value, (called random) which is used as one of the inputs to the key generation procedure. The first 4 bytes of the value are the time of day when the message was generated (in seconds since the Unix epoch, 12:00 midnight, January 1, 1970 GMT) and the other 28 bytes are supposed to be randomly generated. Note that the protocol will operate correctly even if the first 4 bytes are not time. Client and server clocks do not have to be synchronized. The purpose of the random data is to ensure that even if the same pre_master_secret is used the generated encryption and MAC keys will be different. This prevents replay attack.

Cipher Suite	Auth	Key Exchange	Encryption	Digest	Number
TLS_RSA_WITH_NULL_MD5	RSA	RSA	NULL	MD5	0x0001
TLS_RSA_WITH_NULL_SHA	RSA	RSA	NULL	SHA	0x0002
TLS_RSA_EXPORT_WITH_RC4_40_MD5	RSA	RSA_EXPORT	RC4_40	MD5	0x0003
TLS_RSA_WITH_RC4_128_MD5	RSA	RSA	RC4_128	MD5	0x0004
TLS_RSA_WITH_RC4_128_SHA	RSA	RSA	RC4_128	SHA	0x0005
TLS_RSA_EXPORT_WITH_RC2_CBC_40_MD5	RSA	RSA_EXPORT	RC2_40_CBC	MD5	0x0006
TLS_RSA_WITH_IDEA_CBC_SHA	RSA	RSA	IDEA_CBC	SHA	0x0007
TLS_RSA_EXPORT_WITH_DES40_CBC_SHA	RSA	RSA_EXPORT	DES40_CBC	SHA	0x0008
TLS_RSA_WITH_DES_CBC_SHA	RSA	RSA	DES_CBC	SHA	0x0009
TLS_RSA_WITH_3DES_EDE_CBC_SHA	RSA	RSA	3DES_EDE_CBC	SHA	0x000A
TLS_DH_DSS_EXPORT_WITH_DES40_CBC_SHA	RSA	DH_DSS_EXPORT	DES_40_CBC	SHA	0x000B
TLS_DH_DSS_WITH_DES_CBC_SHA	DSS	DH	DES_CBC	SHA	0x000C
TLS_DH_DSS_WITH_3DES_EDE_CBC_SHA	DSS	DH	3DES_EDE_CBC	SHA	0x000D
TLS_DH_RSA_EXPORT_WITH_DES40_CBC_SHA	RSA	DH_EXPORT	DES_40_CBC	SHA	0x000E
TLS_DH_RSA_WITH_DES_CBC_SHA	RSA	DH	DES_CBC	SHA	0x000F
TLS_DH_RSA_WITH_3DES_EDE_CBC_SHA	RSA	DH	3DES_EDE_CBC	SHA	0x0010
TLS_DHE_DSS_EXPORT_WITH_DES40_CBC_SHA	DSS	DHE_EXPORT	DES_40_CBC	SHA	0x0011
TLS_DHE_DSS_WITH_DES_CBC_SHA	DSS	DHE	DES_CBC	SHA	0x0012
TLS_DHE_DSS_WITH_3DES_EDE_CBC_SHA	DSS	DHE	3DES_EDE_CBC	SHA	0x0013
TLS_DHE_RSA_EXPORT_WITH_DES40_CBC_SHA	RSA	DHE_EXPORT	DES_40_CBC	SHA	0x0014
TLS_DHE_RSA_WITH_DES_CBC_SHA	RSA	DHE	DES_CBC	SHA	0x0015
TLS_DHE_RSA_WITH_3DES_EDE_CBC_SHA	RSA	DHE	3DES_EDE_CBC	SHA	0x0016
TLS_DH_anon_EXPORT_WITH_RC4_40_MD5	-	DH_EXPORT	RC4_40	MD5	0x0017
TLS_DH_anon_WITH_RC4_128_MD5	-	DH	RC4_128	MD5	0x0018
TLS_DH_anon_EXPORT_WITH_DES40_CBC_SHA	-	DH	DES_40_CBC	SHA	0x0019
TLS_DH_anon_WITH_DES_CBC_SHA	-	DH	DES_CBC	SHA	0x001A
TLS_DH_anon_WITH_3DES_EDE_CBC_SHA	-	DH	3DES_EDE_CBC	SHA	0x001B
TLS_RSA_EXPORT1024_WITH_DES_CBC_SHA †	RSA	RSA	DES_CBC	SHA	0x0062
TLS_DHE_DSS_EXPORT1024_WITH_DES_CBC_SHA †	RSA	RSA	DES_CBC	SHA	0x0063
TLS_RSA_EXPORT1024_WITH_RC4_56_SHA †	RSA	RSA	RC4_56	SHA	0x0064
TLS_DHE_DSS_EXPORT1024_WITH_RC4_56_SHA †	RSA	RSA	RC4_56	SHA	0x0065
TLS_DHE_DSS_WITH_RC4_128_SHA †	RSA	RSA	RC4_56	SHA	0x0066

Figure 3.8 Available TLS cipher suites

The ClientHello also contains a session_id parameter. The session ID is a byte string of up to 32 bytes which can be used by the client to indicate that it wishes to reuse the cryptographic keying material from a previous connection rather than generate new

material. This is typically much faster because public key operations are computationally expensive. In this trace, the client and server had no previous session to resume and so the session_id field was zero length. We'll see an example of a resumed session in Chapter 4.

ServerHello

```
struct {
  ProtocolVersion server_version;
  Random random;
  SessionID session_id;
  CipherSuite cipher_suite;
  CompressionMethod compression_method;
} ServerHello;
```

```
ServerHello
  SSL version 3.1
  random[32]=
    38 f3 cb d1 33 63 1c c7 2e 8c 56 43
    9e fb 20 70 cc 4b 16 06 4d 5a 8b 15
    e3 9f 0d 47 39 16 5f 5c
  session_id[32]=
    74 6d 09 76 1d 2a c9 02 4a a1 a3 4e
    27 5c 18 63 8a d9 4a 59 f9 c3 14 a5
    c4 b3 a4 f6 61 ef f5 cd
  cipherSuite           TLS_RSA_WITH_DES_CBC_SHA
  compressionMethod           NULL
```

The ServerHello message is used by the server to choose from the various options offered to it by the client. The server_version, cipher_suite, and compression_method fields in this message are the version, ciphers, and compression algorithms that will be used for this SSL connection.

In this trace, the client and the server both speak TLS, so the chosen version is 3.1. The server has chosen the cipher suite TLS_RSA_WITH_DES_CBC_SHA: RSA key exchange with DES-CBC for message encryption and SHA for message digest. Note that the server is under no obligation to choose the client's preferred cipher suite, even if it is supported. The SSL specification is silent on how the choice is made among the client's supported ciphers. The server is free to defer to the client's preferences or impose its own preferences. Common practice, however, is for the server to choose the client-supported cipher that it prefers most. This is somewhat inconvenient for client implementors because they cannot predict which cipher the server will choose. Among other things, this means that they can't give the user feedback about the likely consequences of making a given connection until the connection process has started.

Finally, the server has chosen the only available compression algorithm, NULL. The server also provides a random value. Together with the client's random value and the pre_master_secret (to be discussed later), it is used to generate the keying material to be used for this connection. This ensures that even if the client is broken and uses the same random value for two handshakes, the final cryptographic keys will be different. This also serves to protect against attempts to replay the handshake, as discussed in Section 1.8.

Typically, the server provides a session_id which the client could later use to resume the session. If a server doesn't want to resume sessions, a zero-length session ID can be provided. In this case, the server has provided one, indicating that it's prepared to resume this session. Exactly how session IDs are generated is left up to the implementation. Although they're often generated purely randomly, it is possible for them to have some structure that makes it easier for the server to look up the session in its cache.

Certificate

```
struct {
   ASN.1Cert certificate_list<1..2^24-1>;
} Certificate;

opaque ASN.1Cert<2^24-1>;
```

```
Certificate
  Subject
    C=AU
    ST=Queensland
    O=CryptSoft Pty Ltd
    CN=Server test cert (512 bit)
  Issuer
    C=AU
    ST=Queensland
    O=CryptSoft Pty Ltd
    CN=Test CA (1024 bit)
  Serial         04
  certificate[493]=
    30 82 01 e9 30 82 01 52 02 01 04 30
    0d 06 09 2a 86 48 86 f7 0d 01 01 04
    05 00 30 5b 31 0b 30 09 06 03 55 04
    06 13 02 41 55 31 13 30 11 06 03 55
    04 08 13 0a 51 75 65 65 6e 73 6c 61
    6e 64 31 1a 30 18 06 03 55 04 0a 13
    11 43 72 79 70 74 53 6f 66 74 20 50
    74 79 20 4c 74 64 31 1b 30 19 06 03
    55 04 03 13 12 54 65 73 74 20 43 41
    20 28 31 30 32 34 20 62 69 74 29 30
```

```
1e 17 0d 39 38 30 36 32 39 32 33 35
32 34 30 5a 17 0d 30 30 30 36 32 38
32 33 35 32 34 30 5a 30 63 31 0b 30
09 06 03 55 04 06 13 02 41 55 31 13
30 11 06 03 55 04 08 13 0a 51 75 65
65 6e 73 6c 61 6e 64 31 1a 30 18 06
03 55 04 0a 13 11 43 72 79 70 74 53
6f 66 74 20 50 74 79 20 4c 74 64 31
23 30 21 06 03 55 04 03 13 1a 53 65
72 76 65 72 20 74 65 73 74 20 63 65
72 74 20 28 35 31 32 20 62 69 74 29
30 5c 30 0d 06 09 2a 86 48 86 f7 0d
01 01 01 05 00 03 4b 00 30 48 02 41
00 9f b3 c3 84 27 95 ff 12 31 52 0f
15 ef 46 11 c4 ad 80 e6 36 5b 0f dd
80 d7 61 8d e0 fc 72 45 09 34 fe 55
66 45 43 4c 68 97 6a fe a8 a0 a5 df
5f 78 ff ee d7 64 b8 3f 04 cb 6f ff
2a fe fe b9 ed 02 03 01 00 01 30 0d
06 09 2a 86 48 86 f7 0d 01 01 04 05
00 03 81 81 00 95 be f7 e4 19 27 b6
18 78 03 15 f9 8e b9 ae 08 b2 36 fd
25 58 48 99 63 00 4a 23 82 96 46 65
30 44 83 26 3b 2c ce 0f fa f9 df d6
fb c4 eb 6c e6 e1 6b 3a 65 f7 91 62
bd 70 55 b9 c6 e3 f5 db 9d 87 b0 0e
21 9b b5 87 53 00 3e 5c a4 9d cf 54
77 cd 7a bf 3d c5 7a 30 78 aa a5 28
69 78 e7 96 4a c8 80 46 eb fe e9 fb
8d 24 bc e9 63 9e d2 14 61 c0 79 09
15 41 9c 3d 97 fd 34 3d b6 12 d7 3e
01
```

The Certificate message is simply a sequence of X.509 certificates. The certificates are presented in order, with the first certificate being the certificate belonging to the server and the next certificate (if there is one) containing the key that certifies the server's certificate, and so on. In all common SSL and TLS cipher suites, the server must send its certificate (there are several rarely used cipher suites where the server remains anonymous). The key in the certificate is then either used to encrypt the pre_master_secret or to verify the ServerKeyExchange, depending on which cipher suite was chosen.

In this case, the server has sent only one certificate, for *Server Test Cert (512 bit)*. It's assumed that the client already has the CA certificate for *Test CA (1024 Bit)*. What's presented here is a summary of the most important certificate information. More detail about the structure of X.509 certificates can be found in Chapter 1.

ServerHelloDone

```
struct { } ServerHelloDone;
```

The ServerHelloDone is an empty message that indicates that the server has sent all the handshake messages it will send in this phase. This is required because there are some optional messages that can appear after the Certificate message. We'll see some examples of that in Chapter 4.

ClientKeyExchange

```
struct {
   select (KeyExchangeAlgorithm) {
      case rsa: EncryptedPreMasterSecret;
      case diffie_hellman: DiffieHellmanClientPublicValue;
   } exchange_keys;
} ClientKeyExchange;

struct {
   ProtocolVersion client_version;
   opaque random[46];
} PreMasterSecret;

struct {
   public-key-encrypted PreMasterSecret pre_master_secret;
} EncryptedPreMasterSecret;

enum { implicit, explicit } PublicValueEncoding;

struct {
   select (PublicValueEncoding) {
      case implicit: struct {};
      case explicit: opaque DH_Yc<1..2^16-1>;
   } dh_public;
} DiffieHellmanClientPublicValue;
```

```
ClientKeyExchange
  EncryptedPreMasterSecret[64] =
    17 4d 00 32 bc b2 af 95 09 0a 45 24
    97 d8 34 dc 73 20 4d 00 91 a5 0d ed
    c3 f0 b4 f5 32 6f 13 cc ea 41 00 5e
    bb 05 f1 b7 e2 c4 fa 1b 40 c1 2a f3
    00 20 83 43 2d 8c 2a 53 3b 33 cc 5f
    0b bc db a2
```

The ClientKeyExchange message provides the client's contribution to the creation of the pre_master_secret. When RSA key exchange is being used, this simply means that the client generates a PreMasterSecret structure and encrypts it under the server's RSA key. The pre_master_secret is a 48-byte quantity consisting of two version number bytes followed by 46 randomly generated bytes. It is critical that these bytes be chosen using a cryptographically secure RNG (see Chapter 5 for information on RNGs.) RFC 1750 [Eastlake1994] provides guidance on random number generation. Wagner and Goldberg's successful attack on Netscape's SSLv2 implementation [Goldberg1996] was based on flaws in Netscape's RNG. We provide some guidance on generating strong random numbers in Chapter 5.

> Wagner and Goldberg disassembled the Netscape RNG and observed that the RNG was seeded with the time of day and the process ID. Since both of these numbers are fairly limited in range, it was fairly easy to search the entire space and recover the SSL session key. This process took about an hour at the time and would be faster now.

The public-key-encrypted operator in the EncryptedPreMasterSecret struct means that the pre_master_secret field is to be encrypted using the recipient's public key. The encrypted key is then represented as a variable length vector:

```
opaque encrypted_data<0..2^16-1>
```

The ClientKeyExchange message is one of only two messages that are different between SSLv3 and TLS. This difference is due to a bug in Netscape's SSLv3 implementation that was mimicked by other SSLv3 implementors. When RSA is being used for encryption, the length of the encrypted data is redundant because the length of the EncryptedPreMasterSecret is unambiguously determined both by the length of the ClientKeyExchange message and by the length of the server's RSA key. As a consequence, Netscape's SSLv3 implementation omitted the length bytes when encoding the EncryptedPreMasterSecret, and most other SSLv3 implementations followed their (incorrect) lead. That is to say that most SSLv3 implementations violate the SSLv3 specification. An implementation note in the TLS specifcation points out that most SSLv3 implementations are broken and requires that TLS implementations correctly include the length bytes. The trace we are looking at is from an implementation in TLS mode, and therefore the length bytes are included.

ChangeCipherSpec

```
struct {
    ChangeCipherSpecType type;
} ChangeCipherSpec;

enum { change_cipher_spec(1), (255) } ChangeCipherSpecType;
```

The ChangeCipherSpec message indicates that the sending implementation has switched to the newly negotiated algorithms and keying material and that future messages will be protected using those algorithms. The ChangeCipherSpec message is unique in that it is not properly part of the handshake but rather has its own content type. This is described as a performance improvement in the specification, which says:

> Note: To help avoid pipeline stalls, ChangeCipherSpec is an independent TLS Protocol content type, and is not actually a TLS handshake message.

However, only one implementation known to the author (Netscape's) makes use of this feature in any interesting way. Netscape uses this feature to make sure that the ChangeCipherSpec message is transmitted separately from the Finished message. Because the ChangeCipherSpec message must not be encrypted and the Finished message must be encrypted, they cannot be transmitted in the same record. Having them in different layers ensures this.

This trick is useful only if your implementation attempts to send multiple handshake messages in the same record. As we'll see in Chapter 6, it's sometimes a performance win to send multiple handshake messages in the same TCP segment, and sending them in the same record is one way of doing this. However, many implementations instead opt to transmit multiple records in the same TCP segment, which has much the same effect. For such implementations, this protocol feature is an inconvenience which complicates the handshake state machine.

A typical implementation is to have the handshake state machine read only handshake type traffic. However, it also needs to be aware that the ChangeCipherSpec has been received, and this information must be carried out of band from the record layer processing. If the ChangeCipherSpec message were a handshake message, the handshake layer could simply read it off the network. Instead, the record layer code needs to have special support for recognizing the ChangeCipherSpec message and either telling the handshake layer or throwing an error depending on whether the handshake is in the right state.

The ChangeCipherSpec message consists only of a single type byte, which is mainly a placeholder since it can currently have only a single value of 1. No expansion of the use of this message has been publicly proposed.

Finished

```
SSLv3:
struct {
  opaque md5_hash[16];
  opaque sha_hash[20];
} Finished;
```

```
TLS:
struct {
   opaque verify_data[12];
} Finished;

Finished
   verify_data[12]=
      03 3b 69 b7 26 e3 0e 23 fc 03 79 27
```

The Finished message is the first message encrypted with the new cryptographic parameters. It allows an implementation to verify that none of the handshake messages have been tampered with by an attacker.

Although the details of the SSLv3 and TLS Finished messages are different, the general concept is similar. Each side sends a digest of the negotiated master secret and the concatenated handshake messages to the other, which compares it to the same result computed locally. If they differ, the handshake has been tampered with.

```
enum { client(0x434C4E54), server(0x53525652) } Sender;

md5_hash = MD5(master_secret + pad2 +
          MD5(handshake_messages +
             Sender + master_secret + pad1));
sha_hash = SHA-1(master_secret +  pad2 +
          SHA-1(handshake_messages +
             Sender + master_secret + pad1));
```

Figure 3.9 SSLv3 Finished computations

The SSLv3 Finished message is modeled after the popular cryptographic message authentication function HMAC [Krawczyk1997], which uses nested hashes to create a construction which provably has certain security properties. Figure 3.9 shows the SSLv3 computation. The plus sign indicates concatenation, so the input to the first invocation of MD5 (the inner one) is the string created by concatenating all the handshake messages, the Sender constant, the master secret, and some pad bytes (pad1 is a string of the byte 0x36 repeated 48 times). The input to the second invocation of MD5 is the string created by concatenating the master secret, some pad bytes (pad2 is the a string of the byte 0x5c repeated 48 times), and the output of the first MD5 computation. The SHA-1 computation is identical except that the pad bytes are repeated 40 times instead of 48. The purpose of the Sender constant is to ensure that the client and server produce different Finished messages. This prevents an attacker from reflecting a Finished message back to its sender.

Although the SSLv3 Finished message is based on HMAC, it was modified to be usable for SSL. In particular, because the master secret is available only midway through the handshake, the handshake messages are the first input to the inner digest, whereas in HMAC, the key is always the first input. Consequently, some of the security

properties of HMAC may not apply. Comments from the HMAC designers were the primary reason why the computation was changed in TLS.

```
verify_data =
  PRF(master_secret, finished_label, MD5(handshake_messages) +
    SHA-1(handshake_messages)) [0..11];
```

Figure 3.10 TLS Finished computation

TLS makes wide use of an HMAC-based *pseudo-random function* (PRF) for a number of cryptographic computations, including the Finished message (see Figure 3.10). (We'll see more of the PRF in the section on key derivation.) This PRF is used to create the verify_data value. The inputs are the master secret, an ASCII string for the finished label ("`client`" or "`server`") and an MD5 and SHA-1 digest of the handshake messages. The [0..11] indicates that the first 12 bytes of PRF output are used to create the verify_data.

Note that both MD5 and SHA-1 are used in the creation of the Finished message. The rationale for this is to provide additional security against weaknesses in the digest algorithm. This way, an attacker must break both MD5 and SHA-1 in order to forge a Finished message.

Programming Note: Processing Finished

Note that although the Finished message must include the digest of all the handshake messages so far, it does not include its own digest. However, the second Finished message must include a digest of the first. This creates an interesting programming problem for implementors. A naive implementation might write its handshake handler something like the code in Figure 3.11.

Unfortunately, when the recipient goes to compute the Finished value, the digest will include the just-received Finished message, and so the values won't match. However, the server can't just stop digesting because it needs to include the digest of the first Finished in its response to the client's Finished. Thus, it must either maintain two sets of digests or make a copy of the digest objects before processing the Finished message, as shown in Figure 3.12.

3.11 Key Derivation

Once the pre_master_secret has been exchanged, each implementation needs to expand it into the individual cryptographic keys that will be used for encryption, authentication, etc. This expansion is accomplished using a key derivation function. The SSLv3 and TLS key derivation functions are similar, but differ in the precise cryptographic transformations they use. We'll describe the TLS key derivation function with an aside at the

```
receive_handshake_message(context,message) {
  digest_handshake_message(context,message);
  switch(message.type){
    case ClientHello:
      handle_client_hello(context,message);
      break;
      ...
    case Finished:
      handle_finished(context,message);
      break;
  }
```

Figure 3.11 Naive Finished handling

```
receive_handshake_message(context,message) {
  if(message.type==Finished){
    digests=copy_digests(context);
  }
  digest_handshake_message(context,message);

  switch(message.type){
    case ClientHello:
      handle_client_hello(context,message);
      break;
      ...
    case Finished:
      handle_finished(context,message,digests);
      break;
  }
```

Figure 3.12 Correct Finished handling

end describing the differences from the SSLv3 process. You can refer to Figure 3.13 for a graphical representation of the process.

The PRF

The entire TLS key derivation process uses PRF as the key derivation function. The point of this is to expand the pre_master_secret into another secret called master_secret, and then expand the master_secret into the cryptographic keys that we

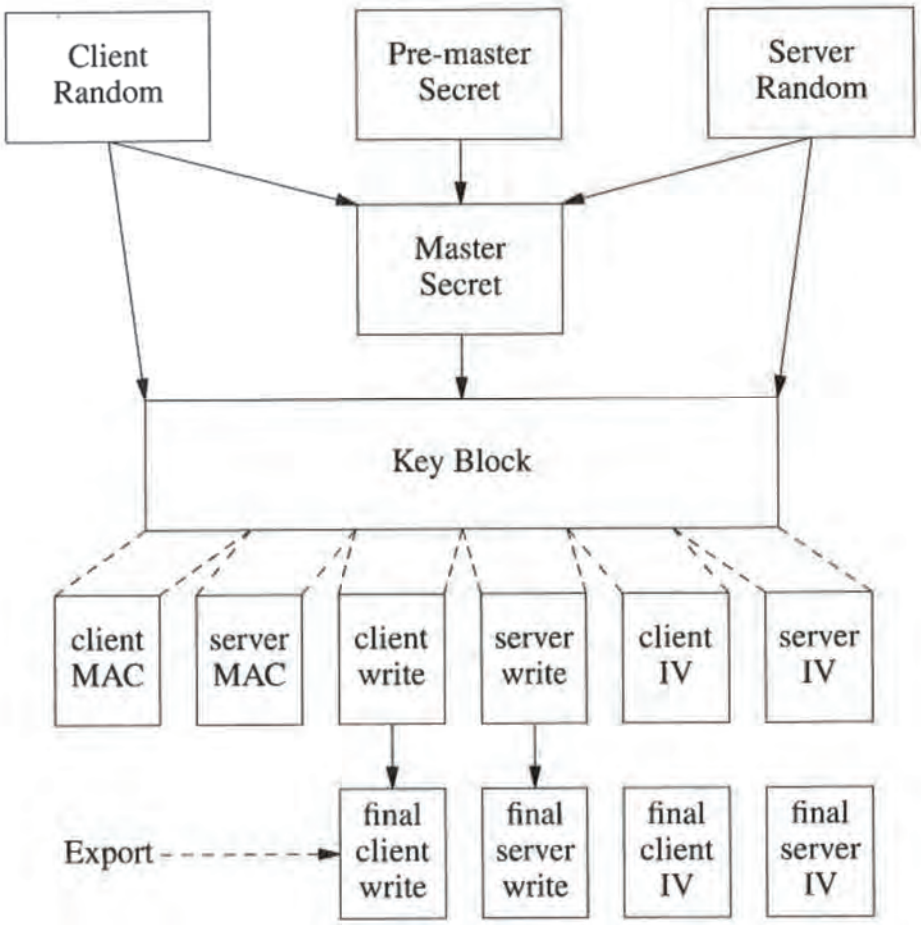


Figure 3.13 Key derivation

need for the various encryption and MAC algorithms. As with the key derivation function in Section 1.8, the client and the server each independently use the same key derivation function to generate the keys from the shared master_secret. The PRF takes three arguments, a secret (presumably random), a fixed ASCII string (the label), and a seed (presumably random, but public), as in PRF(secret, label, seed). The purpose of the label is to allow you to use the PRF to generate different keys from the same secret material simply by using different labels for each key. The label is simply digested as the corresponding ASCII string.

The PRF generates an arbitrary length string of pseudo-random bytes, which are referred to by using the TLS vector notation. So, PRF()[0..9] refers to the first 10 bytes of the PRF output. The PRF computation is fairly complicated and not that interesting except to implementors. We cover it here only for completeness, so feel free to skip over this stuff. It won't be on the exam.

First, the secret part is split into two halves, S1 and S2. S1 is the first half of the secret and S2 is the second half. If the original secret is odd-sized, then the last byte of

S1 and the first byte of S2 are the same.

Each half is used as the secret component in an expansion function P_hash, which is based on HMAC. P_hash generates an arbitrary length byte string by the process shown in Figure 3.14.

```
P_hash(secret, seed) = HMAC_hash(secret, A(1) + seed) +
                       HMAC_hash(secret, A(2) + seed) +
                       HMAC_hash(secret, A(3) + seed) + ...

A() is defined as:
    A(0) = seed
    A(i) = HMAC_hash(secret, A(i-1))

HMAC_hash means HMAC using a given hash algorithm, e.g.,
HMAC_MD5 means HMAC using MD5.
```

Figure 3.14 P_hash function

Now that we've got these functions defined, we construct PRF by XORing a P_MD5 and a P_SHA-1:

```
PRF(secret, label, seed) = P_MD5(S1, label + seed) ⊕
                           P_SHA-1(S2, label + seed);
```

Because MD5 digests are 16 bytes long and SHA-1 digests are 20 bytes long, P_MD5 and P_SHA-1 don't line up. So, if you want to produce an 80-byte output, you have to run P_MD5 five times and P_SHA-1 four times.

The three non-fixed inputs to the key derivation function are the pre_master_secret, the client_random, and the server_random. The pre_master_secret is simply a secret byte string. When RSA key exchange is being used, it's a 48-byte string generated by the client. It's encrypted under the server's public key. The pre_master_secret is the only source of secret information in this process. The other two inputs, the client_random and server_random, are the random values from the ClientHello and ServerHello messages.

The first step in the key derivation is to transform the pre_master_secret into the master_secret. This is done by applying the pseudo-random function PRF to the pre_master_secret, client_random, and server_random:

```
master_secret = PRF(pre_master_secret, "master secret",
                    client_random + server_random) [0..47];
```

We now use the PRF with the master_secret to give us enough keying material for all the algorithms we'll need. Depending on the cipher suite being used, we might need up to six values, an encryption key, a MAC key, and an initializaton vector (IV) for both

client and server. First, we add up the lengths of all the values we need and produce that many bytes using the PRF. For instance, if we're using DES and MD5, we need 8 bytes for each encryption key, 16 bytes for each HMAC-MD5 key, and 8 bytes for each IV, for a total of 48 bytes:

```
key_block = PRF(master_secret, "key expansion",
          server_random + client_random)
```

Note that the order of the client_random and server_random is different here from the master_secret computation.

Once we have the key block, we simply slice and dice it one key at a time, in the following order: client_write_MAC_secret, server_write_MAC_secret, client_write_key, server_write_key, client_write_IV, server_write_IV.

Exportable Algorithms

Due to U.S. export restrictions at the time of the SSLv3 and TLS design, exportable ciphers needed to have their encryption keys shortened so that they could be broken with no more than 2^{40} operations. In order to do this, we first generate a short intermediate and then reexpand it using the PRF. This approach prevents a precomputation attack in which the attacker simply builds a big table for all 2^{40} keys. The server and client random values act as a *salt* preventing this attack. The attacker can still attack the key by trying all of the intermediate keys but the time/memory tradeoff won't work.

> The export situation has changed since the design of SSL/TLS. 40-bit keys are essentially worthless for data security. Attacking a 40-bit key is easily within the range of the most modest commercial attacker. In 1998 a dedicated DES keysearch machine was built [Gilmore1998], showing that 56-bit keys could be compromised by a dedicated attacker. In response to this, the U.S. government decided that cryptographic systems with 56-bit keys would now be exportable. Changes have been proposed to the TLS spec to include 56-bit exportable ciphers. These ciphers also allow 1024-bit RSA keys instead of 512-bit RSA keys. In January 2000 the U.S. government essentially removed all export restrictions. However, there is still a lot of old software that has only exportable modes so it's important to know about them. The Bureau of Export Administration Web site describes them and can be reached at
> `http://www.bxa.doc.gov/Encryption/Default.htm/`.

Also, we're required (again for export reasons) to use a non-secret IV (normally the SSL IV is generated partially from secret data and is therefore secret) so we derive it purely from the public client_randomand server_random values. The first half of the iv_block is used to generate the IV that the client uses to encrypt data, and the second half for the IV that the server uses to encrypt data (see Figure 3.15).

```
final_client_write_key =
    PRF(client_write_key, "client write key", client_random + server_random)
final_server_write_key =
    PRF(server_write_key, "server write key", client_random + server_random)

iv_block = PRF(0, "IV block", client_random + server_random)
```

Figure 3.15 Export keying material derivation

SSLv3 Key Derivation

SLv3 key derivation is similar to TLS key derivation, except that the PRF is replaced with a series of expansion functions based on a combination of MD5 and SHA-1. The use of the constants "A","BB", etc. ensures that the output of each digest is different, even though the secret data is the same. This process is shown in Figure 3.16.

```
master_secret =
  MD5(pre_master_secret + SHA-1("A" + pre_master_secret +
    client_random + server_random)) +
  MD5(pre_master_secret + SHA-1("BB" + pre_master_secret +
    client_random + server_random)) +
  MD5(pre_master_secret + SHA-1("CCC" + pre_master_secret +
    client_random + server_random))

key_block =
  MD5(master_secret + SHA-1("A" + master_secret +
    server_random + client_random)) +
  MD5(master_secret + SHA-1("BB" + master_secret +
    server_random + client_random)) +
  MD5(master_secret + SHA-1("CCC" + master_secret +
    server_random + client_random)) +
  ...
```

Figure 3.16 SSLv3 key derivation

Note that the order of the client_random and server_random is opposite for these two computations.

As with TLS, when being used with exportable ciphers, SSLv3 postprocesses the keying material to weaken it, as shown in Figure 3.17.

```
final_client_write_key =
    MD5(client_write_key + client_random + server_random);
final_server_write_key =
    MD5(server_write_key + server_random + client_random);
```

```
client_write_IV =
  MD5(client_random + server_random);
server_write_IV =
  MD5(server_random + client_random);
```

Figure 3.17 SSLv3 export key derivation

Note that even though the SSLv3 key derivation was changed for TLS, no known weaknesses exist in the SSLv3 key derivation.

3.12 Record Protocol

The SSL record protocol is a fairly simple encapsulation protocol. It consists of a series of encrypted, integrity-protected records. Each record consists of a small header block and an encrypted data block which itself contains the content and a MAC. Figure 3.18 shows an example record. The header is shown in white and the encrypted payload is shaded. We've drawn the various pieces of the encrypted payload as they would appear if the encryption were removed.

The record shown has been encrypted with a block cipher such as DES using MD5 as a MAC. Therefore, the record needs to be padded out to fit the DES block length. Note that although the encrypted data must be a multiple of the block length, the record itself is not aligned on block boundaries.

In this trace, we have the client and the server writing one record each. The cipher suite, as we know from the ServerHello message, is TLS_RSA_WITH_DES_CBC_SHA.

SSL data is broken up into a series of *fragments*. The protocol definition provides a maximum fragment size but otherwise doesn't specify how to determine what fragment boundaries are, but usually they're one-to-one with application data write calls. However, boundaries of calls to `SSL_write()` (or equivalent) need not be preserved. Consequently, the record layer may break up any given write into multiple records or conversely aggregate the input from multiple calls into a single record. The maximum fragment data size is 2^{14} bytes, so larger writes must be fragmented into pieces of this size or smaller. Note that the length field is long enough to indicate records up to $2^{16} - 1$ bytes. Indeed, older versions of Internet Explorer incorrectly generated records of greater than 2^{14} bytes.

Once the data has been fragmented, it's compressed—or it would be if any compression algorithms were defined. Because none are, no compression is done. A MAC is computed over the compressed data fragment and concatenated with the data. The MAC covers the data and the headers, as well as including a sequence number to prevent replay and reordering attacks. Figure 3.19 shows the MAC algorithms.

```
TLS MAC = HMAC_hash (MAC_write_secret, seq_num + type + version +
                     length + content)
```

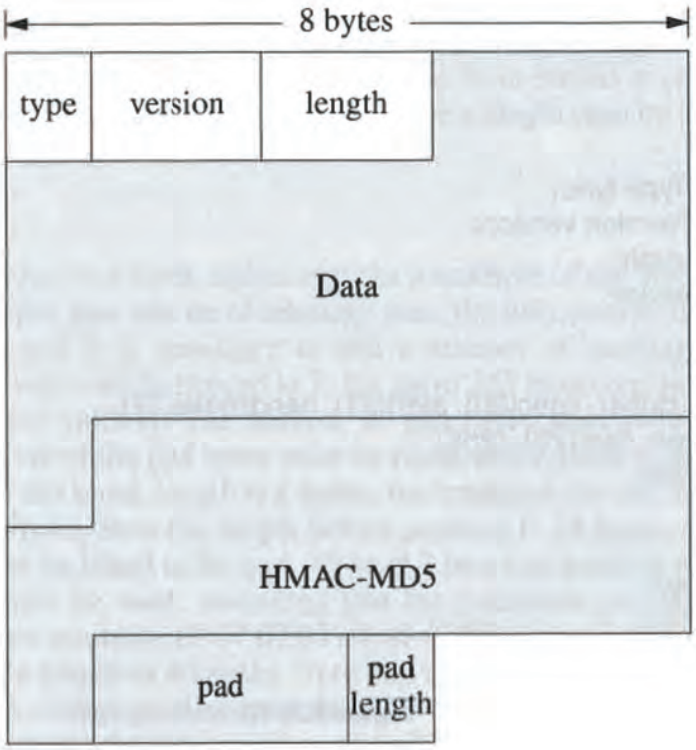


Figure 3.18 Diagram of an SSL record

```
SSLv3 MAC=hash(MAC_write_secret + pad_2 +
        hash(MAC_write_secret + pad_1 + seq_num + content_type +
          length + content));
```

Figure 3.19 MAC computations

seq_num is a 64-bit sequence number, which is independent for the client and server (i.e. the first message that both the client and server send has sequence number zero). The content type here is one of the content types listed in Figure 3.20. The version is simply the SSL/TLS version number. The length is the length of the content (i.e., the data in the record); it is *not* the same as the length in the record header, because that length includes the MAC and possibly padding.

The TLS MAC is standard HMAC. The SSLv3 MAC is based on an early draft of HMAC. pad_1 is the byte 0x36 repeated 48 times for MD5 and 40 times for SHA-1. pad_2 is the byte 0x5c repeated the same number of times. The intention of the padding is to ensure that the MAC key plus pad fills the entire first block of the message digest. Because MD5 and SHA-1 blocks are 64 bytes long, a 16-byte key means a 48-byte pad. Note that this implies that the padding for SHA-1 (a 20-byte key) should be 44 bytes,

but it's 40 in the specification. This is an error in the SSLv3 specification, but it became part of Netscape's implementation and thus the specification was never changed. Once again, we can see that Netscape's implementation constituted the real baseline for SSLv3.

```
struct {
   ContentType type;
   ProtocolVersion version;
   uint16 length;
} RecordHeader;

enum {
   change_cipher_spec(20), alert(21), handshake(22),
   application_data(23), (255)
} ContentType;

struct {
   uint8 major;
   uint8 minor;
} ProtocolVersion;
```

Figure 3.20 The record header

The sequence number serves to prevent various attacks on the message stream, including *replay* attacks, where the attacker repeats a record, and *reordering* attacks, where the attacker exchanges the order of one or more records. Note, however, that although the sequence number allows these attacks to be detected, because the sequence number isn't in the record, it does not allow them to be corrected. Thus, SSL must be run over a transport protocol that provides *reliable* transport (i.e., it must deliver data to the receiving application in the order it was transmitted without duplicates or omissions). TCP meets these requirements (the data may arrive in any order, but the kernel ensures that the application sees it in transmission order). However, UDP does not, and so SSL cannot be successfully run over UDP, because records might be arbitrarily lost or reordered, which would look like an active attack to the SSL implementation.

Note that the TLS specification describes records somewhat differently, without breaking them up into header and body. This presentation is intended to be easier to follow, and is more typical of how network protocols are usually presented.

The different type values correspond to the different types of data that can be carried over the TLS record layer, as shown in Figure 3.4. The length and version fields we've seen before. The length in the header is the total length of all the data following in the record, including the MAC and padding.

Stream Ciphers

Stream ciphers don't need any padding, so the MAC is simply appended to the data and the entire data block is encrypted. Because the stream cipher keystream can never be reused, the cipher encryption state must be retained from record to record. Logically, it is as if all the data to be encrypted were encrypted in a single pass by the cipher.

Block Ciphers

The length of the input to a block cipher must be a multiple of the block size (typically 8 bytes). Since the input data can be of arbitrary size, the data usually does not line up on a block boundary, and it is necessary to add a number of padding bytes. The most padding bytes that will ever be needed is 7, but up to 255 bytes can be added to hide the plaintext length from sniffers. The number of pad bytes used is recorded in the pad length byte. The value of the pad bytes must be equal to the value in the pad length byte.

For example, if the block length is 8 bytes, the length of the data is 16 bytes, and the MAC length is 16 bytes, then the length before padding is 33 bytes, because the length byte must always be included at the end. At least 7 bytes of padding will be needed, but 15, 23, etc., may also be used. Assuming that the minimum pad length of 7 bytes is used, the last 8 bytes would be 07 07 07 07 07 07 07 07 (7 pad bytes plus the pad length byte) for a total data length of 40 bytes. Note that this padding construct is incompatible with standard block cipher padding (see [RSA1993c]) which does not use a pad length byte, but merely records the total number of pad bytes in the padding. In the example here, the standard algorithm would give the pad 08 08 08 08 08 08 08 08.

All the current block ciphers use *cipher-block chaining* (CBC) mode. The chaining must be continued from record to record: The last cipher-block of record X is used as the IV for record X+1.

Null Cipher

As a special case, the Null cipher simply passes the data untransformed from input to output, with no padding. The starting cipher suite TLS_NULL_WITH_NULL_NULL doesn't even add a MAC, but simply puts the plaintext directly in the record. Note that TLS_NULL_WITH_NULL_NULL is available only during the handshake at the start of a connection. It is illegal to negotiate to it.

3.13 Alerts and Closure

The alert protocol is designed to allow one side to alert the other to exceptional conditions. Alert messages are very simple, conveying a severity level and a description. On

receiving an alert message of level fatal, an implementation must terminate the connection and the session cannot be resumed. A warning message may be ignored; however, an implementation may choose to treat warnings as fatal. Figure 3.21 shows the TLS alert message structure. The SSL alert message structure is identical, but fewer alerts are defined. See Chapter 4 for detailed coverage of all the alerts.

```
enum { warning(1), fatal(2), (255) } AlertLevel;

enum {
    close_notify(0),
    unexpected_message(10),
    bad_record_mac(20),
    decryption_failed(21),
    record_overflow(22),
    decompression_failure(30),
    handshake_failure(40),
    no_certificate(41), SSLv3 only
    bad_certificate(42),
    unsupported_certificate(43),
    certificate_revoked(44),
    certificate_expired(45),
    certificate_unknown(46),
    illegal_parameter(47),
    unknown_ca(48),
    access_denied(49),
    decode_error(50),
    decrypt_error(51),
    export_restriction(60),
    protocol_version(70),
    insufficient_security(71),
    internal_error(80),
    user_cancelled(90),
    no_renegotiation(100),
    (255)
} AlertDescription;

struct {
    AlertLevel level;
    AlertDescription description;
} Alert;
```

Figure 3.21 TLS alert message

Closure

The close_notify alert is used to signal that the sender has sent all the data that it is going to send on this connection. The purpose of this alert is to prevent a truncation attack in which the attacker inserts a TCP FIN before the sender is finished sending data, forcing the receiver to think that all data has been received. As a consequence, unless the receiver receives a close_notify it can never be sure that more data wasn't forthcoming. If an implementation receives a TCP FIN without first receiving a close_notify it must mark the session as not resumable.

In practice, many protocols contain their own end-of-data markers, and it's fairly common practice for an implementation receiving one to simply send a close_notify and close the connection immediately without waiting for a close_notify. This behavior is implicitly allowed by the SSL and TLS specifications. This is what has happened in the trace we provided at the beginning of the chapter, where the client provided a close_notify but the server did not. Normally, the closure alert is sent with a level of warning.

Other Alerts

All the other alerts signal errors of various kinds. Although in theory implementations can continue past a non-fatal alert, in practice even such alerts indicate that one side requested something that the other side couldn't fulfill and so they result in connection closure. The TLS specification provides a complete list of all the available alerts and we'll discuss each alert in detail in Chapter 4.

Figure 3.22 shows an example trace terminating in an alert. The client has been configured to require DSS/DH, and the server only supports RSA. Thus, the server rejects the connection with a handshake_failure alert. Note that the server closes the connection immediately after sending the alert.

```
New TCP connection: speedy(1085) <-> romeo(4433)
1 0.0168 (0.0168) C>S   Handshake
      ClientHello
        Version 3.0
        cipher suites
                  TLS_DHE_DSS_WITH_DES_CBC_SHA
        compression methods
                  NULL
2 0.0171 (0.0002) S>C   Alert
        level       fatal
        value       handshake_failure
Server FIN
Client FIN
```

Figure 3.22 A connection ending in an alert

3.14 Summary

This chapter has provided a detailed discussion of the most basic mode of SSL, RSA server-only authentication. This provides the background for our discussion of future modes, all of which share many features with the basic mode.

The SSL handshake negotiates cryptographic algorithms and establishes keying material between client and server. The client offers a choice of cryptographic algorithms and the server selects the algorithms to be used. The server provides its public key using a Certificate message.

The pre_master_secret *is encrypted under the server's public key.* A *key derivation function* is used to convert the pre_master_secret into the cryptographic keys used for the connection.

A Finished *message is used to provide integrity for the entire handshake.* The Finished message contains a message digest of all the handshake messages, preventing an attacker from tampering with the handshake.

The data stream is segmented into records. Each record is individually encrypted and MACed, providing both confidentiality and message integrity. The MAC is computed over a sequence number to prevent replay and reordering attacks.

Errors are signalled via alerts. Alerts are a special type of record distinct from the main data stream, and they are used to signal handshake errors or problems with MACing or decryption.

A close_notify *alert is used to signal the end of the connection.* This prevents truncation attacks where the attacker forges a close on the underlying transport layer.

This material may be protected by Copyright law (Title 17 U.S. Code)

4

Advanced SSL

4.1 Introduction

The previous chapter covered the simplest type of SSL connection, server-only authentication using RSA. This chapter presents the rest of the major operational modes. As with the previous chapter, this chapter is divided into two sections. The first section covers the modes in overview form and the second describes each message in gory detail. Most readers will want to read the entire first section and skim the second section. Those who wish to challenge themselves should read the entire second section.

Both sections follow the same order of presentation. First, we discuss session resumption, which allows a new SSL connection to be formed without performing a full handshake. Next, we cover client authentication, which allows the client to authenticate itself using its public key and digital certificate.

Because United States export regulations restricted the size of keys that could be used by applications exported from the U.S., SSL has two modes designed to provide maximum security while still remaining exportable, Ephemeral RSA and Server Gated Cryptography. Although these modes are no longer necessary, they're still in widespread use. We show how these work and describe their benefits. We also show the TLS standard DH/DSS key exchange and less-used cipher suites that use Elliptic Curves, Kerberos, and FORTEZZA. We finish with a detailed discussion of all the SSL alerts and of SSLv2 backward compatibility.

A note on presentation: although the timelines we show will include all the messages in the handshake, the detailed presentation will only show the messages which are new or different from the basic handshake.

Ex.1024
Askeladden, L.L.C. / Page 625 of 788