

4.2 Session Resumption

As we'll see in Chapter 6, the full SSL handshake can be very expensive, both in terms of CPU time and in terms of the number of round trips required to execute. In order to reduce this performance cost, SSL incorporates a session resumption mechanism. If the client and server have already communicated once, they can short-circuit the full handshake and proceed directly to data transfer.

The most expensive part of the handshake is the establishment of the `pre_master_secret`, which usually (except in the case of Kerberos) requires public key cryptography. A resumed handshake allows a new connection to use a `master_secret` established in a previous handshake. This avoids the computationally expensive operations required by public key cryptography.

Sessions versus Connections

SSL makes a distinction between a *connection* and a *session*. A *connection* represents one specific communications channel (typically mapped to a TCP connection), along with its keys, cipher choices, sequence number state, etc. A *session* is a virtual construct representing the negotiated algorithms and the `master_secret`. A new session is created every time a given client and server go through a full key exchange and establish a new `master_secret`.

Multiple connections can be associated with a given session. Although all connections in a given session share the same `master_secret`, each has its own encryption keys, MAC keys, and IVs. This is absolutely necessary for security reasons, because reuse of symmetric keying material can be extremely dangerous. Resumption allows the generation of a new set of symmetric keys and IVs from a common `master_secret` because the keys depend on the random values which are fresh for each connection. The new random values are combined with the old `master_secret` to produce new keys.

How It Works

The first time a client and server interact, they create both a new connection and a new session. If the server is prepared to resume the session, it gives the client a `session_id` in its `ServerHello` message and caches the `master_secret` for later reference. When the client initiates a new connection with the server, it uses that `session_id` in its `ClientHello` message. The server agrees to resume the session by using the same `session_id` in its `ServerHello`. At this point, the rest of the handshake is skipped and the stored `master_secret` is used to generate all the cryptographic keys. Figure 4.1 shows the timeline for this process.

4.3 Client Authentication

So far, all the handshakes we've discussed authenticate only the server. SSL also provides a mechanism for cryptographically authenticating the client. This would be useful

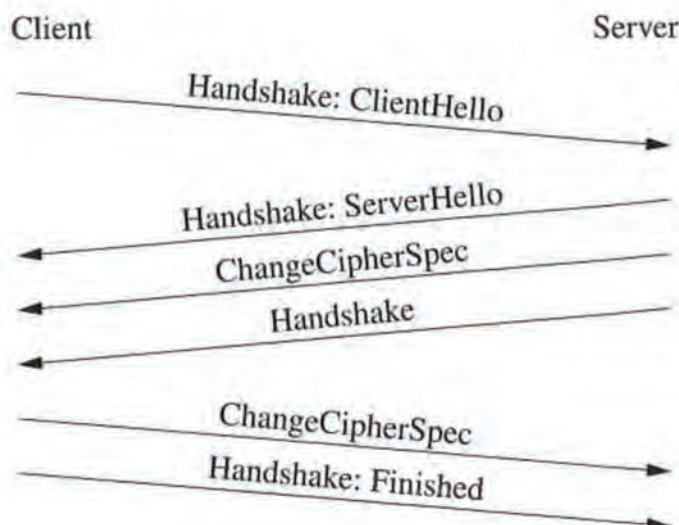


Figure 4.1 Resumed SSL session timeline

if a server wanted to restrict access to some services to only certain authorized clients and could use client authentication to do so. The idea here is that the client uses its private key to sign something, thus proving that it has possession of the private key corresponding to the certificate. Client authentication is initiated by the server sending a `CertificateRequest` message to the client. The client responds by sending a `Certificate` message (the same message that the server uses to transmit its certificate) and a `CertificateVerify` message. The `CertificateVerify` message is a string signed with the private key associated with the certificate that it transmitted. Client authentication is *always* initiated by the server. There is no mechanism for the client to unilaterally offer client authentication. Figure 4.2 shows the timeline for this process.

4.4 Ephemeral RSA

When SSL was designed, United States export regulations limited RSA encryption key lengths to 512 bits for exportable applications. Unfortunately, a 512-bit permanent RSA key presents an attractive target for attack. Thus, a server that wished to communicate with both domestic and exportable clients would have liked to have two key exchange keys, one 1024-bit and one 512-bit, assuming that it wanted to have strong cryptography when possible.

Although SSLv2 simply forced such a server to have two certified keys if it wanted to communicate at the highest possible security with both export and domestic clients, SSLv3 and TLS incorporate a feature called *ephemeral RSA* which allows

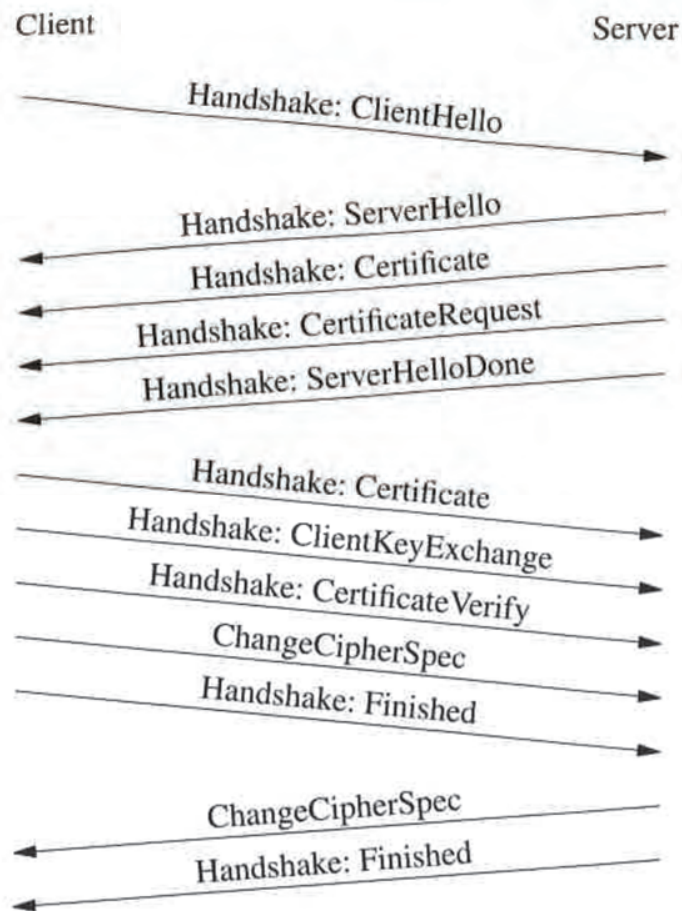


Figure 4.2 Client authentication timeline

communication between an exportable client and a domestic server with a permanent strong key. The way this works is that the server generates a temporary 512 bit key which it signs with its strong key. When communicating with domestic clients, it simply uses its strong key as discussed in Chapter 3.

The only difference in the handshake between ephemeral RSA mode and normal RSA mode is the presence of a single new message: `ServerKeyExchange`. The server uses the `ServerKeyExchange` message to transmit the signed RSA key. When the client receives the `ServerKeyExchange` message, it verifies the server's signature over the ephemeral key and then uses it to wrap the `pre_master_secret` exactly as described in Chapter 3. Figure 4.3 shows the timeline for this process.

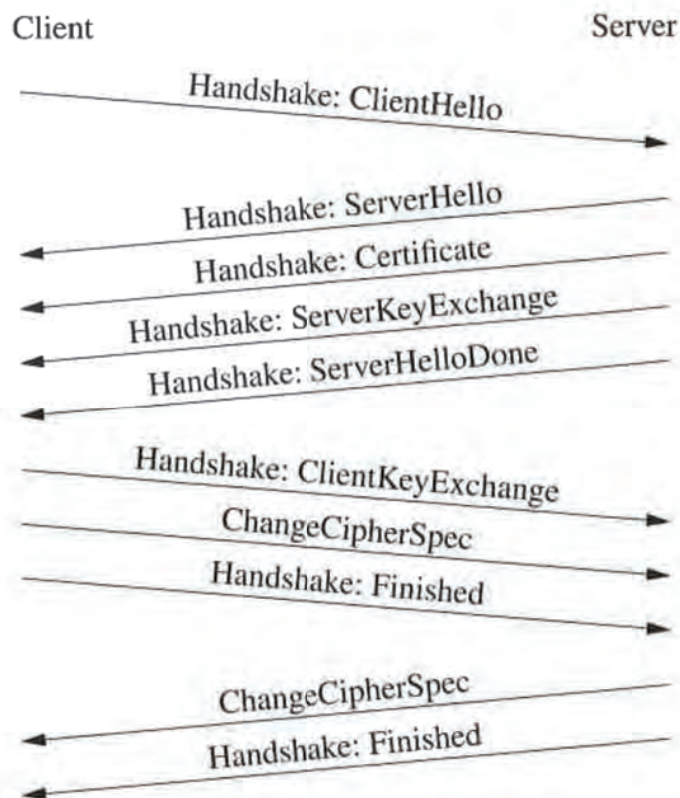


Figure 4.3 Ephemeral RSA timeline

4.5 Rehandshake

Once an SSL connection has been established, it's possible to perform a *rehandshake*. A rehandshake is simply a new SSL handshake performed over the current protected connection. Thus, the handshake messages are encrypted in transit. Once the new handshake has finished, data will be protected using the new session state.

```
struct { } HelloRequest;
```

The client can initiate a new handshake simply by sending a new `ClientHello` message. If the server wishes to initiate a rehandshake, it must send the empty `HelloRequest`

handshake message. The client responds to a `HelloRequest` with a `ClientHello`, after which the handshake proceeds normally. There's no requirement that either client or server agree to rehandshake. An implementation which does not wish to rehandshake can simply ignore the message or send a `no_renegotiation` alert. The next section shows an example of a client-initiated rehandshake. Chapter 8 shows an example of a server-initiated rehandshake.

4.6 Server Gated Cryptography

Even before they were liberalized, the United States export regulations provided an exception allowing for the use of strong cryptography for certain financial transactions. In order to accommodate this, many SSL implementations include a feature that allows otherwise weak clients to detect that they're talking to a special server and upgrade to strong cryptography. This feature is variously referred to as *Server Gated Cryptography* (SGC) by Microsoft and *Step-Up* by Netscape. Actually, SGC and Step-Up are different variations of SSL. However, the difference isn't relevant at the moment and we'll simply refer to them collectively as SGC for the rest of this section.

SGC relies on the server having a special certificate that indicates that it is allowed to engage in strong cryptography with exportable clients. This certificate is tagged in two ways. First, it must be issued by one of a very short list of trustworthy CAs. Currently, Verisign and Thawte are the only permitted issuers. In the non-SGC situation, CAs merely need to be trusted by the user. However, in this case the CAs need to be trusted by the U.S. government not to issue SGC-capable certificates to non-financial servers. Second, the certificate contains an extension indicating that the certificate-holder is SGC-capable.

Note that this means that SGC-capable clients must actually contain all the code to do strong cryptography. Whether it is used or not actually depends on a simple test in the code. Actually, because the key reduction stage happens in the handshake, even export-only clients must contain strong versions of the symmetric encryption algorithms. In practice, then, browsers contain all the strong crypto. In Netscape, a compiled-in table controls which algorithms are available. In 1998 a program called *Fortify* was released, which takes advantage of this fact to turn on the strong cryptography in an exportable Netscape by patching this table in the binary.

SGC works by using the rehandshake feature of SSL. In the first handshake, the client offers only weak ciphers. Once the client has seen the server's certificate and can verify that SGC is appropriate, it initiates a second handshake, offering strong ciphers in its `ClientHello`. The messages for the second handshake are transmitted over the current protected session. Figure 4.4 shows a timeline of a Netscape client doing Step-Up.

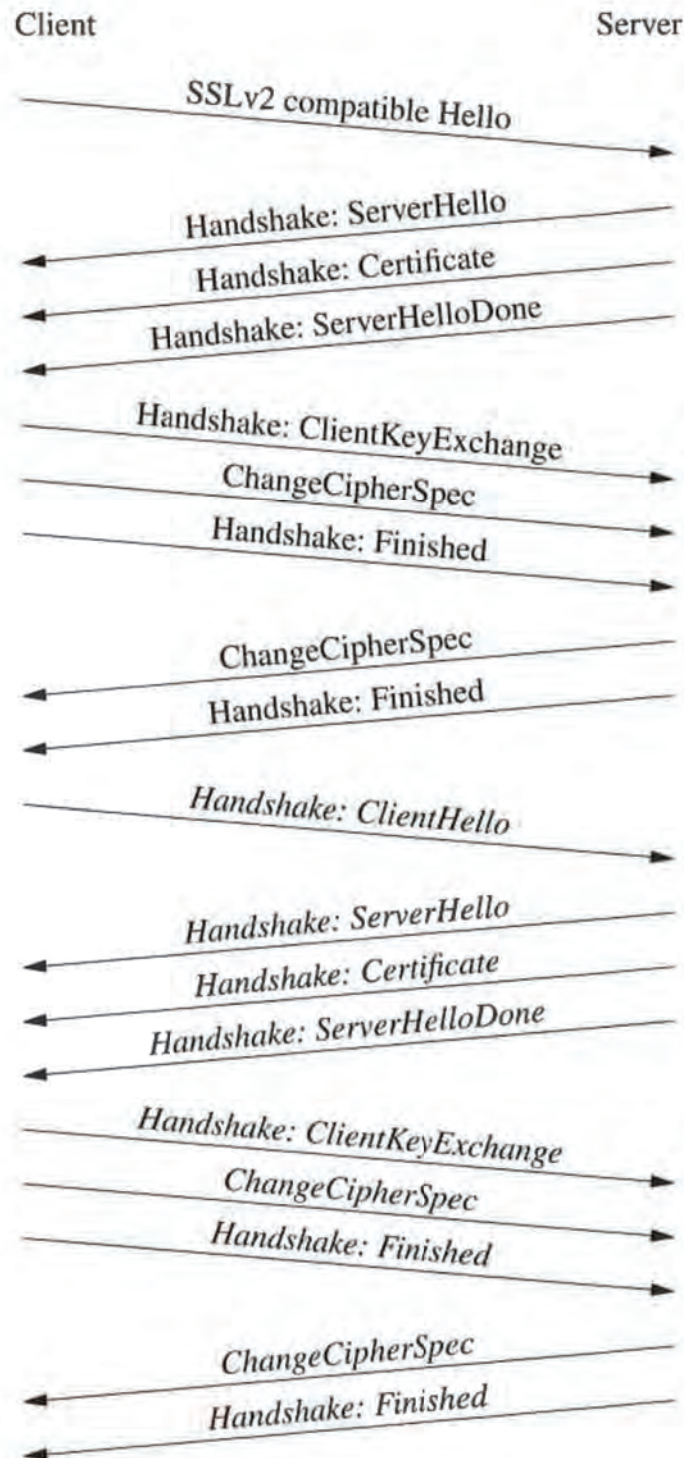


Figure 4.4 Step-Up timeline

4.7 DSS and DH

Although RSA is by far the dominant public key algorithm supported by SSLv2 and SSLv3 implementations, SSLv3 supports a number of cipher suites based on other algorithms, notably Digital Signature Standard (DSS) and Diffie-Hellman (DH). As discussed in Chapter 2, support for these ciphers was made mandatory in TLS. The main idea here was to avoid patented algorithms—in particular RSA. The patent on Diffie-Hellman expired in 1997. The patent situation of DSS is unclear in that RSA Data Security has claimed that the Schnorr patent (U.S. Patent 4995082) [Schnorr1991] covers DSS, but the U.S. National Institute of Standards and Technology claims otherwise. With all that in mind, the IETF decided to go with the devil it *didn't* know and mandate DH/DSS. Since the RSA patent has since expired, this motivation is no longer compelling, but support for DH/DSS lingers on.

Unlike RSA, which can be used for either key exchange or signature, DH can only be used for key agreement and DSS can only be used for digital signature. As a consequence, in order to get a complete solution, DH and DSS are typically used together. In such a system, the certificates are also usually signed with DSS.

There are two primary ways to use DH and DSS in SSL. The most common is to use *ephemeral DH* keys, which are analogous to ephemeral RSA keys. In this case, however, the server generates a temporary DH key and signs it with its DSS key, transmitting the signed key in the `ServerKeyExchange` message. The client then uses that DH key for key agreement. It is also possible to have a long-term DH key. In this case, the server will have a certificate (signed with DSS) containing its DH key. The client then uses the key in the certificate for key agreement.

The process of key agreement is the same no matter how the client gets the server's DH key. The client needs a DH key in the same group. Normally it won't have one, in which case it will have to generate one. It then transmits the DH key in the `ClientKeyExchange` message. At this point, the client and the server each have their own private key and the other side's public key, so they can independently compute the DH shared secret. This shared secret is used as the `pre_master_secret`. From there, the rest of the connection proceeds exactly as with RSA. Figure 4.5 shows the timeline for a handshake using ephemeral DH/DSS.

Note that the cipher suite numbers for DH/DSS are different from those for RSA, even if the symmetric algorithms are identical. Furthermore, the numbers for static DH/DSS are different from those for ephemeral DH/DSS. This highlights an unfortunate consequence of specifying all the algorithms together as a single cipher suite: there is no guarantee that symmetric algorithm support is orthogonal to public key algorithm support. For instance, RFC 2246 specifies five symmetric ciphers that can be used with RSA key exchange (DES, 3DES, RC2, RC4 and IDEA) but only two (DES and 3DES) for use with DH/DSS.

This situation is partially a result of resistance on the part of the working group to using proprietary algorithms such as RC4 and IDEA and partly a result of simple omission. Because positive action had to be taken to provide a full set of symmetric ciphers for DH/DSS, the WG only bothered to standardize the ciphers for which there was significant support and minimal resistance.

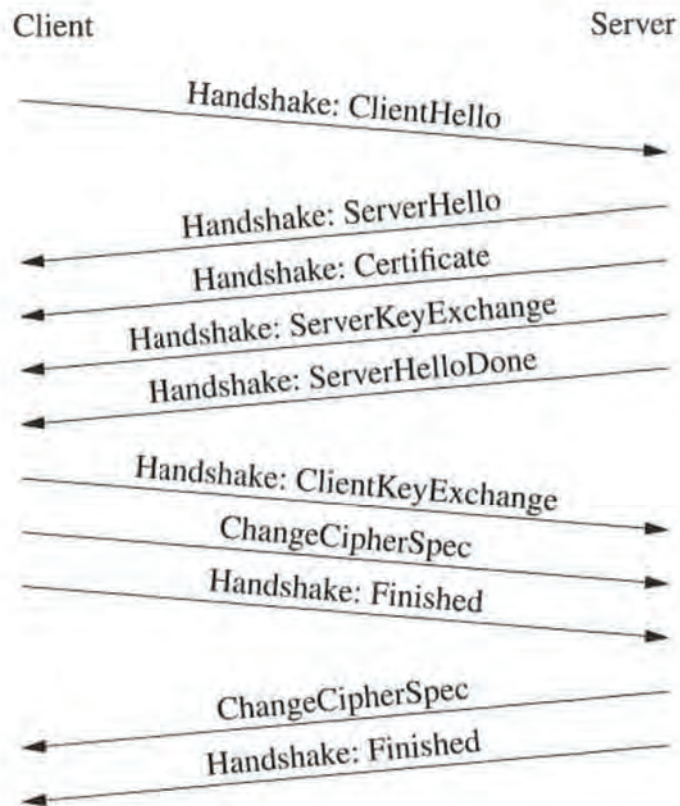


Figure 4.5 DSS/ephemeral DH timeline

4.8 Elliptic Curve Cipher Suites

Recently there has been a fair amount of interest in Elliptic Curve (EC) ciphers. To grossly oversimplify, EC ciphers replace the prime integer field of DH and DSS with a field composed of points on an elliptic curve. The discrete logarithm problem may be harder in such a group, so keys can be shorter and the public key operations comparatively faster. This is especially useful for smartcards where both memory and processing power are extremely limited, but even on fast servers the cost of the modular exponentiations in conventional public key crypto systems is quite high. The idea is that an EC system can be dropped in as a replacement for DH and DSS.

As of this writing, an Internet-Draft has been published for adding several EC-based cipher suites, but no such cipher suites have been standardized. One of the primary obstacles to standardizing such ciphers is that the intellectual property situation is rather vague. At the very least, Certicom and Apple both hold important patents on efficient implementation of EC ciphers, and there is some concern that the ciphers themselves may be patent encumbered.

4.9 Kerberos

Kerberos [Miller1987] is a popular symmetric-key-based authentication system developed at MIT. The general idea is that there is a central server (called a *Ticket Granting Server* (TGS)) that is trusted by every entity in the system. Each entity shares a symmetric key with the TGS. When a client wishes to communicate with a server, it requests a *ticket* from the TGS. The ticket is encrypted using the target's shared key and contains authenticating information for the requestor and a session key to be used for further communication between the two entities. The requestor then sends the ticket to the target.

RFC 2712 [Medvinsky1999] describes how to use Kerberos with TLS. The Client-KeyExchange contains both the ticket and an encrypted `pre_master_secret`. The shared key found in the ticket is used to encrypt the `pre_master_secret`. The server extracts the shared key from the ticket and decrypts the `pre_master_secret`. From that point on the handshake proceeds as usual. The Kerberos cipher suites have not seen wide deployment. Figure 4.6 shows the Kerberos cipher suites defined in RFC 2712.

Cipher Suite	Authentication	Key Exchange	Encryption	Digest	Number
TLS_KRB5_WITH_DES_CBC_SHA	Kerberos	Kerberos	DES_CBC	SHA	0x001e
TLS_KRB5_WITH_3DES_EDE_CBC_SHA	Kerberos	Kerberos	3DES_EDE_CBC	SHA	0x001f
TLS_KRB5_WITH_RC4_128_SHA	Kerberos	Kerberos	RC4_128	SHA	0x0020
TLS_KRB5_WITH_IDEA_CBC_SHA	Kerberos	Kerberos	IDEA_CBC	SHA	0x0021
TLS_KRB5_WITH_3DES_EDE_CBC_MD5	Kerberos	Kerberos	3DES_EDE_CBC	MD5	0x0022
TLS_KRB5_WITH_DES_CBC_SHA	Kerberos	Kerberos	DES_CBC	SHA	0x0023
TLS_KRB5_WITH_RC4_128_MD5	Kerberos	Kerberos	RC4_128	MD5	0x0024
TLS_KRB5_WITH_IDEA_CBC_MD5	Kerberos	Kerberos	IDEA_CBC	MD5	0x0025
TLS_KRB5_EXPORT_WITH_DES_CBC_40_SHA	Kerberos	Kerberos	DES_40_CBC	SHA	0x0026
TLS_KRB5_EXPORT_WITH_RC2_CBC_40_SHA	Kerberos	Kerberos	RC2_40_CBC	SHA	0x0027
TLS_KRB5_EXPORT_WITH_RC4_40_SHA	Kerberos	Kerberos	RC4_40	SHA	0x0028
TLS_KRB5_EXPORT_WITH_DES_CBC_40_MD5	Kerberos	Kerberos	DES_40_CBC	MD5	0x0029
TLS_KRB5_EXPORT_WITH_RC2_CBC_40_MD5	Kerberos	Kerberos	RC2_40_CBC	MD5	0x002A
TLS_KRB5_EXPORT_WITH_RC4_40_MD5	Kerberos	Kerberos	RC4_40	MD5	0x002B

Figure 4.6 The Kerberos cipher suites

4.10 FORTEZZA

The FORTEZZA card is a U.S. government-designed cryptographic token in a PCMCIA form factor. It uses DSA and SHA for signature and message digests but NSA-designed key agreement and encryption algorithms. The key agreement algorithm is a variant of Diffie-Hellman called *Key Exchange Algorithm* (KEA) and the encryption algorithm is a block cipher called SKIPJACK.

FORTEZZA was originally designed by the NSA to provide strong cryptography while allowing the NSA to intercept communications. The conflict between these goals was resolved by incorporating a *key escrow* feature into the device. Each card had its own key, which was escrowed with the NSA. Thus, when the NSA wished to decrypt a communication, it could recover the key for the card used to encrypt it, but the encryption would be secure against all other attackers.

The FORTEZZA cipher suites require that the client be authenticated. This can happen in one of two ways. The client can either have a KEA certificate or it can have a DSA certificate and use its DSA key to sign a KEA share. In either case, the client will have a certificate to the server. Figure 4.7 shows a FORTEZZA handshake.

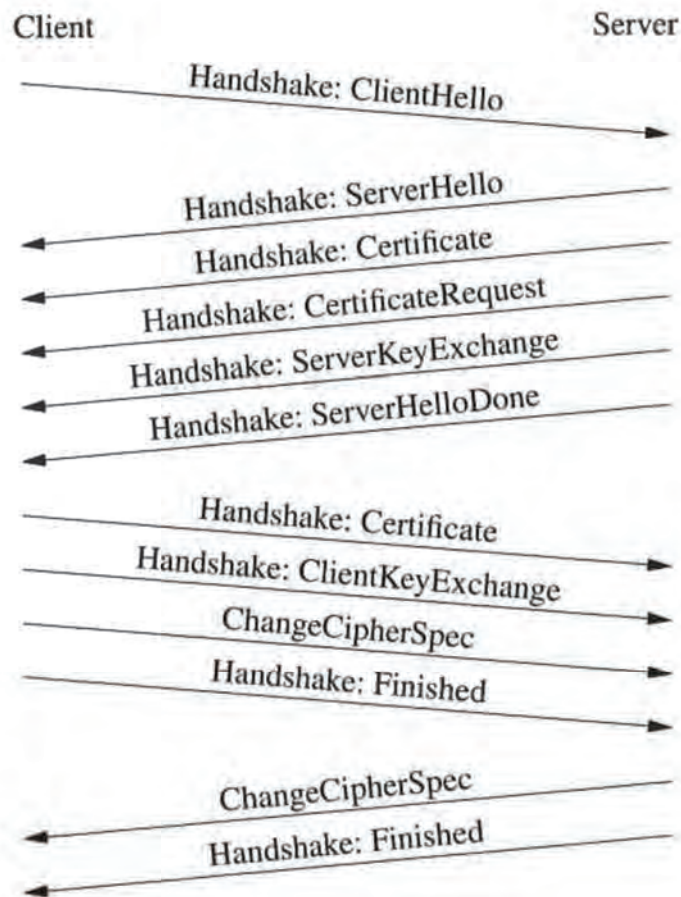


Figure 4.7 FORTEZZA handshake

Even though the server's KEA key is in its certificate, the server still sends a *ServerKeyExchange* message. This is used to transmit a random value which is used as part of the KEA key agreement process. Also, because the client must be authenticated, the server sends a *CertificateRequest* message. However, since client authentication happens via key agreement, no *CertificateVerify* is sent.

When FORTEZZA cards were originally released, SKIPJACK and KEA were classified. FORTEZZA support was specified in SSLv3 but removed for TLS because IETF won't standardize algorithms that aren't fully specified. Netscape did deliver FORTEZZA-capable versions of Navigator and Netscape Server, but they never saw wide use outside the U.S. government. Nevertheless, using FORTEZZA with SSL presents some interesting technical challenges so we'll discuss the details in Section 4.17.

4.11 The Story So Far

As we've seen, in addition to the basic RSA server-only authentication mode, SSL supports a number of other modes with special properties. All of these modes require a different handshake than that required for server-only RSA. The next half of the chapter covers these modes in detail, showing the contents of each new message and how they work together to provide new handshake functionality. The important thing to remember through all this is that, with the exception of FORTEZZA, these variations are simply changes to the handshake. Key derivation from the `master_secret` and data protection at the record layer are identical no matter which handshake variant you use.

4.12 Session Resumption Details

Session resumption is accomplished using the `session_id` field of the `ClientHello` and `ServerHello` messages. On the initial handshake between any client and server pair, the client's `session_id` field is absent because the client has no session cached. If the server is willing to allow resumption—it almost always is—it generates a `session_id` and returns it in the `session_id` field of the `ServerHello`. When the handshake is finished, both the client and the server store the session information (the keying material and negotiated ciphers) somewhere that they can get to them later if the session is resumed.

When a client wishes to resume a session, it uses the appropriate `session_id` in the `ClientHello`. If the server is willing to resume the session, it responds with the same `session_id` in the `ServerHello`. The server then skips the rest of the handshake and immediately sends a `ChangeCipherSpec` message. The `key_block` is recomputed using the stored `master_secret` and the new client and server random values, so all the keys are different.

Figure 4.8 shows the detailed handshake messages for a resumed TLS session. Note that the `session_id` field in the `ClientHello` matches the `session_id` in the `ServerHello`. If the server did not want to resume the session, it would simply generate a new `session_id` and continue with the handshake as if the client had not attempted to resume. Typically this happens because the server no longer remembers the session that the client wants to resume, either because it maintains a limited size cache or because the session has timed out.

When resuming a session, the client is required to offer at least the cipher suite that was originally negotiated. If the server agrees to resume, it must choose that cipher suite. In this case, the client has offered a full set of cipher suites. If the server were to choose not to resume the session, it could choose any of the offered cipher suites.

```
New TCP connection: romeo(1300) <-> speedy(4433)
1 0.0022 (0.0022) C>S Handshake
  ClientHello
    resume [32]=
      6d f3 02 8a 44 a7 42 94 14 7b 59 ad
      8f 00 32 71 e9 a5 d1 bc 3f c0 23 0c
      50 fa 4f 9f 27 cf 45 c4
    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 0.0288 (0.0266) S>C Handshake
  ServerHello
    session_id[32]=
      6d f3 02 8a 44 a7 42 94 14 7b 59 ad
      8f 00 32 71 e9 a5 d1 bc 3f c0 23 0c
      50 fa 4f 9f 27 cf 45 c4
    cipherSuite          TLS_RSA_WITH_RC4_128_SHA
    compressionMethod    NULL
3 0.0288 (0.0000) S>C ChangeCipherSpec
4 0.0288 (0.0000) S>C Handshake: Finished
5 0.0293 (0.0005) C>S ChangeCipherSpec
6 0.0293 (0.0000) C>S Handshake: Finished
```

Figure 4.8 Resumed TLS session handshake messages

Session ID Lookup

Servers and clients use different information to retrieve sessions. When a client initiates an SSL connection, the only available information is the server's hostname (or IP address if no hostname was used) and port, so these must be used to look up the session ID. However, when a server receives a request for a resumed session for the client, the `session_id` is available and can be used to look up the session. Since the lookup is based on the `session_id`, session resumption will work even when the client is connecting from a different IP address and port than the one used for the initial handshake.

Because initiating TCP ports are sequentially chosen, the client's port will almost always be different from the port used for the original connection. Usually the client's IP address will be the same for every connection. However, because many clients have their IP addresses dynamically assigned via DHCP or similar mechanisms it is quite possible for a client to attempt to resume a session using a different IP address than the one it used to create it. Servers should support this.

Although permitting resumption from a client with a different IP address may seem dangerous, it is actually secure: first, it's trivial for an attacker to forge messages from a given IP address, so checking the IP address offers little value. More important, the server checks the client's Finished message at the end of the handshake. In order to correctly generate the Finished message, the client must have access not only to the new encryption and MAC keys (in order to encrypt and MAC the message) but to the `master_secret` in order to compute the Finished hashes. Since an attacker with access to the `master_secret` could easily take over an existing connection, this is equally secure. Note that to reduce the effect of compromise of any given `master_secret`, servers typically expire sessions after some period of time (a day is common.)

4.13 Client Authentication Details

As we've seen, SSL client authentication uses two new messages that we haven't seen before, as well as a new usage for an old message. The first new message is the `CertificateRequest` message which is used by the server to request client authentication. The client responds by sending the `Certificate` message (which we previously saw being used by the server) to send its certificate and the `CertificateVerify` message to authenticate itself.

CertificateRequest

```
struct {
    ClientCertificateType certificate_types<1..2^8-1>;
    DistinguishedName certificate_authorities<3..2^16-1>;
} CertificateRequest;
```

```
enum {
    rsa_sign(1), dss_sign(2), rsa_fixed_dh(3), dss_fixed_dh(4),
    (255)
} ClientCertificateType;
```

```
opaque DistinguishedName<1..216-1>;
```

```
CertificateRequest
    certificate_types          rsa_sign
    certificate_types          dss_sign
    certificate_authority
        C=AU
        ST=Queensland
        O=CryptSoft Pty Ltd
        CN=Server test cert (512 bit)
    certificate_authority
        C=US
        O=AT&T Bell Laboratories
        OU=Prototype Research CA
    certificate_authority
        C=US
        O=RSA Data Security, Inc.
        OU=Commercial Certification Authority
    certificate_authority
        C=US
        O=RSA Data Security, Inc.
        OU=Secure Server Certification Authority
    certificate_authority
        C=ZA
        ST=Western Cape
        L=Cape Town
        O=Thawte Consulting cc
        OU=Certification Services Division
        CN=Thawte Server CA
        Email=server-certs@thawte.com
    certificate_authority
        C=ZA
        ST=Western Cape
        L=Cape Town
        O=Thawte Consulting cc
        OU=Certification Services Division
        CN=Thawte Premium Server CA
        Email=premium-server@thawte.com
```

The `CertificateRequest` message indicates to the client that the client should perform client authentication and provides guidance as to what sorts of certificates the server is willing to accept. The server is allowed to specify both the cryptographic algorithm that should be used and a list of the certificate authorities which it trusts to sign client certificates.

The `certificate_types` field is an array of single byte values, each of which describes a permitted signature algorithm. In this case, the server has specified `rsa_sign` and `dss_sign`, indicating that it is prepared to accept a response signed with either RSA or DSS. Note that this value also specifies the algorithm which was used to sign the certificate.

The other two possible values, `rsa_fixed_dh` and `dss_fixed_dh` actually do not involve a signature at all. Rather, they indicate that the client and server will use fixed Diffie-Hellman key pairs to agree on the `pre_master_secret`. If these methods are chosen, the client does not send a `CertificateVerify` because the connection is implicitly authenticated by the ability to generate the appropriate keys from the `pre_master_secret`. However, see Section 4.16 for an explanation of fixed DH.

The `certificate_authorities` field specifies a list of certificate authorities (CAs) from which the server is willing to accept certificates. These CAs are specified by using a BER encoded version of the CA's distinguished name. See Chapter 1 for more information on distinguished names and certificates. It is important to note that if certificate chains are being used, then the CA name specified in the `CertificateRequest` message need not refer to the CA that signed the client's certificate, but may instead refer to one of the parent CAs.

Certificate

```
struct {
    ASN.1Cert certificate_list<1..224-1>;
} Certificate;

opaque ASN.1Cert<224-1>;
```

We saw the `Certificate` message before when the server used it to provide its certificate to the client. Here it is used by the client to send its certificate for client authentication. This message has exactly the same structure as the server's `Certificate` message, although the certificates themselves are different.

CertificateVerify

```
select (SignatureAlgorithm) {
    case anonymous: struct { };
```

```

case rsa:
    digitally-signed struct {
        opaque md5_hash[16];
        opaque sha_hash[20];
    };
case dsa:
    digitally-signed struct {
        opaque sha_hash[20];
    };
} Signature;

struct {
    Signature signature;
} CertificateVerify;

CertificateVerify
Signature[64]=
    5a a7 6a 48 54 17 c9 a3 09 87 90 37
    42 17 45 c7 bf de 24 0f 4d 73 9b a8
    69 a6 c7 f3 a0 41 c5 26 fc 6a 48 0d
    e1 02 4f 9d 1b d2 17 55 dc 4e 55 11
    6f 89 7c ed 68 59 70 db b4 e7 fe 11
    d7 2f f6 83

```

The CertificateVerify does all the actual work of client authentication. It contains a digest of the handshake messages signed by the client's private key (the one that corresponds to the certificate in the client's Certificatemessage). This is what the digitally-signed operator means. Because only the holder of the private key corresponding to the certificate could sign this message, the server can conclude that the entity named in the certificate is connecting.

In SSLv3, we compute the hashes of the handshake messages as shown:

```

CertificateVerify.signature.md5_hash =
    MD5(master_secret + pad2 +
        MD5(handshake_messages + master_secret +
            pad1));

CertificateVerify.signature.sha_hash =
    SHA-1(master_secret + pad2 +
        SHA-1(handshake_messages + master_secret +
            pad1));

```

Including the `master_secret` in the messages actually doesn't add much security value because it's implied by the other handshake messages. TLS uses a simple digest of the handshake messages.

```
CertificateVerify.signature.md5_hash = MD5(handshake_messages);
CertificateVerify.signature.sha_hash = SHA-1(handshake_messages);
```

The details of the signature computation depend on which signature algorithm is being used. In this handshake, the client is signing using RSA, so the signed object is the concatenation of MD5 and SHA-1 digests of the handshake messages up to this point in the handshake. Because DSS requires a 160-bit input, when DSS is being used, only the SHA-1 digest of the messages is used. Because the digest covers the server's random value, the server can be sure that this signature is *fresh* (that is, not replayed) as long as it used a fresh random value in its `ServerHello`.

It is important to note that this form of RSA signature is different from the usual RSA signature described in [RSA1993b], which only includes one digest. In PKCS #1, the signature is typically computed over a DER encoded `DigestInfo` structure which also contains the algorithm identifier for the digest. Because the MD5 and SHA-1 digests are mandated as part of the SSL protocol, the digest algorithm identifier is unnecessary (and because there is no identifier for the MD5 and SHA-1 combination, it is also problematic). As a result, the concatenated hash value is signed directly without being first encoded in a `DigestInfo`. This can cause problems with hardware implementations or all-in-one PKI toolkits, which often want to perform the `DigestInfo` encoding as part of the signature.

Failure Conditions

If the client has no suitable certificate, a `Certificate` message containing no certificates should be sent. At this point, the server can either proceed with the connection, using some other means of authentication (for instance, prompting for a password in the application layer) or no authentication at all. If the server is unwilling to continue, it sends a fatal `handshake_failure` alert.

4.14 Ephemeral RSA Details

```
struct {
    select (KeyExchangeAlgorithm) {
        case diffie_hellman:
            ServerDHParams params;
            Signature signed_params;
        case rsa:
            ServerRSAParams params;
            Signature signed_params;
    };
} ServerKeyExchange;
```

```

struct {
    opaque RSA_modulus<1..216-1>;
    opaque RSA_exponent<1..216-1>;
} ServerRSAParams;

select (SignatureAlgorithm) {
    case anonymous: struct { };
    case rsa:
        digitally-signed struct {
            opaque md5_hash[16];
            opaque sha_hash[20];
        };
    case dsa:
        digitally-signed struct {
            opaque sha_hash[20];
        };
} Signature;

enum { rsa, diffie_hellman } KeyExchangeAlgorithm;
enum { anonymous, rsa, dsa } SignatureAlgorithm;

```

ServerKeyExchange

```

params
    RSA_modulus[64]=
        c9 61 5e 58 08 6a 5c f0 76 a4 5d 6e
        ae 99 51 f9 ae c6 2c 8a e5 34 1e 9d
        a6 cb 68 b4 d6 47 b2 01 d4 5d 9b 32
        3c 49 2a 90 ac c2 41 cd b4 20 4f 54
        c2 a0 26 e4 b8 ee 69 61 04 23 64 72
        f2 40 e3 9f
    RSA_exponent[3]=
        01 00 01
signature[128]=
    09 6d 34 ee 8c 2f 14 e5 52 05 26 25
    4c 34 72 35 ef 79 1f 07 c1 71 82 cd
    3d f6 7c 67 05 ff eb 02 bc da bf 8e
    3a e7 c8 37 14 5c ca 05 4a 69 5f 09
    f3 b0 fa 53 33 50 ab 41 e8 63 6b f5
    5a 5d c1 5d f0 09 6d aa bd d8 0b e2
    bb 13 0d e6 09 f8 91 11 b1 ec 25 58
    1f 7e b3 56 81 4e c0 03 71 a4 95 e4
    be 1f 2e 10 75 80 75 1a 65 ed c4 70
    09 0b 19 64 3f dd f0 42 4a 32 73 36
    82 b1 2b 24 2c a4 6b 3a

```

The `ServerKeyExchange` message contains a signed version of the server's ephemeral public key. Because this message is also used for ephemeral DH, the specification permits both RSA and DH keys. We'll see an example of ephemeral DH in Section 4.16. The message is signed using the server's long-term key, in this case an RSA key. The permitted algorithms are RSA signature, DSS, or anonymous, which simply means that the message is unsigned. Note that the anonymous mode offers no security at all against active attack. The attacker can trivially pose as the server to the client using a man-in-the-middle attack.

Although the specification permits cipher suites that mix-and-match key exchange and signature algorithms, the common combinations are RSA signature with RSA key exchange (shown at the beginning of this section) and DSS signature with DH key exchange (TLS implementations must implement DH/DSS). Cipher suites are defined for RSA signature with DH key exchange, but they are not commonly used. No cipher suites are even defined for DSS signature with RSA key exchange.

The RSA key is presented in the message as a pair of byte strings representing the RSA modulus and exponent. Following Internet convention, the large integers are mapped to byte strings in "network" byte order, with the first byte representing the most significant byte of the integer. The size of the RSA modulus is the important number, and it's 64 bytes long, consistent with a 512-bit key. As we saw in Chapter 1, standard practice is to use a very small public exponent for performance reasons, and here 65537 has been chosen. See Chapter 1 for more information on RSA keys.

The RSA signature shown in the message is computed using a pair of hashes as in the `CertificateVerify` message, but the input data to the hashes are different. The `ServerKeyExchange` hash incorporates only the parameters to be signed and the client and server random values. The random values serve to prevent replay of a previous `ServerKeyExchange` message in a new handshake:

```
md5_hash =  
    MD5(client_random + server_random + ServerParams);  
  
sha_hash =  
    SHA(client_random + server_random + ServerParams);
```

Because generating RSA keys is computationally expensive, SSL server implementations typically use the same ephemeral key to service a large number of clients. This isn't as secure as generating a fresh key for each client, because a large number of transactions (often as much as a day's worth) are protected under the same 512-bit key. However, the cost of generating a fresh key for each transaction would almost always be prohibitive.

There are two conditions under which ephemeral RSA can be used: first, the cipher suite used must be one of the export cipher suites (i.e., the key exchange algorithm is `RSA_EXPORT`). Second, the signing key must be longer than 512 bits. Otherwise, normal RSA must be used.

Before we get too excited about ephemeral RSA, it's worthwhile to do a back-of-the-envelope computation as to how much security it adds. Typical key lifetimes are on the order of a year or two and servers usually generate a new ephemeral RSA key at most

once a day (though on a modern CPU it's possible to do it far more frequently, say every few minutes). Often, they'll generate it at server startup only, and servers can stay up for weeks. So, if you wanted to read all the traffic to a given server, ephemeral RSA would increase your workload by less than a thousand times given typical use patterns. On the other hand, if you want to read a specific message, and you know when that message was sent, ephemeral RSA doesn't make your job any more difficult at all.

4.15 SGC Details

The difficulty faced by SGC/Step-Up is that the client must send its ClientHello before it knows whether the server is a normal SSL server or an SGC/Step-Up capable server. Thus, it must offer only export cipher suites. Otherwise, a normal server might select a strong cipher suite that the client couldn't use. Only when the client receives the server's Certificate message can it detect that the server is SGC/Step-Up capable. At this point, SGC and Step-Up differ. We'll discuss Step-Up first.

Step-Up

The server signals to the client that it's willing to do Step-Up by providing a special value in the extendedKeyUsage extension. When the client sees this value, it knows that it can Step-Up. The browser completes the SSL handshake that it's currently doing (which negotiates an export cipher suite) and then immediately rehandshakes to negotiate a strong cipher suite. Figure 4.9 shows a Step-Up connection in action.

```
New TCP connection: romeo(4290) <-> romeo(443)
1 0.0004 (0.0004) C>S SSLv2 compatible client hello
  Version 3.0
  cipher suites
    SSL2_CK_RC4_EXPORT40
    SSL2_CK_RC2_EXPORT40
    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 0.0058 (0.0053) S>C Handshake
  ServerHello
    session_id[32]=
      5e ae 37 80 c5 30 ff 35 59 65 2a 8c
      9b a1 8a 73 ad 2d 88 30 59 9a f9 58
      45 ac 8d a3 85 5f ca e9
    cipherSuite          TLS_RSA_EXPORT1024_WITH_RC4_56_SHA
    compressionMethod    NULL
```

```

3 0.0058 (0.0000) S>C  Handshake
    Certificate
      Subject
        C=US
        O=RTFM, Inc.
        CN=romeo.rtfm.com
      Issuer
        C=US
        ST=California
        O=RTFM, Inc.
        CN=Step-Up CA
      Serial          03
      Extensions
        Extension: X509v3 Subject Alternative Name
          <EMPTY>
        Extension: X509v3 Basic Constraints
          CA:FALSE, pathlen:0
        Extension: Netscape Comment
          mod_ssl generated custom server certificate
        Extension: Netscape Cert Type
          SSL Server
        Extension: X509v3 Extended Key Usage
          Netscape Server Gated Crypto
4 0.0058 (0.0000) S>C  Handshake
    ServerHelloDone
5 0.0248 (0.0189) C>S  Handshake
    ClientKeyExchange
      EncryptedPreMasterSecret[128]=
        a1 40 7c 99 2f 40 4d 01 dd b0 0a 7b
        f8 8e ee e3 1d f1 ed 35 04 ea 56 5f
        1a 3f 62 75 cf 6e bc b9 58 cf ad 33
        ba be 30 3c 63 d0 85 ea a7 a4 24 e2
        b5 dd d1 21 03 e1 87 bd cb cf 56 54
        8d ed 02 f9 67 d7 bc 9a 73 cb 51 a6
        c2 d6 c2 12 b6 96 06 45 db a3 ed d8
        40 0c ea 11 22 09 61 7c 98 85 8f ed
        ca 4a 51 52 bd e9 14 0a b3 5d 04 be
        58 79 1e 51 cd fd dc ae 66 23 b8 4f
        1b ea 68 10 eb 8a e7 87
6 0.2199 (0.1951) C>S  ChangeCipherSpec
7 0.2199 (0.0000) C>S  Handshake
    Finished
8 0.2223 (0.0023) S>C  ChangeCipherSpec
9 0.2223 (0.0000) S>C  Handshake
    Finished
First handshake completed.

```

Client initiates a new handshake offering strong cipher suites.

```

10 0.2231 (0.0008) C>S   Handshake
    ClientHello
      Version 3.0
      cipher suites
        TLS_RSA_WITH_RC4_128_MD5
        value unknown: 0xffe0 Netscape proprietary cipher suite
        TLS_RSA_WITH_3DES_EDE_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
11 0.2271 (0.0039) S>C   Handshake
    ServerHello
      session_id[32]=
        de 59 fc ef 7f b5 59 e0 92 f9 31 20
        da a3 f1 82 bf 78 ba 53 8b b6 8a a9
        d6 69 82 af 55 8b 99 27
      cipherSuite          TLS_RSA_WITH_RC4_128_MD5
      compressionMethod    NULL
12 0.2271 (0.0000) S>C   Handshake
    Certificate
      Subject
        C=US
        O=RTFM, Inc.
        CN=romeo.rtfm.com
      Issuer
        C=US
        ST=California
        O=RTFM, Inc.
        CN=Step-Up CA
      Serial              03
      Extensions
        Extension: X509v3 Subject Alternative Name
          <EMPTY>
        Extension: X509v3 Basic Constraints
          CA:FALSE, pathlen:0
        Extension: Netscape Comment
          mod_ssl generated custom server certificate
        Extension: Netscape Cert Type
          SSL Server
        Extension: X509v3 Extended Key Usage
          Netscape Server Gated Crypto

```

(continued)

Figure 4.9 (continued)

```

13 0.2271 (0.0000) S>C  Handshake
    ServerHelloDone
14 0.2381 (0.0110) C>S  Handshake
    ClientKeyExchange
      EncryptedPreMasterSecret[128]=
        98 29 0c 5f 72 b0 46 03 fd 3d ed 62
        c6 fc ec d3 e3 73 d3 f5 c8 a0 60 0a
        f7 94 de 18 0a 9c 3c db db 7c f4 9c
        34 65 90 dd 11 fd a2 d3 ae 50 f1 d6
        e8 22 79 72 fe 99 b8 e1 03 a4 4a 64
        22 4d e8 a5 58 d5 80 22 d5 da d4 dc
        fc 42 a0 f3 f3 50 a6 56 4f f4 ae 69
        37 7b 9c 8c 53 c8 d7 7a ac 91 2d 7b
        48 8b 10 e5 b0 55 65 45 99 3e ca 69
        f7 c5 9e ae 0b fe f1 36 7b 4c 1c 6d
        7e 4d 42 96 6b 21 55 d4
15 0.4199 (0.1818) C>S  ChangeCipherSpec
16 0.4199 (0.0000) C>S  Handshake
    Finished
17 0.4215 (0.0015) S>C  ChangeCipherSpec
18 0.4215 (0.0000) S>C  Handshake
    Finished

```

Figure 4.9 A connection using Step-Up

Figure 4.9 shows a Step-Up connection between a Netscape client and an Apache server with `mod_ssl`. Note that in the initial message, the client sends an SSLv2-compatible hello offering only export cipher suites. It must do this on the initial connection because it doesn't yet know that the server is Step-Up capable. On future connections, of course, it could offer strong ciphers immediately if it remembered that the server had a Step-Up certificate.

The server's certificate, shown in record 3, contains the Step-Up extension (shown here as Netscape Server Gated Crypto), so the client knows it can do Step-Up. However, the client and the server have already agreed to a set of cipher suites, so this information isn't immediately useful because there's no way in SSL for the client to change its mind at this point. The client could drop the connection and reconnect, but that's not the way that Step-Up works. Instead, the client completes the handshake.

Once the handshake is completed, the client initiates a rehandshake over the newly established encrypted channel. The rest of Figure 4.9 shows that rehandshake. In record 10 the client sends a new `ClientHello`. This time it offers strong cipher suites as well as

weak ones and naturally the server accepts one. From there, the handshake completes as usual.

Note that the client does not attempt to resume the session. Most of the export cipher suites use ephemeral RSA, in which case resuming is impossible, because we need to establish a new master_secret using the server's 1024-bit RSA key. However, the new (and still unstandardized) EXPORT1024 cipher suites use the long RSA key for key exchange, so this session could technically be resumed. However, the client doesn't offer this. As a consequence, an entirely new RSA key exchange must be performed. Thus, Step-Up handshakes are (at least) twice as expensive computationally as ordinary handshakes.

This trace also shows the SSL feature of *rehandshake*. At any time during a connection, SSL implementations can initiate a new handshake. The new handshake is carried over the currently protected channel. Thus, the second handshake is entirely encrypted. The reason we're able to read it is that `ssldump` has been given the server's private key and is automatically decrypting the data for us. All the encrypted records are rendered in a *fixed-width italic* font to show that we've decrypted them. At the end of the handshake, all traffic switches over to the newly negotiated connection.

Careful inspection of Figure 4.9 reveals that the server's certificate was signed by the author's local CA. However, we said earlier that Step-Up certificates must be signed by specific trusted CAs. Surely the author's CA is not one of them! We've taken advantage of a weakness in Netscape's implementation. The CAs are stored in a database and being able to issue Step-Up certificates is simply a database attribute. We've changed that attribute in our local database, thus allowing ourselves to issue Step-Up certificates. We could have just as well connected to a legitimate Step-Up server, but then we wouldn't have the server's private key and be able to decrypt the traffic.

SGC

In the previous section, we saw that Step-Up requires two key exchange stages. This is expensive and the first stage isn't really adding much security value. Microsoft noticed this and SGC used only one key exchange stage. Figure 4.10 shows an SGC connection between IE and Internet Information Server (IIS).

```
New TCP connection: swagger(1897) <-> 151(443)
/ 0.3679 (0.3679) C>S SSLv2 compatible client hello
Version 3.0
cipher suites
  TLS_RSA_EXPORT_WITH_RC4_40_MD5
  TLS_RSA_EXPORT_WITH_DES40_CBC_SHA
  TLS_RSA_EXPORT_WITH_RC2_CBC_40_MD5
  SSL2_CK_RC4_EXPORT40
  SSL2_CK_RC2_EXPORT40
```

(continued)

Figure 4.10 (continued)

```

2 0.7508 (0.3829) S>C Handshake
  ServerHello
    session_id[32]=
      22 00 00 00 4b 66 29 fe 02 a7 f3 cc
      56 38 a6 4e 41 ad d8 fd 55 08 75 5d
      fe e3 41 02 fe 4e 27 62
    cipherSuite          TLS_RSA_EXPORT_WITH_RC4_40_MD5
    compressionMethod    NULL
  Certificate
    Subject
      C=IT
      ST=Milano
      L=Assago
      O=BANCO AMBROSIANO VENETO
      OU=INNOVAZIONE TECNOLOGICA
      OU=Terms of use at www.verisign.com
      RPA (c)99
      CN=HB.AMBRO.IT
    Issuer
      O=VeriSign Trust Network
      OU=VeriSign, Inc.
      OU=VeriSign International Server CA - Class 3
      OU=www.verisign.com
      CPS Incorp.by Ref. LIABILITY LTD.(c)97 VeriSign
    Serial          69 5a a3 c0 d8 8e bf ac 6b 64 ca cc
    ce ec 1b 59
  Extensions
    Extension: X509v3 Basic Constraints
      CA:FALSE
    Extension: 2.5.29.3
      value omitted
    Extension: Netscape Cert Type
      SSL Server
    Extension: X509v3 Extended Key Usage
      Netscape Server Gated Crypto, Microsoft Server Gated Crypto
    Extension: 2.16.840.1.113733.1.6.7
      value omitted
  Subject
    O=VeriSign Trust Network
    OU=VeriSign, Inc.
    OU=VeriSign International Server CA - Class 3
    OU=www.verisign.com

```

```

CPS Incorp.by Ref. LIABILITY LTD.(c)97 VeriSign
Issuer
  C=US
  O=VeriSign, Inc.
  OU=Class 3 Public Primary Certification Authority
Serial      23 6c 97 1e 2b c6 0d 0b f9 74 60 de
f1 08 c3 c3
Extensions
  Extension: X509v3 Basic Constraints
    CA:TRUE, pathlen:0
  Extension: X509v3 Key Usage
    Certificate Sign, CRL Sign
  Extension: Netscape Cert Type
    SSL CA, S/MIME CA
  Extension: X509v3 Extended Key Usage
    2.16.840.1.113733.1.8.1, Netscape Server Gated Crypto
  Extension: X509v3 Certificate Policies
    Policy: 2.16.840.1.113733.1.7.1.1
    CPS: https://www.verisign.com/CPS
    User Notice:
      Organization: VeriSign, Inc.
      Number: 1
      Explicit Text: VeriSign's Certification Practice Statement, ↵
        www.verisign.com/CPS, governs this certificate & is ↵
        incorporated by reference herein. SOME WARRANTIES DISCLAIMED ↵
        & LIABILITY LTD. (c)1997 VeriSign
ServerKeyExchange
ServerHelloDone

```

Client aborts first handshake and initiates a new one on the same connection, offering strong cipher suites.

```

3 0.7599 (0.0090) C>S  Handshake
  ClientHello
    Version 3.0
    cipher suites
      TLS_RSA_WITH_RC4_128_MD5
      TLS_RSA_WITH_RC4_128_SHA
      value unknown: 0x80  Microsoft proprietary cipher suite
      value unknown: 0x81  Microsoft proprietary cipher suite
      value unknown: 0x80  Microsoft proprietary cipher suite
      TLS_RSA_WITH_DES_CBC_SHA
      TLS_RSA_EXPORT_WITH_RC4_40_MD5
      TLS_RSA_EXPORT_WITH_DES40_CBC_SHA
      TLS_RSA_EXPORT_WITH_RC2_CBC_40_MD5
    compression methods
      NULL

```

(continued)

Figure 4.10 (continued)

```

4 1.1402 (0.3802) S>C    Handshake
   ServerHello
     session_id[32]=
       22 00 00 00 4b 66 29 fe 02 a7 f3 cc
       56 38 a6 4e 41 ad d8 fd 55 08 75 5d
       fe e3 41 02 fe 4e 27 62
     cipherSuite          TLS_RSA_WITH_RC4_128_MD5
     compressionMethod    NULL
   Certificate
     Subject
       C=IT
       ST=Milano
       L=Assago
       O=BANCO AMBROSIANO VENETO
       OU=INNOVAZIONE TECNOLOGICA
       OU=Terms of use at www.verisign.com
       RPA (c)99
       CN=HB.AMBRO.IT
     Issuer
       O=VeriSign Trust Network
       OU=VeriSign, Inc.
       OU=VeriSign International Server CA - Class 3
       OU=www.verisign.com
       CPS Incorp.by Ref. LIABILITY LTD.(c)97 VeriSign
     Serial              69 5a a3 c0 d8 8e bf ac 6b 64 ca cc
     ce ec 1b 59
     Extensions
       Extension: X509v3 Basic Constraints
         CA:FALSE
       Extension: 2.5.29.3
         value omitted
       Extension: Netscape Cert Type
         SSL Server
       Extension: X509v3 Extended Key Usage
         Netscape Server Gated Crypto, Microsoft Server Gated Crypto
       Extension: 2.16.840.1.113733.1.6.7
         value omitted
     Subject
       O=VeriSign Trust Network
       OU=VeriSign, Inc.
       OU=VeriSign International Server CA - Class 3
       OU=www.verisign.com
       CPS Incorp.by Ref. LIABILITY LTD.(c)97 VeriSign

```

```

Issuer
  C=US
  O=VeriSign, Inc.
  OU=Class 3 Public Primary Certification Authority
Serial      23 6c 97 1e 2b c6 0d 0b f9 74 60 de
f1 08 c3 c3
Extensions
  Extension: X509v3 Basic Constraints
    CA:TRUE, pathlen:0
  Extension: X509v3 Key Usage
    Certificate Sign, CRL Sign
  Extension: Netscape Cert Type
    SSL CA, S/MIME CA
  Extension: X509v3 Extended Key Usage
    2.16.840.1.113733.1.8.1, Netscape Server Gated Crypto
  Extension: X509v3 Certificate Policies
    Policy: 2.16.840.1.113733.1.7.1.1
    CPS: https://www.verisign.com/CPS
    User Notice:
      Organization: VeriSign, Inc.
      Number: 1
      Explicit Text: VeriSign's Certification Practice Statement, ↵
        www.verisign.com/CPS, governs this certificate & is ↵
        incorporated by reference herein. SOME WARRANTIES DISCLAIMED ↵
        & LIABILITY LTD. (c)1997 VeriSign
ServerHelloDone
5 1.1459 (0.0057) C>S  Handshake
ClientKeyExchange
  EncryptedPreMasterSecret[128]=
    7d 4f fb ae fb 3e 5c 5b 9f 72 82 86
    04 c7 7e 5e 72 e0 81 d3 8a 1a a0 10
    97 36 d3 ab 0a 8b d8 e3 ba 83 1b 1c
    8e 82 73 34 0a 4e 74 ec 0c 57 08 f7
    ce 61 13 cf 8b c2 7b 5e 8a 14 29 28
    94 5f 8a ff 93 42 59 93 12 ee 6d d4
    d6 15 c2 90 c1 b1 df 2a 73 1e fb 11
    bc e1 83 81 cd c7 4e ec 2c 1c e2 ba
    14 ab fa 7c 80 b1 e5 8e 52 9b 55 bf
    b7 84 b3 60 98 67 58 29 cd 50 ab 4d
    2e 3f 21 09 4a 16 a0 57
6 1.1459 (0.0000) C>S  ChangeCipherSpec
7 1.1459 (0.0000) C>S  Handshake Finished
8 1.4361 (0.2901) S>C  ChangeCipherSpec
9 1.4361 (0.0000) S>C  Handshake Finished

```

Figure 4.10 A connection using SGC

The first two records in the handshake are more or less the same as those in Figure 4.10. The client offers export cipher suites and the server provides its certificate showing that it's SGC-capable. (Although note that IIS transmits all of the handshake messages in a single record rather than in separate records as OpenSSL does. This is a matter of taste. There's no real technical reason to do one or the other.) However, instead of completing the connection, IE immediately responds in record 3 with a new ClientHello offering strong cipher suites. At this point, the old handshake is aborted and an entirely new one takes place. Note that IE is also offering the proprietary cipher suites 0x0080 and 0x0081. These are from a precursor to SGC used for Microsoft Money [Banes2000].

This seems like a clever trick, and it is. Unfortunately, it also violates the SSL specification, which has no provision for sending ClientHellos in the middle of a handshake. Thus, to preserve interoperability, Microsoft had to use a different extension from Netscape to signal that not only was a server allowed to use Step-Up, but that it could also do SGC. Note that the certificate in Figure 4.10 contains both the Netscape and Microsoft SGC extensions. Thus, when talking to this server, Step-Up only clients will do Step-Up but SGC-capable clients will do SGC. OpenSSL only added support for SGC in OpenSSL 0.9.5.

4.16 DH/DSS Details

```
struct {
    opaque DH_p<1..216-1>;
    opaque DH_g<1..216-1>;
    opaque DH_Ys<1..216-1>;
} ServerDHPParams;

ServerKeyExchange
    params
        DH_p[64] =
            da 58 3c 16 d9 85 22 89 d0 e4 af 75
            6f 4c ca 92 dd 4b e5 33 b8 04 fb 0f
            ed 94 ef 9c 8a 44 03 ed 57 46 50 d3
            69 99 db 29 d7 76 27 6b a2 d3 d4 12
            e2 18 f4 dd 1e 08 4c f6 d8 00 3e 7c
            47 74 e8 33
        DH_g[1] =
            02
        DH_Ys[64] =
            98 5a ee fc ce ac cf f1 05 cf 08 07
            63 18 dd 50 53 66 a5 b8 0b 88 4d 7e
            7d ea 11 3e 2a 99 63 e8 92 7a 56 cb
            f1 36 74 97 36 4a f0 3e 4e 29 3e a2
            e2 53 36 d8 9c a0 40 aa 8c fc eb c0
            93 b6 c3 e8
```

```
signature[47]=
  30 2d 02 14 78 bb 87 40 13 e4 8d e9
  73 16 4e 0c dd 1c 9e a8 bd 58 99 a1
  02 15 00 95 10 42 e1 cb b9 1d 26 34
  d4 5f b1 0d b8 66 ba 8c 61 20 c3
```

Ephemeral DH with DSS is conceptually very similar to ephemeral RSA. Only the details of the messages are different.

A DH/DSS ServerKeyExchange message contains Diffie-Hellman parameters signed with a DSS key. The provided parameters are the prime modulus (p), the generator for the group (g) and the server's public key (Y_s). Because DSS requires a 160-bit input, the DSS signature is computed over the sha_hash value only. Recall that DSS signatures consist of two large integers, called r and s . The SSL representation of these in the message is the DER encoding of an ASN.1 structure defined in ANSI X9.57 [ANSI1995].

```
Dss-Sig-Value ::= SEQUENCE {
    r    INTEGER,
    s    INTEGER
}
```

Note: The description of how to encode DSS signatures in the SSLv3 specification is insufficiently specific and Netscape interprets it differently from most other vendors. Instead of DER encoding r and s , Netscape merely concatenates them into a single 40-byte field. Thus, although Netscape implements DSS for client authentication, it is not interoperable with other implementations. Despite widespread agreement on the “right thing,” Netscape has refused to change their implementation, claiming that it complies with the SSLv3 specification and citing installed base. The TLS specification clears up this issue; all TLS implementations must DER encode DSS signatures.

ClientKeyExchange

```
struct {
    select (KeyExchangeAlgorithm) {
        case rsa: EncryptedPreMasterSecret;
        case diffie_hellman: DiffieHellmanClientPublicValue;
    } exchange_keys;
} ClientKeyExchange;

struct {
    select (PublicValueEncoding) {
        case implicit: struct {};
        case explicit: opaque DH_Yc<1..2^16-1>;
    } dh_public;
} DiffieHellmanClientPublicValue;
```

```
ClientKeyExchange
  DiffieHellmanClientPublicValue[64] =
    14 ea e2 18 c1 69 b3 60 fc ea c7 54
    f7 18 db b9 47 c7 cf 95 80 2a 32 b7
    0c 07 11 ab 7a 9d dc 0a 1c 82 a1 35
    23 1f 90 71 2a 94 6d d8 86 b4 e2 84
    e9 a6 a2 00 5e bb 82 09 a3 8a ba f2
    e8 29 87 61
```

Because Diffie-Hellman requires that the client and the server share a common set of parameters (g and p , found in the `ServerKeyExchange`), the client needs to send only its public key (Y_c) in the `ClientKeyExchange` message. Y_c is represented in the usual SSL style for big integers, high byte first.

Fixed DH Keys

Because the client has a DH key, it is possible for the client to have a certificate binding the key to the client's identity. This certificate can be used to perform client authentication, using the `rsa_fixed_dh` and `dss_fixed_dh` certificate types. These certificates are certificates containing DH keys signed with RSA and DSS respectively. The client sends its certificate in response to the `CertificateRequest` but does not send a `CertificateVerify`. Rather, it proves that it possesses the corresponding private key by being able to produce the `pre_master_secret` using its private key. Because the client's public key is present in the `Certificate` message, no public key is present in the `ClientKeyExchange` message (the implicit choice in the `DiffieHellmanClientPublicValue` structure).

Similarly, it is also possible for the server to have a static DH key that is contained in its certificate. The server's identity is proven by being able to reproduce the `pre_master_secret`. Neither of these operational modes is as common as the ephemeral DH modes using DSS.

4.17 FORTEZZA Details

The FORTEZZA support described in [Freier1996] does not work. It was apparently written without ever being tested. The various changes required for successful operation are described in [Relyea1996].

The FORTEZZA command interface has a number of properties that make FORTEZZA difficult to use with SSL. The most important one is that the card will not use arbitrary keys. Thus, keys must either be generated on the card by the card random number generator or received encrypted from another card. As a result, it's not possible to use the standard SSL key generation procedure to generate the FORTEZZA encryption keys.

The second important limitation is that the card must generate the IV. This requirement was imposed by the escrow feature. The escrowed session is carried in something called the *Law Enforcement Access Field* (LEAF). The LEAF contained the session key

used to encrypt the actual traffic. The session key was wrapped in the card key. In order to ensure that the LEAF is carried along with the ciphertext, it was embedded in the IV, resulting in a 24-byte IV. The card automatically extracts the actual 8-byte IV from the IV given to it.

In 1997, the NSA bowed to pressure over the escrow issue and removed the LEAF. However, for compatibility, the 24-byte IV remained with a dummy LEAF. Finally, on June 23, 1998, SKIPJACK and KEA were declassified.

```
struct {
    opaque r_s [128];
} ServerFortezzaParams;

struct {
    opaque y_c<0..128>;
    opaque r_c[128];
    opaque y_signature[40];
    opaque wrapped_client_write_key[12];
    opaque wrapped_server_write_key[12];
    opaque client_write_iv[24];
    opaque server_write_iv[24];
    opaque master_secret_iv[24];
    block-ciphered opaque encrypted_pre_master_secret[48];
} FortezzaKeys;
```

The ServerFortezzaParams structure is used to carry the server's random value `r_s` to the client. The FortezzaKeys structure is carried in the ClientKeyExchange message. It contains all the information that the server will need to derive the same keys as the client.

The `y_c` value contains the client's KEA public key if it's not in the client's certificate. If it is, this field is empty. The `r_c` value contains the client's random value. `y_signature` contains the client's signature over `y_c`. The specification is unclear on what the value of `y_signature` should be if the client's KEA public key is in its certificate. In KEA, the client and server public keys are combined with the random values to generate a *Token Encryption Key* (TEK) which can then be used to encrypt other keys.

`wrapped_client_write_key` and `wrapped_server_write_key` contain the encryption keys wrapped under the KEA-generated shared key. `client_write_iv` and `server_write_iv` contain the IVs for these keys (with the LEAF embedded in them). Finally, the FortezzaKeys structure contains the `pre_master_secret` encrypted under the shared key with an IV given in `master_secret_iv`. The `pre_master_secret` is used only to generate the MAC keys.

The FORTEZZA command interface imposes one other limitation: IVs are intended to be generated by the sender and read by the recipient. Thus, it's possible to load an IV onto the card when decrypting but not encrypting. However, both keys and IVs are

generated by the client. The result is that the first block of data encrypted by the server uses an essentially random IV—which the client has no way of knowing. In order to synchronize the CBC state, the server first transmits a dummy encrypted block that is discarded by the client. From then on, the cipher states will be synchronized.

Note that the encryption of this dummy block means that the server never needs to use the `server_write_iv`. [Relyea1996] says that the server implementation should load the IV in decryption mode to verify the IV and then change over to encryption mode. This step is not required for interoperability. Its purpose is to verify that IV correctly matches the key, thereby preventing a rogue client from using a fake IV (with a bogus LEAF) and thereby thwarting the escrow provisions.

The inability to use random keys also means that ordinary session resumption will not work, since there is no way to use the newly generated keys. [Relyea1996] describes two options:

- Don't resume sessions at all.
- Resume but continue encrypting and decrypting as if the data were a continuation of the previous session. Note that if this approach is taken it is not possible to have two simultaneous connections with a given session, as it ordinarily would be. This is problematic for systems such as HTTPS which use a number of simultaneous connections.

Only two FORTEZZA cipher suites were defined, one with SKIPJACK and one with NULL encryption. Figure 4.11 shows them.

Cipher Suite	Authentication	Key Exchange	Encryption	Digest	Number
SSL_FORTEZZA_DMS_WITH_NULL_SHA	DSA	KEA	NULL	SHA	0x001c
SSL_FORTEZZA_DMS_WITH_FORTEZZA_CBC_SHA	DSA	KEA	SKIPJACK	SHA	0x001d

Figure 4.11 FORTEZZA cipher suites

4.18 Error Alerts

In Chapter 3, we discussed alerts in general without discussing the specific alert cases other than `close_notify`. All the other alerts indicate errors of some kind. This section presents a detailed discussion of those alerts.

Recall that alerts may have a level of warning or fatal. A fatal alert always terminates the connection, whereas implementations may continue past a warning alert. While in theory implementations can continue past a non-fatal alert, in practice even warning alerts indicate that one side requested something that the other side couldn't fulfill and so they usually result in connection closure.

The specification requires some alerts to be fatal whereas others may be either fatal or warning. An implementation which receives an alert which should be fatal but which is incorrectly tagged as warning should nevertheless treat it as fatal. In practice, since

many implementations treat all alerts as fatal, implementations should refrain from sending warning alerts unless they're willing to have the other side terminate the connection.

This section describes each SSL error alert in detail. Although some of the alerts could signal an attack in progress, if you're debugging your SSL implementation, they most likely represent bugs. Where appropriate we'll attempt to identify the common implementation errors that these alerts might indicate. Reading this section may allow you to determine what's happened when your implementation receives or generates one of these alerts. See Figure 4.12 for a list of all the 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

Figure 4.12 All SSL/TLS alerts

unexpected_message

The `unexpected_message` alert signals that an inappropriate message was received. This alert almost always signals that one of the implementations is broken and generating inappropriate messages or sending messages in the incorrect order. You might see this message often when debugging an SSL implementation but should never see it in working systems. As a special case, a non SGC-capable server might send an `unexpected_message` to a client attempting SGC. This alert must always be sent as fatal.

bad_record_mac

The `bad_record_mac` alert is transmitted if an implementation receives a record with a bad MAC. This might represent an attack. Thus, this alert must be treated as fatal.

If you're testing your implementation and you receive a `bad_record_mac` alert, it most likely indicates an error in your MAC code. However, if the alert is sent in response to a Finished message, it could also represent an error in key derivation. Even if the encryption keys are also wrong, the decryption may appear to succeed. However, the MAC will fail. In fact, even if the MAC key is correct but the encryption key is wrong, the error will still most likely appear in the MAC check (see the `decryption_error` alert for a discussion of decryption errors).

Finally, some SSLv3 implementations use the `bad_record_mac` message to signal an error in verifying the Finished message itself. In TLS, failure to verify the Finished is signalled with the `decrypt_error` alert. Thus, when debugging a `bad_record_mac` error, it's important to consider other culprits besides the MAC computation itself.

decryption_failed

The `decryption_failed` signals that the ciphertext couldn't be decrypted. This error must be fatal. This error is really only relevant with block ciphers. RC4 decryption always appears to succeed whatever the decryption key. The output is simply wrong.

Block cipher decryption can fail in a number of ways. First, the block cipher input length can be something other than a multiple of the block length. This is an obvious message formatting error. Also, after the data is decrypted, the padding check may reveal errors resulting from incorrect padding or the use of the wrong encryption key.

If the wrong encryption key is used, the padding bytes will be essentially random. However, it's possible that they will still appear correct, particularly if the implementation chooses to examine only the length byte and then remove the padding without examining it. (This procedure is safe because the data is covered by the MAC anyway.) Note that the padding does not have to actually be the same as the original padding. It merely has to be consistent.

In this case, the receiver will have the wrong plaintext but this will only be discovered when the MAC check and a `bad_record_mac` alert will be received. Obviously, since RC4 decryption always appears to succeed, errors in RC4 keying always show up in the MAC check.

record_overflow

This alert indicates that the record was larger than allowed by the SSL specification. This message must be fatal. In practice, this message almost always represents a bug in the SSL implementation since an attack would likely be caught in the MAC check. Note that early versions of Internet Explorer generated incorrectly long records and so it's not uncommon to see an unsuspecting implementation generate this alert when talking to IE. This alert is new with TLSv1.

decompression_failure

This alert indicates that a record could not be decompressed. It is always fatal. However, since compression is almost never used, this error will rarely be seen in practice.

handshake_failure

The `handshake_failure` alert is the all-purpose alert used in SSLv3 to indicate problems handshaking. TLS introduced a number of finer-grained alerts to indicate what actually went wrong. In TLS, the `handshake_failure` alert is only used when no common cipher can be negotiated.

When debugging a `handshake_failure` alert, you need to examine which stage of the handshake went wrong and from there determine what happened. If you received a `handshake_failure` alert after sending a series of messages at once, it's often useful to send each message individually, checking for the alert after each one using a network sniffer. This alert must always be fatal.

no_certificate

The `no_certificate` alert is an SSLv3-only alert sent to indicate that "no appropriate certificate is available." It has been replaced in TLS with `certificate_unknown` which serves essentially the same purpose.

bad_certificate

The `bad_certificate` alert indicates that a certificate was corrupt or the signatures were wrong or some similar error. In theory this alert may be a warning. An implementation might choose to ignore the bad certificate but issue the alert as a warning. However, in practice implementations either silently accept the certificate or send an alert. Thus, this alert is really always fatal. This comment applies to the next four alerts as well. Interestingly, Internet Explorer doesn't use this alert at all. Instead, it simply closes the connection. We'll see an example of this in Chapter 8.

unsupported_certificate

This alert means that the implementation received a certificate which was of an unsupported type. Since SSL only supports X.509 certificates, this alert means that an unsupported algorithm was used. Because the algorithm used to sign certificates is not specified in the cipher suite, it's possible to have (for instance) a certificate for an RSA key signed with a DSA key. An implementation which did not support DSA would send an `unsupported_certificate` error. However, in practice such mixed certificate regimes nearly never occur and an `unsupported_certificate` error represents a bug in one of the implementations.

revoked_certificate

This alert indicates that the sender received a certificate that was revoked. Most real life SSL implementations do not currently support revocation so this alert almost never appears.

certificate_expired

This alert indicates that one of the certificates has expired. Implementations shouldn't send expired certificates, but sometimes they are misconfigured. This alert is the result.

certificate_unknown

This alert is a catch-all certificate error introduced in TLSv1. Certificate errors that don't fall into any of the previous classes or the `unknown_ca` class described below generate a `certificate_unknown` error. SSLv3 implementations would generate a `bad_certificate` alert for those conditions.

illegal_parameter

The `illegal_parameter` alert indicates that one of the handshake fields was out of range or inconsistent with some other fields. The handshake obviously can't complete so this error is fatal.

unknown_ca

The `unknown_ca` alert is sent when an implementation receives a certificate signed by a CA it doesn't recognize or can't find the certificate for. This alert must be fatal. Note that this treatment is inconsistent with the other certificate errors which can be sent as warnings. I know of no reason for the inconsistency. In my opinion, best practice is to either silently accept certificates or to reject them with an alert. Note that in this context *silent* refers only to SSL messages. An implementation might accept a bad certificate only after prompting the user or sending error messages to a logging facility. The

unknown_ca message is new with TLS. SSLv3 implementations send the no_certificate alert instead.

access_denied

This alert means that although a valid certificate was received the identity in the certificate fails access control checks so the connection is being rejected. This alert is always fatal. I know of no implementation which generates this alert.

decode_error

This alert indicates that a message couldn't be decoded because one of the fields was out of range or the length was incorrect. In most cases this represents some sort of implementation error. This alert is always fatal. decode_error was introduced in TLS and it's not clear what to use with SSLv3. OpenSSL and PureTLS both use handshake_failure, presumably on the theory that most of these errors occur in the handshake.

decrypt_error

The decrypt_error alert indicates that one of the handshake cryptographic operations failed. The likely cases are failure to decrypt a ClientKeyExchange, verify a signature, or check a Finished message. This alert is always fatal. decrypt_error was introduced with TLSv1. SSLv3 implementations usually use handshake_failure instead. Note that if you receive a decrypt_error in response to a Finished, this means that your encryption and MAC keys and code are correct but that your Finished computation is broken.

export_restriction

This alert indicates that one of the implementations tried to violate export restrictions. The example provided in RFC 2246 is trying to use a 1024 bit ephemeral key with RSA_EXPORT key exchange. This is a new alert type for TLSv1 and isn't seen much. Note that this is not the alert that you will see when an implementation that supports only strong ciphers tries to talk to one that supports only weak ones. That will be either insufficient_security or handshake_failure. This alert is always fatal.

protocol_version

The protocol_version alert is sent by the server to indicate that the client used an unrecognized protocol version. Because the client can only supply the maximum version that it supports, the server will only send this alert when the client asked for a version older than it will support. For instance, if the client offered only SSLv3 and the server supported only TLS, the server would generate a protocol_version alert. Since essentially all TLS implementations are also SSLv3 implementations, this alert is only seen if one

of the sides has been explicitly configured to support only TLS or if there is a bug. There is no value in configuring for TLS-only support.

Note that if the server only supports a version older than the client offers, it will *not* send a `protocol_version` alert. Instead, it will simply respond with its version in the `ServerHello`. If the client does not want to support the server's version choice, it will have to send an alert (presumably `protocol_version` but the SSLv3 and TLS specs don't specify). The `protocol_version` alert is always fatal. The `protocol_version` alert was added in TLS. SSLv3 implementations usually use `handshake_failure` for this purpose.

insufficient_security

The `insufficient_security` alert is used to indicate that the cipher suites offered by the client were weaker than required by the server. This alert is new with TLS. SSLv3 implementations use `handshake_failure`. Since the `insufficient_security` alert indicates failure to negotiate a cipher suite, it is always fatal.

internal_error

The `internal_error` alert is used to indicate that the transmitting implementation encountered some sort of internal error such as a memory allocation or hardware failure. This alert is new with TLS so SSLv3 implementations need to use some other error. The `internal_error` alert is always fatal.

user_cancelled

The `user_cancelled` alert is used when the user cancels the handshake for some reason. This alert is new for TLS. RFC 2246 says that it should be used as a warning, which is confusing—if the user cancels the handshake then the connection will be unsuccessful so the alert should be fatal.

no_renegotiation

As we'll see in Chapter 4, SSL allows either client or server to initiate a new handshake at any time other than during a handshake. The client initiates renegotiation by sending a new `ClientHello`. The server initiates renegotiation by sending a `HelloRequest` message. RFC 2246 says that the client can silently refuse to renegotiate or send a `no_renegotiation` alert. RFC 2246 is unclear on how the server should respond to an unwanted `ClientHello` but presumably it should send a `no_renegotiation` alert as well.

The `no_renegotiation` alert is new for TLS. SSLv3 implementations simply ignore unwanted renegotiation requests. Since most existing TLS implementations are actually upgraded SSLv3 implementations, they typically silently ignore renegotiation rather than sending the `no_renegotiation` alert. In any case, the `no_renegotiation` alert is always a warning.

4.19 SSLv2 Backward Compatibility

SSLv2 had already been widely deployed when SSLv3 was invented. Thus, one of the requirements for SSLv3 was for SSLv3 clients and servers to automatically talk to SSLv2 clients and servers. However, the message formats were totally different. (See Appendix B for a discussion of SSLv2.) It's fairly straightforward for servers to auto-detect whether they are talking to an SSLv3 or SSLv2 client. The SSLv2 and SSLv3 length bytes are in different locations and SSLv3 records appear to be unreasonably long when interpreted as SSLv2 handshake messages. Thus, the server can wait for the client's first message and respond appropriately.

Clients, however, must speak first and so cannot auto-detect. Thus, the SSLv3 specification contains support for sending the ClientHello in a format that was backward-compatible with SSLv2. In essence, the backward-compatible ClientHello maps all the SSLv3 fields onto SSLv2 fields in the SSLv2 CLIENT-HELLO. Thus, when a server receives a backward-compatible CLIENT-HELLO, it can either interpret it as an SSLv2 message and continue with the SSLv2 handshake or translate the fields to SSLv3 values and perform the SSLv3 handshake. Figure 4.13 shows the SSLv2 CLIENT-HELLO.

```

char MSG-CLIENT-HELLO
char CLIENT-VERSION-MSB
char CLIENT-VERSION-LSB
char CIPHER-SPECS-LENGTH-MSB
char CIPHER-SPECS-LENGTH-LSB
char SESSION-ID-LENGTH-MSB
char SESSION-ID-LENGTH-LSB
char CHALLENGE-LENGTH-MSB
char CHALLENGE-LENGTH-LSB
char CIPHER-SPECS-DATA[(MSB<<8)|LSB]
char SESSION-ID-DATA[(MSB<<8)|LSB]
char CHALLENGE-DATA[(MSB<<8)|LSB]

```

Figure 4.13 SSLv2 CLIENT-HELLO

The CLIENT-HELLO shown is in the SSLv2 specification language of [Hickman1995]. Unlike SSLv3, SSLv2 places all the object lengths in one part of the message and all the objects in another. Also, the high and low bytes of lengths are represented individually. Thus, the length of the CIPHER-SPECS-DATA field is given by computing

$$(\text{CIPHER-SPECS-LENGTH-MSB} \ll 8) \mid \text{CIPHER-SPECS-LENGTH-LSB}$$

All of the SSLv3 handshake parameters can be mapped to SSLv2 values.

Version

The SSLv2 version number is 2 (MSB=0, LSB=2). To represent SSLv3 or TLS compatibility, we use the appropriate version numbers from the SSLv3 or TLS Version field: 0x300 and 0x301 for SSLv3 and TLS, respectively.

Random

The SSLv2 CHALLENGE value is used as part of the SSLv2 key generation procedure in much the same way as the ClientRandom is used by SSLv3. It's randomly generated, so it can be used as a ClientRandom value. However, it's allowed to be between 16 and 32 bytes, whereas the SSLv3/TLS ClientRandom must be 32 bytes. To convert a CHALLENGE to a ClientRandom value, it is padded with zeros on the left.

Session ID

Whenever resuming an SSLv3 session, the SSLv3 ClientHello must be used. Thus, the SESSION-ID field should always be empty and the length zero.

Cipher Suites

SSLv2 CIPHER-SPECSs are analogous to SSLv3 CipherSuites. They specify the encryption and digest algorithms. (SSLv2 only supports RSA, so they don't specify key exchange or signature.) Thus, in backward-compatibility mode, the CLIENT-HELLO should contain both the SSLv2 and SSLv3/TLS ciphers that the client supports. SSLv2 CIPHER-SPEC values are 3 bytes, so 2-byte SSLv3 values are represented with a zero first byte.

It's also possible to use SSLv2 ciphers when using the backward compatibility handshake. Some of these ciphers correspond to SSLv3/TLS ciphers, but some do not. For instance, SSLv2 has an RC2-128 mode. SSLv3 does not define such a cipher suite. However, it's possible for two SSLv3 clients to negotiate it if they are using the SSLv2 backward compatible handshake. All these cipher suites use RSA for signature and key exchange. Figure 4.14 lists these ciphers.

Cipher Spec	Encryption	Digest	Number
SSL_CK_RC4_128_WITH_MD5	RC4_128	MD5	0x010080
SSL_CK_RC4_128_EXPORT40_WITH_MD5	RC4_40	MD5	0x020080
SSL_CK_RC2_128_CBC_WITH_MD5	RC2_128_CBC	MD5	0x030080
SSL_CK_RC2_128_CBC_EXPORT40_WITH_MD5	RC2_40_CBC	MD5	0x040080
SSL_CK_IDEA_128_CBC_WITH_MD5	IDEA_CBC	MD5	0x050080
SSL_CK_DES_64_WITH_MD5	DES_CBC	MD5	0x060080
SSL_CK_DES_192_EDE3_CBC_WITH_MD5	DES_EDE3_CBC	MD5	0x070080

Figure 4.14 SSLv2 compatibility cipher suites

Rollback Protection

SSL uses PKCS #1 padding to encrypt data using RSA. However, in order to protect against an active attack in which the attacker forces negotiation back to SSLv2, SSLv3- and TLS-capable client implementations must use the value 03 for the last 8 bytes of the padding when doing SSLv2. The server can check this to make sure that it really is talking to an SSLv2-only client (which would fill these 8 bytes with random data). Note that this unusual padding will not appear and does not need to be checked if doing an SSLv3 handshake. The purpose of all this is to prevent an attacker from forcing a client/server pair back to the (weaker) SSLv2 from SSLv3/TLS.

Compatibility

In order for the backward-compatibility handshake to work correctly, an SSLv2-only server must simply negotiate to SSLv2 if it receives a higher version in the CLIENT-HELLO. Unfortunately, the SSLv2 specification wasn't very clear on how servers should handle CLIENT-HELLO messages with version numbers higher than they support. This problem was made worse by the fact that Netscape's SSLREF reference implementation simply rejected connections with higher version numbers. Thus, it's not guaranteed that all SSLv2 servers will respond correctly to the backward-compatible handshake, although the vast majority will.

4.20 Summary

This chapter concludes our detailed coverage of SSL. Each of these modes offers some benefit over the traditional RSA server-only mode discussed in Chapter 3.

Session resumption allows a client/server pair to reuse keying material between connections. This makes subsequent handshakes between a client/server pair faster.

Client authentication uses the client's public key to prove the client's identity to the server. Traditionally, SSL connections have anonymous clients, but with client authentication, clients as well as servers can authenticate using certificates.

Ephemeral RSA allows a server to have a single long (>512 bits) RSA key but still to communicate with exportable clients which require 512-bit RSA keys, by generating a temporary 512-bit key and signing it with the long-term key.

Server Gated Cryptography allows exportable clients to use strong cryptography with specifically designated servers. These servers must have special certificates that indicate that they are SGC eligible.

DSS and DH cipher suites make it possible to implement SSL in the United States without infringing on the RSA patent. Support for DSS and DH has been mandated for TLS.

Elliptic Curve ciphers may be faster than either DH/DSS or RSA, but they have not yet been standardized by IETF as part of TLS.

Kerberos keys can be used to exchange keying material between client and server, while keeping the rest of SSL the same. This makes it possible to bootstrap a Kerberos infrastructure to authenticate SSL connections.

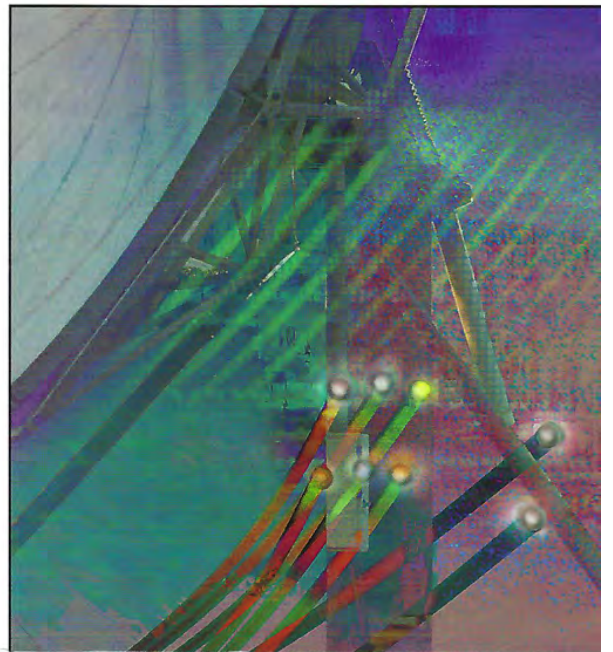
SSLv3 is backward-compatible with SSLv2. The SSLv3 ClientHello can be cast as an SSLv2 CLIENT-HELLO, thus permitting SSLv3 agents which also speak SSLv2 to communicate with SSLv2-only agents.

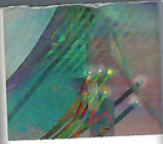
ATTACHMENT A-2

SSL and TLS

Designing and Building Secure Systems

Eric Rescorla





SSL and TLS

Rescorla

TK
5105.59
.R47
2001



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

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@pearsoned.com

Visit us on the Web at www.awl.com/cseng

TK
5105.59
R47
2001

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

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

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 <code>select()</code>	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 <code>CONNECT</code>	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	
11.12	CMS	389
11.13	Message Format	390
11.14	Cryptographic Options	391
11.15	Putting It All Together: S-HTTP	392
11.16	S-HTTP versus HTTPS	393
11.17	S/MIME	397
11.18	Basic S/MIME Formatting	400
11.19	Signing Only	401
11.20	Algorithm Choice	401
11.21	Putting It All Together: S/MIME	403
11.22	Implementation Barriers	405
11.23	S/MIME versus SMTP/TLS	406
11.24	Choosing the Appropriate Solution	407
11.25	Summary	408
		410
Appendix A. Example Code		411
A.1	Chapter 8	
A.1.1	C Examples	411
A.1.2	Java Examples	411
A.2	Chapter 9	425
A.2.1	HTTPS Examples	431
A.2.2	mod_ssl Session Caching	431
		436
Appendix B. SSLv2		455
B.1	Introduction	
B.2	SSLv2 Overview	455
B.3	Missing Features	455
B.4	Security Problems	457
B.5	PCT	458
B.6	What about SSLv1?	461
		463
Bibliography		465
Index		475

Preface

The *Secure Sockets Layer* (SSL) is by far the most widely deployed security protocol in the world. Essentially every commercial Web browser and server supports secure Web transactions using SSL. When you buy online using “secure” Web pages (an estimated 20 billion dollars’ worth of such transactions will occur in 2000), you’re almost certainly using SSL.

Although its most common use is for securing Web traffic, SSL is actually quite a general protocol suitable for securing a wide variety of kinds of traffic. File transfer (FTP), remote object access (RMI, CORBA, IIOP), e-mail transmission (SMTP), remote terminal service (Telnet) and directory access (LDAP) are just some of the applications that have already been secured with SSL or its successor, *Transport Layer Security* (TLS).

The effort to secure all these protocols has taught us a number of significant lessons. First, doing a good job of using SSL/TLS to secure a protocol requires having a fairly deep knowledge of how it works. It is not possible to simply treat SSL/TLS as a black box that somehow magically provides security when used.

Second, although each application is slightly different, there seems to be a set of security problems that are common to every application you wish to secure. For instance, we usually need to figure out some way for the insecure and secure versions of an application protocol to coexist. Although there aren’t cookie-cutter solutions to these problems, the security community is starting to develop a common set of techniques for solving these problems using SSL/TLS.

These techniques can often be applied to a new application protocol with minimal modification. In essence, we’ve developed a set of *design patterns* for securing protocols. Much of the work of securing a system is in recognizing which pattern most closely matches the system you’re working with and then using the appropriate techniques.

The purpose of this book, then, is to address both of these needs. After reading this book, you should know most if not all of what you need to know in order to design secure systems using SSL/TLS. You’ll know enough about SSL/TLS to understand what security features it can deliver and what it can’t deliver. Further, you’ll be familiar with the common design patterns for using SSL/TLS and be ready to apply them to new situations.

xvii

What This Book Provides

This book is intended for anyone who wants to understand and use SSL/TLS.

- For designers, it provides information on designing systems that use SSL/TLS as well as a library of the techniques that have already been used.
- For programmers who program with SSL/TLS, it provides information on what your libraries are doing under the covers and what those functions you're calling are really doing. Understanding these details is critical for obtaining acceptable and predictable application performance.
- For SSL/TLS implementors it acts as an adjunct to the standard, explaining obscure sections and describing both common practice and why things are the way they are.

Intended Audience

This book assumes a basic familiarity with how the TCP/IP protocols work. Readers who are unfamiliar with TCP/IP would be best served to consult one of the many fine books describing TCP/IP. *TCP/IP Illustrated, Volume 1* [Stevens1994] is a good choice. RFC 791 [Postel1991a], RFC 792 [Postel1991b], and RFC 793 [Postel1991c] provide the definitive reference for TCP/IP. Although some of this book will be understandable without a deep understanding of TCP/IP, much of the discussion of performance will be difficult to follow without an understanding of TCP behavior.

Because SSL/TLS is a cryptographic protocol, properly understanding it requires at least basic familiarity with cryptographic algorithms, including public key cryptography, symmetric cryptography, and digest algorithms. Chapter 1 provides an introduction to cryptography and communications, but space is too limited to do a complete job. We attempt to cover the requisite cryptographic details for understanding SSL/TLS; however, readers interested in a broader understanding of the cryptographic issues should consult a cryptography text such as [Schneier1996a] or [Kaufman1995].

Organization of the Book

This book is written in two halves, matching the two primary goals described previously: understanding the protocol and understanding how to use it. The first half, Chapters 1–6, is devoted to describing SSL and TLS. We concern ourselves with the technical details of how they work and their security and performance properties in isolation.

The second half of the book, Chapters 7–11, covers the design of application protocols and systems that use SSL/TLS for security. First we describe general guidelines for using SSL/TLS and then we discuss several protocols that have already been secured using SSL/TLS.

Chapter 1 — Security Concepts provides an introduction to cryptography and communications security, with an eye toward its use in SSL/TLS. If you're already familiar with communications security, you may want to skip this chapter. If, on the other hand, you're not familiar with security, you'll want to read this chapter carefully so you don't get lost later.

How

Chapter 2 — Introduction to SSL is a broad overview of the history of SSL/TLS and what sorts of security features it provides. We also provide a snapshot of the status of SSL/TLS-secured protocols as of the time of this writing.

Chapter 3 — Basic SSL covers the most common SSL/TLS operational mode. We describe an entire SSL/TLS connection from start to finish. This chapter should give you a very good idea of how SSL/TLS works in practice. All the other operational modes can be easily understood once you understand this chapter.

Chapter 4 — Advanced SSL covers the rest of the major operational modes. We cover session resumption, client authentication, and a number of algorithms that are only now seeing deployment with SSL/TLS, such as DH/DSS and Kerberos.

Chapter 5 — SSL Security describes the security benefits that SSL offers as well as (even more important) those it doesn't offer. Whereas previous chapters mostly focused on describing how things work, this chapter focuses on what you need to do to make a system that uses SSL/TLS secure.

Chapter 6 — SSL Performance describes the performance profile of TLS-based systems. It's been widely observed that security imposes significant performance demands on systems, but it's not widely understood that this impact is limited to certain parts of the protocol. We'll discuss these issues with an eye to getting better performance while preserving good security.

Chapter 7 — Designing with SSL is a guide to using SSL/TLS to secure application layer protocols. We focus on identifying the required security properties and on well-understood design techniques for satisfying these properties.

Chapter 8 — Coding with SSL discusses the common programming idioms required to write software that uses SSL/TLS. We provide complete sample programs in C and Java using the OpenSSL and PureTLS toolkits.

Chapter 9 — HTTP over SSL describes the application that started it all. SSL was originally designed by Netscape to work with HTTP, and we cover both the traditional way of doing things and the replacement that's currently being proposed.

Chapter 10 — SMTP over TLS describes the use of TLS to secure the *Simple Mail Transport Protocol* (SMTP), which is used for transporting e-mail. SMTP is a bad match for TLS, and this chapter illustrates some of the limitations of SSL and TLS.

Chapter 11 — Contrasting Approaches is devoted to describing other alternatives to securing your applications. SSL/TLS isn't always the best solution and part of knowing how to use a protocol is knowing when not to. This chapter tries to give you a perspective on your other choices. We discuss IPSEC, S-HTTP, and S/MIME as alternatives to SSL/TLS.

How to Read This Book

This book is suitable for a number of audiences of different technical abilities and requirements. You should read any section that interests you, but depending on your needs you may want to focus on specific sections.

Protocol Designers

If you're designing a new application-level protocol or securing an existing protocol with SSL/TLS, you should read the first parts of Chapters 1–6 so that you have a good general understanding of how SSL/TLS works. Then carefully read Chapter 7 for a guide to SSL/TLS design principles. You can skip Chapter 8 unless you intend to implement your design, but be sure to read Chapters 9 and 10 so you can see real-world examples of how SSL/TLS should and should not be used in practice. Finally, before you start to design, read Chapter 11 to make sure that SSL/TLS is appropriate for your design and that you wouldn't be better served by using another security protocol.

Application Programmers

If you're writing an application that uses a preexisting SSL/TLS toolkit you can safely read only the first parts of Chapters 1–6. You should also read the summaries at the end of each chapter. These sections discuss SSL/TLS and SSL/TLS implementation techniques in overview form. This will provide enough information to understand what your SSL/TLS toolkit is doing. You should carefully read Chapters 7 and 8, paying special attention to the programming techniques discussed in Chapter 8. If you are implementing HTTP or SMTP over SSL, you should also read the chapters that deal with those protocols.

SSL/TLS Implementors

If you're implementing SSL/TLS from scratch, you should read the entire book. If you're already familiar with cryptography you can skip Chapter 1; however, if you don't have a detailed knowledge of cryptography you should read the entire chapter. You should pay particular attention to Chapters 2–6, which provide a detailed description of SSL/TLS and of the various implementation techniques required to produce a fast and secure implementation.

Just Curious

If you just want to learn about SSL/TLS you can skip around in the book. If you don't already know about cryptography, read all of Chapter 1. Then read Chapters 2–6 so you know how SSL/TLS works. Then you can read as much or as little of the rest of the book as interests you. It's probably worth reading Chapter 11 to get some perspective on how SSL/TLS compares to other security protocols.

SSL/TLS Versions

By now you've no doubt gotten tired of seeing the name SSL/TLS. We've been using it to avoid being specific about exactly which version we mean. There are currently two

versions of SSL in wide deployment: SSL version 2 (SSLv2) and SSL version 3 (SSLv3). TLS, a modification of SSLv3, was standardized by the *Internet Engineering Task Force* (IETF) in 1999. Despite what you might think from the names, SSLv2 and SSLv3 are completely different protocols, and TLS is extremely similar to SSLv3. SSLv2 is essentially obsolete, and TLS isn't really in wide deployment as of this writing. In general, we'll use the term SSL to refer to SSLv3/TLS interchangeably. When we mean one or the other, we'll specify it explicitly. In the few instances when we're talking about SSL version 2, we'll use SSLv2.

Typographical Conventions

This book contains a number of network traces of real SSL or TLS sessions. In the presentation of such traces, we use a constant-width font to show the output of the program (CONSTRUCTED), and an italic font for comments inserted afterwards (*Comment*). Where the network trace displays a hexadecimal representation of the protocol data, we render that in a bold constant-width font (**01 02 03**). When we display encrypted data in plaintext form, we'll use an italic constant-width font (*data*).

Excerpts from standards (e.g., Internet RFCs) and protocol structure definitions in the main text are rendered in a sans serif font: (helvetica). In figures, we use Times for readability. Code snippets are rendered in a constant-width font (`int`). In a few special cases we'll need to wrap long lines, in which case we'll use the symbol ¶ at the end of the wrapped line to indicate that the text continues.

Historical notes and asides will be indented and rendered in this smaller font.

Network Traces

The network traces in this book are all traces of real sessions, mostly captured on the author's home ethernet. A variety of client and server programs were used, including OpenSSL, Netscape Navigator, Internet Explorer, and qmail. The traces were captured using the `tcpdump` program and stored to disk. The author's `ssldump` SSL decoding package was used to generate the traces shown in the text. `tcpdump` can be obtained from <http://www.tcpdump.org/>. `ssldump` can be obtained from <http://www.rtfm.com/ssldump/>.

Source Code

This book contains a number of snippets of source code. The source code in Chapters 8 and 9 were written by the author and are Copyright © 1999–2000 Eric Rescorla. They may be used and reproduced for any purpose and without fee but no warranty of any kind is provided. Machine-readable versions are available from the author's Web site at <http://www.rtfm.com/sslbook/examples>.

The Java source code in Chapter 5 is part of the PureTLS Java SSL/TLS implementation. It may be obtained at <http://www.rtfm.com/puretls>. This source code is covered by the following copyright.

Copyright (C) 1999, Claymore Systems, Inc.
All Rights Reserved.

ekr@rtfm.com Tue May 18 09:43:47 1999

This package is a SSLv3/TLS implementation written by Eric Rescorla
<ekr@rtfm.com> and licensed by Claymore Systems, Inc.

Redistribution and use in source and binary forms, with or without
modification, are permitted provided that the following conditions
are met:

1. Redistributions of source code must retain the above copyright
notice, this list of conditions and the following disclaimer.
2. Redistributions in binary form must reproduce the above copyright
notice, this list of conditions and the following disclaimer in the
documentation and/or other materials provided with the distribution.
3. All advertising materials mentioning features or use of this software
must display the following acknowledgement:
This product includes software developed by Claymore Systems, Inc.
4. Neither the name of Claymore Systems, Inc. nor the name of Eric
Rescorla may be used to endorse or promote products derived from this
software without specific prior written permission.

THIS SOFTWARE IS PROVIDED BY THE REGENTS AND CONTRIBUTORS 'AS IS' AND
ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE
IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE
ARE DISCLAIMED. IN NO EVENT SHALL THE REGENTS OR CONTRIBUTORS BE LIABLE
FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL
DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS
OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION)
HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT
LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY
OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF
SUCH DAMAGE.

The SSL session caching example in Appendix A is from Ralf S. Engelschall's (rse's) Mod_ssl package, available at <http://www.modssl.org/>. The version used is 2.6.1-1.3.12. It is subject to the following copyright.

```
=====
Copyright (c) 1998-1999 Ralf S. Engelschall. All rights reserved.
```

```
Redistribution and use in source and binary forms, with or without
modification, are permitted provided that the following conditions
are met:
```

1. Redistributions of source code must retain the above copyright notice, this list of conditions and the following disclaimer.
2. Redistributions in binary form must reproduce the above copyright notice, this list of conditions and the following disclaimer in the documentation and/or other materials provided with the distribution.
3. All advertising materials mentioning features or use of this software must display the following acknowledgment:
"This product includes software developed by
Ralf S. Engelschall <rse@engelschall.com> for use in the
mod_ssl project (<http://www.modssl.org/>)."
4. The names "mod_ssl" must not be used to endorse or promote products derived from this software without prior written permission. For written permission, please contact rse@engelschall.com.
5. Products derived from this software may not be called "mod_ssl" nor may "mod_ssl" appear in their names without prior written permission of Ralf S. Engelschall.
6. Redistributions of any form whatsoever must retain the following acknowledgment:
"This product includes software developed by
Ralf S. Engelschall <rse@engelschall.com> for use in the
mod_ssl project (<http://www.modssl.org/>)."

```
THIS SOFTWARE IS PROVIDED BY RALF S. ENGELSCHALL ``AS IS'' AND ANY
EXPRESSED OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE
IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR
PURPOSE ARE DISCLAIMED. IN NO EVENT SHALL RALF S. ENGELSCHALL OR
HIS CONTRIBUTORS BE LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL,
SPECIAL, EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT
NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES;
```

LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION)
HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT,
STRICT LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE)
ARISING IN ANY WAY OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED
OF THE POSSIBILITY OF SUCH DAMAGE.

=====

Acknowledgments

It's impossible to write any book, much less a technical book, without the assistance of many people. As is customary, I'd like to use this space to acknowledge some of them.

My technical reviewers not only kept me honest but also ensured that my writing was as clear as I could make it. Joshua Ball, Joe Balsama, Douglas Barnes, Debasish Biswas, Andrew Brown, Robert Bruen, Megan Conklin, Russ Housley, Paul Kocher, Brian Korver, Chris Kostick, Marcus Leech, Robert Lynch, Joerg Meyer, D. Jay Newman, Tim Newsham, Stacey O'Rourke, Radia Perlman, Mark Schertler, Win Treese, Tom Weinstein, and Tom Woo all reviewed sections of the manuscript.

A number of people generously provided answers to technical questions that I had while writing, often without even knowing that they were doing so. In particular, I'd like to thank John Banes, Steve Bellovin, Burt Kaliski, Paul Kocher, Bodo Moeller, Dan Simon, and Robert Zuccherato for helping to fill in gaps in my knowledge. Special thanks goes to Terence Spies who provided valuable comments on the entire manuscript and answered numerous technical questions about Microsoft's SSL implementation.

This book makes extensive use of OpenSSL to generate demonstration SSL traffic. OpenSSL wouldn't exist without the hard work of Eric Young and Tim Hudson in creating SSLeay as well as the OpenSSL team in maintaining and improving OpenSSL after Eric Young's departure for greener pastures.

Brian Korver and Stacey O'Rourke from Network Alchemy/Nokia provided significant assistance in collecting some of the performance data in Chapter 6. Not only did they allow me to use their machines and network, they graciously reconfigured them in obscure ways so that I could produce particular cases.

Although I never met him, the late W. Richard Stevens nevertheless contributed greatly to this book. The idea of showing network traces to illustrate the protocol is taken from Stevens's fine *TCP/IP Illustrated* series. I've striven throughout this book (with only limited success) to emulate his clear and accessible style.

It wouldn't be possible to publish anything without the publisher and I've been quite pleased to be working with Addison-Wesley. I'd particularly like to thank Mary Hart for first suggesting this project to me and then putting up with the endless delays as the book exploded from a trim 200 pages to break through the 400 barrier. I'd also like to thank my production manager, Kathy Glidden, who patiently answered my endless typesetting questions.

Although they didn't contribute directly to the writing of this book, I'd like to thank Allan Schiffman, Marty Tenenbaum, and Jay Weber. Jay and Marty gave me a chance when I had talent and no experience. Over the eight years I've known him, Allan has

taught me an immeasurable amount about computer science. In a similar vein, had my parents not taught me how to think, I wouldn't be able to do anything useful at all.

Jennifer Gates played a major part in keeping me sane during the writing—an unenviable task. In the past few years, she and her husband Lee have many times provided hospitality and friendship above and beyond the call of duty.

The other major factor in my continued sanity was triathlon. Kevin Joyce and Kyle Welch provided valuable advice and motivation in the upkeep of my habit.

Finally, I'd like to thank Lisa Dusseault and Kevin Dick. Lisa and Kevin both read the entire manuscript and helped take my rough drafts and turn them into readable chapters. Without them, I would most likely never have finished at all and the text would certainly be far worse than it is now.

Camera-ready copy was produced by the author using James Clark's Groff package. I welcome electronic mail from readers with comments, suggestions, etc.

*Mountain View, CA
September 2000*

Eric Rescorla
ekr@rtfm.com

Security Concepts

1.1 Introduction

This chapter is intended to provide a basic introduction to communications security and cryptography. Communications security is a complicated topic and many fine books have been written about it. Our intent here is not to provide an exhaustive discussion of the topic but rather to teach you enough to understand the concepts and terminology that will be used throughout the rest of the book. Readers who are already familiar with cryptography and communications security should feel free to skip this chapter entirely.

We start by explaining the sorts of threats we're concerned about and the various sorts of security services we can provide. Next we provide a broad overview of cryptographic algorithms and how to put them together to provide these security services. Finally, we discuss some details of the various algorithms which will be relevant when we discuss their use in SSL/TLS.

1.2 The Internet Threat Model

The first thing that we need to do is define our *threat model*. A threat model describes what resources we expect the attacker to have available and what attacks the attacker can be expected to mount. Nearly every security system is vulnerable to some threat or another. To see this, imagine that you keep your papers in a completely unbreakable safe. That's all well and good, but if someone has planted a video camera in your office they can see your confidential information whenever you take it out to use it, so the safe hasn't bought you that much.

Therefore, when we define a threat model, we're concerned not only with defining what attacks we are going to worry about but also those we're not going to worry about. Failure to take this important step typically leads to complete deadlock as designers try to figure out how to counter every possible threat. What's important is to figure out which threats are realistic and which ones we can hope to counter with the tools available to us.

Designers of Internet security protocols typically share a more or less common threat model. First, it's assumed that the actual end systems that the protocol is being

executed on are secure. Protecting against attacks where one of the end systems is under the control of the attacker is extraordinarily difficult, if not impossible. This assumption comes with two caveats. First, compromise of any single end system shouldn't break security for everyone. There should be no *single point of failure*. For instance, if an attacker breaks system A, then all communications between B and A may be compromised, but communications between B and C should be safe. If we must have a single point of failure it must be possible to harden it against attack. Second, attackers may control systems that attempt to pose as legitimate end systems. All we're assuming is that users can expect that their own machines haven't been compromised.

Other than that, we assume that the attacker has more or less complete control of the communications channel between any two machines. He can certainly inject packets into the network with arbitrary address information, both for the sender and the receiver, and can read any packet that is on the network and remove any packet packet he chooses. Any packet you receive must be assumed to potentially come from the attacker and any packet you send might be modified in transit. An attack that depends on the attacker writing data to the network is known as an *active attack*. An attack that merely involves reading data off the network is known as a *passive attack*.

An obvious corollary of the assumption that the attacker can modify traffic is that the attacker can shut down all communication between any pair of machines simply by removing all relevant packets. This is one form of *denial-of-service* attack. Another form would be to force you to use up enormous CPU resources responding to connections. Conventionally, protocol designers don't worry about denial-of-service attacks, not because these attacks aren't important but because they're extraordinarily difficult to prevent.

One of the most important functions of a threat model is to arrange that security doesn't become more expensive than it is worth. Security measures should be employed only up to the point where the cost to implement them doesn't exceed the expected risk. Failure to make this judgment correctly can easily lead to a situation where no risk is judged acceptable and thus no acceptable system can be designed.

Part of the risk calculation is the effort required by the attacker to mount a given attack, and cost generally increases with each attack prevented. No security system is resistant to every attack. The function of a security model is to allow designers to determine which attacks are worthwhile to prevent.

However, accurately estimating how much security you need requires accurately estimating the attacker's capabilities. If an attack that was originally considered impractical is discovered to be simple, then there will be a window of vulnerability while people adjust their security models and implementations to compensate.

1.3 The Players

To make it easier to understand the various examples we'll be discussing in this chapter, we'll use the same names repeatedly for the various parties. By convention, the two communicating parties are referred to as Alice and Bob, after the names used in the original RSA paper [Rivest1979]. The attacker is known merely as "the attacker."

1.4 The Goals of Security

Most people speak of security as if it were a single monolithic property of a protocol, but as the previous discussion shows, that's clearly not true. Depending on precisely what capabilities the attacker has, he poses different risks to the security of our data. Communications security consists of a number of distinct but somewhat related properties. Exactly how these are partitioned depends on whom you're talking to. The partitioning we've found most useful is to divide them into three major categories: *confidentiality*, *message integrity*, and *endpoint authentication*.

Confidentiality

When most people think of security, they think of *confidentiality*. Confidentiality means that your data is kept secret from unintended listeners. Usually, these listeners are eavesdroppers. When the government taps your phone, that poses a risk to your confidentiality. (Incidentally, it's also a passive attack, unless the Fed on the line starts trying to imitate your voice.)

Obviously, if you have secrets, you're concerned that no one else knows them and so at minimum you want confidentiality. When you see spies in the movies go into the bathroom and turn on all the water to foil bugging, the property they're looking for is confidentiality.

Message Integrity

The second primary goal is *message integrity*. The basic idea here is that we want to be sure that the message we receive is the one that the sender sent. In paper-based systems, some message integrity comes automatically. When you receive a letter written in pen you can be fairly certain that no words have been removed by an attacker because pen marks are difficult to remove from paper. However, an attacker could have easily added some marks to the paper and completely changed the meaning of the message.

On the other hand, in the electronic world, since all bits look alike, it's trivial to tamper with messages in transit. You simply remove the message from the wire, copy out the parts you like, add whatever data you want, and generate a new message of your choosing, and the recipient is no wiser. This is the equivalent of the attacker taking a letter you wrote, buying some new paper, and recopying the message with changes. It's just a lot easier to do electronically because all bits look alike.

Endpoint Authentication

The third property we're concerned with is *endpoint authentication*. What we mean by this is that we know that one of the endpoints in the communication (typically the sender) is the one we intended. Without endpoint authentication, it's very difficult to provide either confidentiality or message integrity. For instance, if we receive a message from Alice, the property of message integrity doesn't do us much good unless we know that it was in fact sent by Alice and not the attacker. Similarly, if we want to send a

confidential message to Bob, it's not of much value to us if we're actually sending a confidential message to the attacker.

Note that endpoint authentication can be provided asymmetrically. When you call someone on the phone, you can be fairly certain that you have the right person—or at least that you got a person who's actually at the phone number you called. On the other hand, if they don't have caller ID, then the receivers of a phone call have no idea who's calling them. Calling someone on the phone is an example of recipient authentication, in which you know who the recipient of the call is (it's difficult but not impossible to compromise the phone network), but they don't know anything about the sender.

On the other hand, cash is an example of sender authentication. A dollar bill is like a message signed by the government. The government has no idea who's got any given dollar bill, but you can be fairly certain that any bill was actually printed by the U.S. Mint because currency is difficult to forge.

A Physical Example

To make this all a little clearer, let's examine a physical system that provides all three properties together and see how the various physical security features interact to provide each property. We'll start with a system with no security features and keep making changes until we have all the security features we want.

You receive a postcard in the mail, addressed to you, ostensibly from Alice. What can you know about it and its contents? Not much. Any random eavesdropper could have read it, and they could have changed it as well.

One potential improvement might be for Alice to write a letter instead of a postcard and put the letter in an envelope. Does this improve the situation? Not much, actually. By examining the envelope, you can be fairly confident that it wasn't tampered with (it could have been steamed open, but you can get envelopes that are resistant to this), so you might think that you've got confidentiality and message integrity for the contents. This isn't necessarily true. The attacker could have opened the original envelope, read the message or modified the contents, and then resealed it in a new envelope. So, in fact, you can't have any confidence in the message at all. Both message integrity and confidentiality have collapsed because you don't have sender authentication.

Note that from Alice's perspective, the situation is even worse. As far as she knows, the message was opened as soon as the mail carrier picked it up from her door. We'll show how to fix this problem a little later.

The traditional way to ensure that a letter is not opened in transit is to affix a wax seal across the envelope flap. It's possible to open and reattach such seals, but let's assume for the moment that it wasn't possible. Another more modern approach is for the sender to sign across the flap. Assuming that you recognize either the seal or the sender's signature, you can have some confidence that the envelope arrived unopened, because signatures are difficult to forge.

However, we can have this confidence only if we have some way of knowing who ought to have sent us the message. What if the attacker opened the envelope and then resealed it with his own envelope, and signed his own name across the flap? If we didn't

know that the signature should in fact be Alice's, we wouldn't be able to detect this attack.

To see why this would be useful, imagine that the envelope contains an order form for something, along with Alice's credit card number. The attacker changes the message to have his shipping address but Alice's credit card number and reseals the envelope. You incorrectly ship to the attacker and bill Alice. This attack is very difficult to stop with the sorts of technology we've discussed so far.

The mechanisms we've been talking about so far—envelopes, seals, etc., are *tamper-evident*. The receiver can tell that the contents were tampered with, but can't stop it from happening. The easiest way to stop this new attack is to use packaging that's *tamper-resistant*. (The phrase *tamper-proof* is often used, but that seems overly optimistic.) Imagine that instead of sending a letter, Alice sends us a safe containing the letter. The only two keys to the safe are in Alice's and our possession.

This pretty much fixes the problem entirely. Because you have the only other key and you know that you didn't send the letter yourself, you can be quite confident that Alice sent it, so you have sender authentication. Because no one besides you or Alice can open the safe, you know that it hasn't been opened and read or modified in transit, so you've got confidentiality and message integrity as well. Similarly, Alice knows that no one else but you can open the safe, so she can be assured that she has recipient authentication, confidentiality, and message integrity.

This approach has (at least) one serious problem. You have to exchange one of a pair of keys with everyone with whom you might ever want to communicate. If you think about how many messages the typical business sends in a day, you'll see that this method is inconvenient.

This example is useful for two purposes. First, it illustrates the various security services we might wish to provide. Just as important, it illustrates the concept of the threat model. Because it's fairly expensive to go around mailing safes, this isn't a very practical approach, and people typically settle for lower levels of security and just assume that the postal service is trustworthy. For particularly confidential information, they might use a courier, but that's about the typical limit of their concern. Attackers who are able to open the mail and arbitrarily change it just aren't part of the standard business threat model. The cost of providing high security for every message mailed is simply so high that no company can afford to do so.

1.5 Tools of the Trade

Note that we haven't even mentioned cryptography yet. We're going to get to it now. *Cryptology* is the theory of designing the various algorithms we use to provide security. *Cryptography* is the study of using these algorithms to secure systems and protocols. This section is going to provide a broad overview of the types of cryptographic algorithms available to us. The next section will discuss how to use them to provide security services.

Encryption

Conceptually, the easiest sort of algorithm to understand is encryption. The idea is simple: an encryption algorithm takes some data (called *plaintext*) and converts it to *ciphertext* under control of a *key*. The ciphertext looks like random data and no useful information about the plaintext (except possibly the length) can be gathered without knowing the key. The key is usually just a short random string, usually on the order of 8–24 bytes. The relationship between these elements is shown in Figure 1.1.

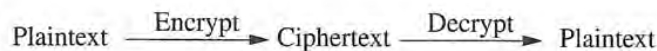


Figure 1.1 Encryption and decryption

An encryption algorithm should be viewed as providing pure confidentiality. The exact effects on the plaintext of tampering with the ciphertext vary depending on the exact encryption algorithm used, but such tampering is often quite difficult to detect. Thus, when we receive a message that has only been encrypted, we don't know that it hasn't been tampered with.

A good encryption algorithm should have its security totally determined by the number of possible keys. The fastest attack should be *exhaustive search*: given a ciphertext, the attacker tries each key one at a time until he finds a key that produces a plausible decryption. The security of the algorithm should depend solely on the secrecy of the key. The algorithm should not need to be secret to be secure.

An attack in which the attacker knows the plaintext corresponding to the ciphertext is called a *known plaintext* attack. An attack in which the attacker doesn't know the plaintext is called a *ciphertext only* attack. Even if the attacker doesn't know the details of the plaintext, he may know something about it, which would allow him to mount a ciphertext only attack. For instance, the attacker may know that the plaintext is ASCII, in which case any decryption that includes non-ASCII characters must be using the wrong key.

Because the sender and the recipient share the same key (which key must be kept secret), what we've just described is sometimes referred to as *secret key cryptography*, as opposed to *public key cryptography*, which we'll talk about in a little bit.

Encryption algorithm design has been a very active field, and there are tens if not hundreds of available algorithms. The most popular algorithms include the *Data Encryption Standard* (DES) [NIST1993a], *Triple-DES* (DES repeated three times) [ANSI1985], *RC2* [Rivest1998], and *RC4* (no official RC4 specification has ever been published but see [Schneier1996a] for a description).

Message Digest

A message digest is simply a function that takes as an input an arbitrary length message and outputs a fixed-length string that is characteristic of the message. The most important property of a message digest is *irreversibility*. It should be extremely difficult to

compute a message given its digest. It's easy to show via a counting argument that the digest value doesn't give enough data to generate the original message; the number of possible messages (of arbitrary length) is far longer than the number of fixed-length digests; many messages must map to a single digest, making it impossible to reverse the function. However, for a digest to be secure, it must be difficult to generate *any* of the messages that digests to this value. By difficult, we mean that you need to search a message space of proportional size to the size of the digest in order to find a matching message text.

The second important property of a message digest is that it should be difficult to produce two messages M and M' such that they have the same digest. This property is known as *collision-resistance*. It turns out that the strength of any message digest against finding collisions is only half the size of the digest, so a 128-bit digest is only 64 bits strong against collisions; it requires roughly 2^{64} operations to generate a collision. As a consequence, the limiting factor in choosing a digest length is usually strength against collisions rather than strength against reversibility.

The primary uses for message digests are for the computation of digital signatures and *message authentication codes* (MACs), both of which we'll talk about later in this chapter. However, there's at least one use of simple digests that's obvious now: they can be used to prove possession of a secret without revealing the secret. Imagine that you've invented something new and you want to show that you invented it first without actually telling anyone what it is. You write the secret down, compute the digest, and publish the digest in the newspaper classified ads. Then, if anyone challenges your primacy, you can show them the original text and they can independently verify the digest.

The most widely used message digest algorithms are Message Digest 5 (MD5) [Rivest1992] and Secure Hash Algorithm 1 (SHA-1) [NIST1994a]. The term *hash algorithm* is often used synonymously with the term *message digest*, because digest algorithms are superficially similar to normal hash algorithms. I'll be using the terms *hash* and *digest* interchangeably throughout the text.

Message Authentication Codes

Imagine that Alice and Bob share a key and Alice wants to send Bob a message that he will know is from her. If she wanted to encrypt it, that would be straightforward—she'd just use the key they share as an encryption key. But as we said, this doesn't provide any real assurance that the message hasn't been tampered with, and only some assurance that it's from Alice. What we need is a new tool, the *message authentication code* (MAC). A MAC is sort of like a digest algorithm, but it also incorporates a key into the computation, so the MAC is dependent on both the key being used and the message being MACed. Actually, MACs are usually constructed from digest algorithms.

Although there have been a number of attempts to construct MACs based on various digest algorithms, the Internet security community seems to be settling on a construct called HMAC [Krawczyk1997], which describes how to create a MAC with provable security properties based on any digest for which a certain somewhat reasonable set of assumptions holds. A variant of HMAC is used in SSLv3, and HMAC itself is used in TLS.

The Key Management Problem

With the addition of a MAC to our toolbox, we now have an electronic system equivalent to mailing safes around, as we discussed at the beginning of the chapter. Naively, Alice can take our message, encrypt it with our shared key, add a MAC also based on the shared key, and send it to Bob. She knows that only Bob can read it because only Bob has the shared key needed to decrypt it. Similarly, Bob knows that only Alice could have sent the message because only Alice has the shared key needed to create the MAC on the message. Thus, he knows that Alice sent the message and that it hasn't been tampered with.

So, we have everything we need, right? Wrong. Although these messages are lighter than safes and thus cheap to mail around, we still have the problem of sharing a key with everyone we communicate with. This is inconvenient because it means having all those keys floating around. But more important, it means that you must actually meet everyone with whom you want to communicate in order to exchange keys. This puts a damper on your being able to buy anything over the Internet unless you've personally met the vendor. This inconvenience is referred to as the key management problem.

Key Distribution Centers

The most popular solution to the key management problem is *public key cryptography* (PKC), which we'll talk about in the next section, but there is also a fix for the key management problem using only the tools we've talked about so far. The basic idea is that we use a *trusted third party*, which we trust to authenticate other parties to us. The third party is usually implemented as a secure machine somewhere on the network. This machine is called a *key distribution center* (KDC). Each person who wants to communicate securely shares a key with the KDC. When Alice wants to communicate with Bob, she sends a message to the KDC, protected with the key she shares with the KDC, requesting to communicate with Bob. The KDC generates a new encryption key for the Alice-Bob communication and returns it in a message called a *ticket*.

A ticket consists of two messages in one. The first is a message to Alice with the new key. The second is a message to Bob encrypted under Bob's key containing the new key. Alice forwards Bob's half of the ticket to Bob and now Alice and Bob share a key. The basic version of this protocol was invented by Needham and Schroeder [Needham1978], but the most widely deployed variant is Kerberos, which is used extensively for authentication and encryption at MIT, among other places (see [Miller1987]).

This scheme has two primary drawbacks. First, the KDC must be online all the time because no communication can be initiated when it's offline. Second, the KDC can read all the traffic sent by any two parties. It can also forge traffic between two parties. Worse yet, if the KDC is ever compromised, all traffic sent between any two KDC users is compromised. This is not good. Nevertheless, there has been a fair amount of interest in this type of protocol for closed systems.

Cer

Public Key Cryptography

In 1976, some very smart people at Stanford figured out a better way to solve the key management problem. In the article “New Directions in Cryptography” [Diffie1976], Whitfield Diffie and Martin Hellman suggested what we now know as public key cryptography. The basic idea is that you have a function in which encryption and decryption use different keys. You publish your encryption key (the *public key*) but keep your decryption key (the *private key*) secret. (Because the public and private keys are different, public key cryptography is sometimes called asymmetric cryptography and secret key cryptography is sometimes called symmetric cryptography). This means that anyone can send you a secret message without ever meeting you. This solves the confidentiality part of the equation while eliminating the inconvenience of preshared keys.

As it turns out, public key cryptography was also invented by one of the British intelligence agencies, the *Communications-Electronics Security Group* (CESG) during 1970 to 1974. They referred to the practice as *non-secret encryption* (NSE). Interestingly, the techniques invented include both Diffie-Hellman and a variant of RSA, which implies that these methods are somehow fundamental. However, the results were classified and the academic community reinvented both techniques independently. The prior existence of these techniques in the intelligence community was only revealed in 1998. [Ellis1987] details the history of NSE. [Ellis1970] describes a demonstration that NSE was possible. [Cocks1973, Williamson1974, Williamson1976] describe the specific techniques.

It turns out that PKC has a solution to the authentication part of the problem, too. Your private key can be used to create something called a *digital signature*, which bears the same relationship to a MAC that public key encryption bears to secret-key encryption; you use your private key to *sign* a message and the receiver uses your public key to *verify* your signature. Note that a digital signature has one important property that a MAC does not: *nonrepudiation*. Either sender or receiver can generate a MAC, but only the signer can generate a signature. Thus, the recipient can prove that the sender signed the message and the sender cannot deny it.

Certification

Unfortunately, what we have so far doesn't completely solve the key management problem, though it gives us the tools we need to solve it. The easiest way to see what is still wrong is to ask how the parties get each other's public keys. If the keys are published electronically or if they're exchanged by the communicating parties, then an attacker can tamper with them while they are in transit to the receiver. When two parties want to communicate, the attacker intercepts their keys and instead sends his own key to each party. Thus, each party encrypts to him and then he reencrypts to the real recipient, as shown in Figure 1.2. This is called a *man-in-the-middle* attack.

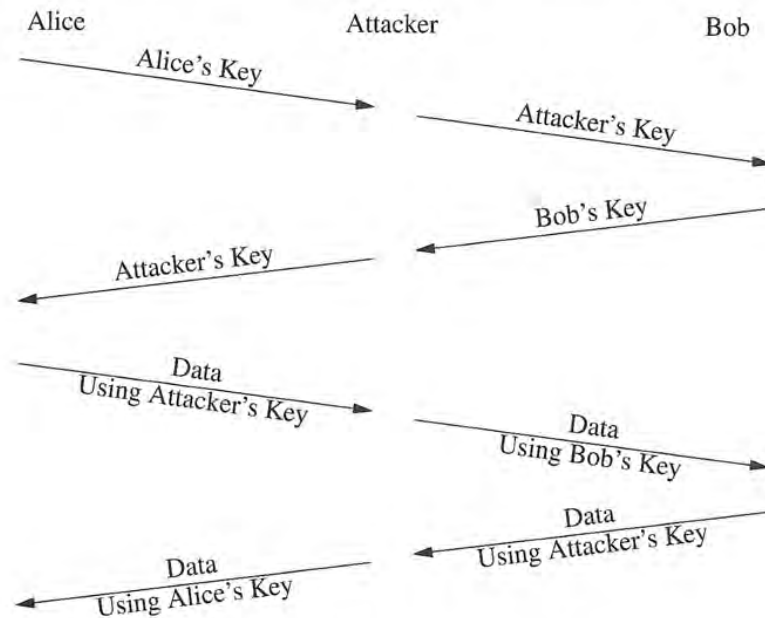


Figure 1.2 The man-in-the-middle attack

However, if keys are published physically, it's a major inconvenience. The solution is (again) to have a trusted third party, called a *Certificate Authority* (CA). The CA publishes a directory signed with the CA's private key. In actual practice, rather than signing a directory, what the CA does is sign individual messages which contain both the name of the key owner and his public key. These messages are generally known as *certificates*, hence the name certificate authority. The primary standard for certificates is X.509 [ITU1988a] profiled for the Internet in RFC 2459 [Housley1999a].

The CA's public key is published in some physical form, but there aren't many CAs and they don't change their keys often, so it's not such a big problem. In actual practice, CA keys are usually compiled into the software that needs them so that they are distributed with that software. This makes a lot of sense when the software is shipped on CD-ROM or floppy, but sometimes that software is just downloaded, in which case we're right back where we started—which just goes to show that people often don't act rationally.

Figure 1.3 shows an expanded view of a public key certificate. The details aren't that important, but notice the basic structure: The certificate contains an *issuer name* (the name of the signer of the certificate, in this case "Secure Server..."), *subject name* (the name of the holder of the key that the certificate vouched for, in this case "www.amazon.com..."), the *subject public key* (the key itself), a bunch of control information such as expiry date and serial number, and a *signature* over the whole data object.

```

version:                v1
serial number:          2A 17 EF 73 97 07 74 7B E2 4B FB
                        61 95 DB 4D 77
signature
  algorithm:            md5WithRSAEncryption
issuer:
  C=US
  O=RSA Data Security, Inc.
  OU=Secure Server Certification Authority
validity
  not before:           Sat Jan 28 02:21:56 1995
  not after:            Thu Feb 15 02:21:55 1996
subject:
  C=US
  ST=Washington
  L=Seattle
  O=Amazon.com, Inc.
  OU=Software
  CN=www.amazon.com
subject public key
  algorithm:            rsaEncryption
  modulus
    bit length:         1024
    value:
      00 C8 1B 8B FA 40 C3 5B E3 46 3F 17 10 56 19 64 C4 F4 F9
      CC AE CA F7 0B 02 1C C3 2D 27 60 91 16 C0 A1 23 8B CA 90
      77 31 25 CA D9 DE B0 87 F5 25 C9 12 7A 95 DF DC 6C E4 1C
      C3 31 9F 77 BE 69 3E 9F BB 35 BF F3 3D BA 7A 72 DA 5D 0C
      60 91 29 F8 89 67 50 5C 32 46 63 F2 FF 42 9D 24 F2 DC 6F
      E5 CA D3 CD 3A AB 9D 5F A9 4D B0 82 91 E3 D3 EA AA EF 78
      8A C1 06 B6 6D EA 56 B8 7E 68 5D AF 4D 85 AF
  public exponent:
    bit length:         2
    value:              03
signature
  value:
    03 43 60 4B 5B 4B F1 78 56 BF B4 9B 81 E6 EE 0D 19 1B 4E 43 BD
    D9 C7 62 62 55 32 C7 15 A4 33 3A CA 0E 60 E5 FE D7 53 94 C6 AC
    17 D0 CE 7B 11 27 0C 3B 26 19 6D 35 55 4C D8 26 F4 5F F0 90 0D
    90 7F FC 39 47 FE EE B4 72 92 93 BF 93 7F 5C 56 38 10 F5 E5 58
    B5 6C 3E E0 B4 55 8D 74 BE 84 F1 53 67 49 5B 14 12 E6 A7 59 A9
    97 9E 6C E4 59 A6 8F 4E 7E B5 D9 2D 80 3F 38 3C 4C 11 A7 37

```

Figure 1.3 A public key certificate

Public key solutions involving certification still involve a trusted third party (the CA), but they do fix the primary problems we described with KDC-based systems. Because the same certificate can be used to prove one's public key to anyone, the CA doesn't have to be online in order for Alice and Bob to communicate. And because the CA doesn't have access to anyone's private keys, it can't read any of the messages.

Distinguished Names

The subject name and the issuer name in the certificate are X.500 *distinguished names* (DNs) [ITU1988b]. The idea behind a distinguished name is to provide a unique name for every network entity. In order to do this, a DN has a hierarchical structure. A DN consists of a series of *relative distinguished names* (RDNs). RDNs are like components of DNS names. Each RDN specifies the name of the entity in the namespace of some other entity. Thus, the highest-level RDN must be globally unique, but each RDN under that must be unique only within the scope of the previous RDNs. RDNs have structure as well. Each RDN consists of a series of *attribute-value assertions*. An AVA is basically a key/value pair. Figure 1.4 shows a possible sample DN.

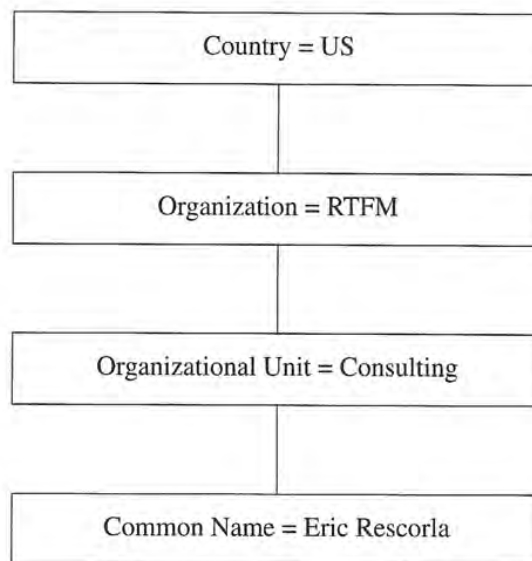


Figure 1.4 A sample DN

Each box in Figure 1.4 represents an RDN. Thus, the top-level RDN contains one AVA, with the *attribute* being *Country* and the *value* being *US*, and so on. Note that although

in theory each RDN may have multiple AVAs, in practice, they almost always have just one. Similarly, although in theory any AVA may appear at any level, in practice they go from least specific (country) to most specific (common name—roughly corresponding to your name—or e-mail address). It's quite common to render distinguished names in text form, in which case the attribute names are abbreviated. The name in Figure 1.4 would be rendered

```
C=US, O=RTFM, OU=Consulting, CN=Eric Rescorla
```

Extensions

Certificates contain a standard set of values, but it's quite common to want to put other information in them. X.509 version 3 provides for an arbitrary series of *extensions*. Extensions are simply type-value pairs. Although X.509 allows for private extensions, a number of standard extensions have been provided. The three important ones for our purposes are

subjectAltName contains other names for this user. These may be other DNs, but may also be other name forms, including **dnsName** (a DNS hostname) and **emailAddress** (an e-mail address).

keyUsage contains a bitmask of acceptable uses for this key, including signing, encryption, etc.

extendedKeyUsage allows a list of arbitrary *object identifiers* (OIDs), (see the *ASN.1*, *BER*, and *DER* section) listing detailed possible uses for this key.

Certificate Revocation

Consider the case where user's key is compromised by being lost or exposed. We'd like to have some way to communicate to other users that a public key and the certificate verifying it were no longer to be trusted. The most common way to approach this problem is to use something called a *Certificate Revocation List* (CRL, pronounced "krill").

A CRL is a signed, dated list of all the certificates that have been *revoked*, that is, they are no longer considered to be valid. Assuming that the CA in question issues CRLs, verifying a certificate requires obtaining the appropriate (latest) CRL and checking that the certificate doesn't appear on that list.

Unfortunately, CRLs don't work very well, for two reasons. First, they need to be issued periodically, which means that there is a window of vulnerability between the time period when a CRL issues and the next CRL is due. The key might have been compromised but users can't tell because the CRL on which it would have appeared has not been issued. Second, CRL distribution is problematic. CRLs can get rather large and either protocols need to carry them around or users need to get them from the CAs, which requires the CA to be online.

If the CA is going to be online, it can also provide *online certificate status*. This can provide completely up-to-date revocation status. The user can make a request of the CA for the status of a given certificate and the CA provides a signed response. The *Online*

Certificate Status Protocol (OCSP) [Myers1999] can be used to do this job. However, the need to sign these responses creates a high load on the server CPU. Obviously, this is undesirable. Some optimizations that reduce this load [Kocher1996a] have been described but are not in wide deployment.

ASN.1, BER, and DER

ASN.1 falls into the category of “highly unpleasant things that it is sometimes necessary to know.” A lot of our basic security tools, in particular X.509 certificates, end up being defined using ASN.1. ASN.1 was designed as part of the International Telecommunications Union’s *Open Standards Interconnect (OSI)* effort as a description language for the OSI protocols. The basic idea was to create a system for describing data structures that would allow machine generation of data encoders and decoders (generically, *codecs*).

This seemed like a good idea at the time: you have a language which can be used to describe data formats as structured types, much like that for C structs and Java classes. This language is called *Abstract Syntax Notation 1 (ASN.1)*. You also describe how to map structures in this language into data encodings on the wire. This allows you to write a compiler that does two things:

1. Generate mappings from ASN.1 structures to whatever language you’re using
2. Automatically generate codecs for the structures

Unfortunately, ASN.1 is complicated and counterintuitive, making it hard to write the ASN.1 specifications themselves as well as making compilers fairly complicated. Worse yet, apparently on the theory that more is better, the people who designed ASN.1 defined not one but upwards of four different sets of encoding rules for laying out the data on the wire. Each encoding serves a slightly different set of goals, but the result is a confusing mess.

The two ASN.1 encoding rules we’ll be concerned with are *Basic Encoding Rules (BER)* and *Distinguished Encoding Rules (DER)*. Whereas BER allows several ways to encode any given piece of data, DER is a subset of BER that picks one and sticks with it so that you can be sure that any given structure will be encoded the same way by any codec. The advantage of BER is that it is often more efficient to BER encode data than to DER encode it. However, if you are digitally signing data, you want semantically equivalent messages to always have the same encoding, and so you need DER.

We’re not even going to try to describe how to read ASN.1 in any detail. In order to get the general idea, all you really need to know is that it’s like C structures except that the type definitions are backwards: the name comes first and the data type comes second. Figure 1.5 shows a sample ASN.1 structure. What it says is that *Foo* is a sequence of two elements, *bar* and *mumble*. *bar* is of type *INTEGER* and *mumble* is of type *BIT STRING*.

1.6

F

V
a
p
fi
T
bc
T
k
e
e
ti
tisi
le
a
ta

```
Foo ::= SEQUENCE {  
    bar INTEGER,  
    mumble BIT STRING  
}
```

Figure 1.5 A sample ASN.1 structure

This example should give you enough information to get the general idea of what ASN.1 structures mean. If you're interested in the details, your best bet is to refer to [RSA1993a]. The actual standard is in [ITU1988c] and [ITU1988d].

For the purposes of this book, you only really need to know about one other thing: *object identifiers* (OIDs). OIDs are globally unique byte strings assigned to any kind of object, such as algorithms, keyusages, etc. The OID space is *federated*—ISO has assigned sections of the OID space to various entities who can further divide the space among other entities. Thus, a large number of organizations can assign OIDs and it's possible to get your own OID section (called an *arc*) if you want to assign private but unique OIDs.

1.6 Putting It All Together

We now have enough pieces to build some simple security systems. The system we're about to describe sounds complicated, but it's important to realize that nearly every piece of communications security technology we'll be discussing is built from one of four simple pieces: *encryption*, *digest*, *public key encryption*, and *digital signature*. These pieces are often called *security primitives*, and primitives can be combined to build more complicated structures.

For instance, public key methods are much slower than secret key methods, so it's convenient to combine the two techniques, using public keys to exchange secret keys. To encrypt a message, Alice generates a random secret key (variously called a *session key*, *message encryption key*, *content encryption key*, or *data encryption key*) and encrypts it under Bob's public key, which she gets from Bob's certificate. She then encrypts the message with a symmetric algorithm using the session key. This combination of public key encryption and symmetric encryption provides fast message encryption with the benefits of certificate-based key management.

Similarly, digital signature algorithms are very slow and can only be used with small messages. But combined with message digests, they can be used to efficiently sign large messages. To sign a message, Alice computes the message digest of the message and signs that digest with her private key. The combination of message digests and digital signatures provides message integrity and sender authentication without shared keys.

The receiving process, shown in Figure 1.7, reverses these steps.

1. Bob uses his private key to decrypt the session key.
2. He uses the session key to decrypt the message, certificate, and digital signature.
3. He computes the message digest of the message himself.
4. He verifies Alice's certificate and extracts Alice's public key.
5. He uses Alice's public key to verify Alice's digital signature.

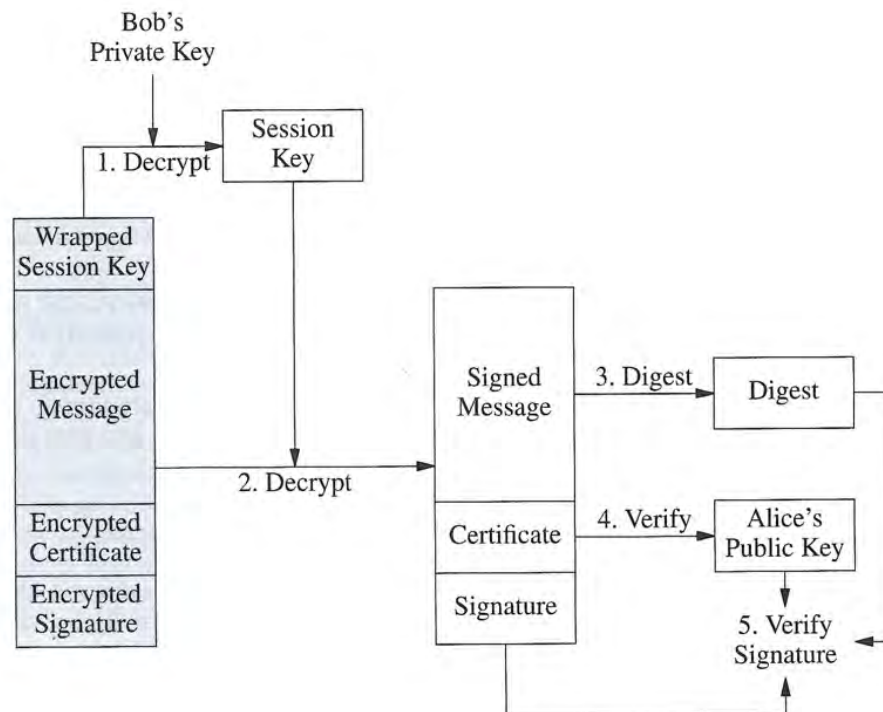


Figure 1.7 The message receiving process

1.8 A Simple Secure Channel

The system we've just described works pretty well if all we're doing is sending single messages around, as with e-mail, but it rapidly becomes inadequate if what we want is to have a communications channel over which we can pass arbitrary messages. We'd like to be able to establish a set of secret keys that we could use for the entire connection. This would let us avoid doing expensive public key operations for every packet,

which is especially important for interactive applications where each keystroke might generate a packet.

A second improvement we'd like to add in an interactive protocol is the ability to do *certificate discovery*. In the messaging protocol we described in the previous section, Alice needs to be able to get Bob's certificate before she can send him a message. This is done either by consulting a directory or by having Bob send a nonencrypted message containing his certificate. In an interactive protocol, though, Bob can send Alice his certificate as the first thing he does.

A typical design for such an interactive system is to have a handshake phase where Alice and Bob authenticate each other and establish a set of keys. They then move on to a data transfer phase where they use those keys to actually transmit the data they're interested in. The rest of this section builds a simple protocol that does this job. We'll call it *Toy Security Protocol (TSP)*.

The basic stages in our protocol will be

Handshake. Alice and Bob use their certificates and private keys to authenticate each other and exchange a shared secret.

Key derivation. Alice and Bob use the agreed upon shared secret to derive a set of cryptographic keys which can be used to protect the traffic.

Data transfer. The data to be transmitted is broken up into a series of *records*, each of which is individually protected. This allows data to be transmitted as soon as it is ready and processed as soon as it is received.

Connection closure. Special protected closure messages are used to securely close the connection. This prevents an attacker from forging closes and truncating the data being transferred.

A Simple Handshake

The first thing that Alice does is to send Bob a message saying that she's ready to communicate. There's no cryptographic content here. It's just a "Hello." Bob responds with his certificate. We've now accomplished certificate discovery, so we could go back to sending around individual messages like we did before, but there's a better way; what we actually want to do is arrange that Alice and Bob share a single secret, which we'll call the *master secret (MS)*. Afterwards, Alice should know that only Bob knows the MS and vice versa. Then we'll use the MS to create a set of keys that we can use to encrypt the data.

The first step is for Alice to generate a random number to be the MS. Actually doing this requires having a *cryptographically secure* random number generator, which is trickier than it sounds. There are a number of techniques, but they border on magic and we won't discuss them here. Once Alice has the master secret, she simply encrypts it under Bob's public key to produce the *encrypted master secret (EMS)*. We've now done the first half of the job. Alice knows that only she and Bob have the MS. If Bob doesn't care about her identity, she could just send the EMS to Bob and they could start

communicating. This is called *one-way authentication* (see Figure 1.8.) This sort of authentication would be useful (for example) if Bob were running an Internet store and only needs Alice's credit card number.

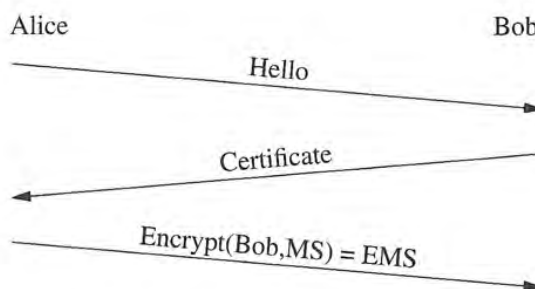


Figure 1.8 A handshake with one-way authentication

On the other hand, if Bob does care about Alice's identity, we need to add one more feature to the protocol to let him have that assurance. If Alice signs the EMS with her private key, then Bob can verify the signature and be sure that Alice vouches for the EMS she just sent.

This new protocol also contains one new subtlety: Bob sends Alice a random number called a *nonce* along with his certificate (see Figure 1.9.) The idea here is to stop an attacker from retransmitting all of Alice's messages to Bob and convincing him that he's engaging in a new conversation. This attack is known as a *replay attack*. We'll use the nonce when we make keys to ensure that the keys produced by this handshake are different from the keys produced by any other handshake.

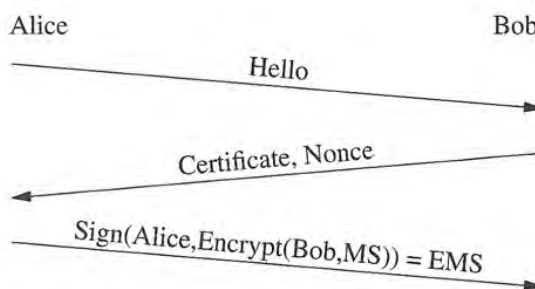


Figure 1.9 A handshake with mutual authentication

A Simple Data Transfer Protocol

At the end of the handshake, Alice and Bob share a master secret, but what good is that? The purpose of the exercise is for Alice and Bob to transfer data, not random numbers. In order to achieve that goal, we need an entirely different protocol whose job it is to arrange that. The rest of this section describes how to build a simplified one.

Making Some Keys

The first thing we need to do is to make some keys. In general, it's considered a bad idea to use the same key for more than one type of cryptographic operation. We'll see one reason for this in the discussion of stream ciphers later on, but you can consider it a form of insurance. Imagine that the attacker figured out how to break one algorithm (say the encryption algorithm). If you use the same key for MAC and encryption and the encryption is broken, then the communication is totally insecure. If you use different keys, then the MAC will still be secure even if the encryption is broken.

In this case we need four keys, one for encryption in each direction and one for MACs in each direction. In order to do this we need a tool we haven't seen yet, which is typically called a *key derivation function* (KDF). A KDF takes the master secret and (usually) some other random data and creates the appropriate keys out of it. In this case, the KDF takes the master secret and the nonce that is transmitted by Bob. KDFs are usually constructed using message digests and we'll skip the details here.

For convenience, we'll label these keys as follows:

E_{cs} —encryption key for data sent from client to server

M_{cs} —MAC key for data sent from client to server

E_{sc} —encryption key for data sent from server to client

M_{sc} —MAC key for data sent from server to client

Data Records

The next task is to describe how we're going to package the data. If we're working over a reliable protocol like TCP, we might imagine that we could just encrypt the data in a constant stream as we write it to the network. But then where would we put the MAC? If we simply transmit it at the end, then we don't have any message integrity until we've processed all the data. This isn't really any good because we can't safely act on received data if we don't know it hasn't been tampered with.

The solution to this problem is to break up the data into a series of records, each of which carries its own MAC. When we've read a record and checked its MAC, then we know that it's ok and we can act on it. What we're about to describe is the minimal record format that does this job.

When we read data off the wire, we need to be able to tell which data bytes are the encrypted data and which are the MAC. If we were willing to commit to a constant amount of data in each record, we could simply know which was which, but this doesn't work very well: in some circumstances, we want to be able to transmit one byte at a time, and having one-byte-long records would mean that we expanded the data

something like twenty times when we transmitted it. This really isn't acceptable if we want to transmit a lot of data in bulk.

Since fixed-length records are so inefficient, we need some way to have variable length records. This means that we have to have a length field to tell us where the record data ends. This field has to come before the record so we know how much data to read. The MAC can come before or after, but it typically comes after to allow it to be appended to the message after it's computed. (This is a matter of programming convenience). This leaves us with the record format shown in Figure 1.10.

Length	Data	MAC
--------	------	-----

Figure 1.10 A simple record format

So, when the client wants to encrypt a data block D of length L , it goes through the following process:

1. Compute a MAC over the data using M_{cs} . We'll call this MAC M .
2. Encrypt D using E_{cs} . We'll call this C . We're assuming here that encrypting the data doesn't expand it. As we'll see in Section 1.13, this sometimes isn't true.
3. Transmit $L||C||M$ (the $||$ stands for concatenation).

To read the data, the server goes through the reverse process.

1. Read L from the wire. We now know that D is L bytes long.
2. Now read C and M from the wire.
3. Decrypt C using E_{cs} to recover D .
4. Compute M' over D using M_{cs} .
5. If $M' = M$ then everything is ok. Process the record. Otherwise report an error.

In other words, the MAC computation is as shown in Figure 1.11 where x is either cs or sc depending on which direction we're sending the data.

Sending:

$$M = \text{MAC}(M_x, D)$$

Receiving:

$$M' = \text{MAC}(M_x, D)$$

compare M to M'

Figure 1.11 MAC computation

Sequence Numbers

Unfortunately, this simple protocol has a security hole. Because the records aren't labelled by the order they're transmitted in, an attacker can take a record off the wire and send it again to the receiver. (Remember, this is called a *replay attack*.) To see why a replay attack might be used, consider the case in which the messages are financial transactions and the attacker might want to arrange to be paid twice. Without protection against replay attack, the attacker could simply replay the message requesting payment.

The current protocol also allows the attacker to remove records or reorder them. It's easy to think up situations in which reordering or replay attacks might be useful and we'll leave it as an exercise for the reader. In order to fix this problem, we need to use a *sequence number*. The first record that each side transmits is numbered 1, the second 2, etc. When you receive a record, you check that its sequence number is the one you expect. If it's not, you report an error.

Of course, the sequence number has to be part of the input to the MAC in order to prevent the attacker from changing the sequence number. The easiest way to do this is simply to prepend the sequence number to D before it's encrypted and MACed. However, if we're communicating over a reliable protocol (like TCP), then the sequence number is implicit and we don't even have to transmit it. Messages will always be delivered in order, so we can just maintain a counter on either side and use it as part of the MAC computation:

$$M = \text{MAC}(M_s, \text{Sequence} \| D)$$

Note that the sequence number doesn't prevent the attacker replaying *all* of Alice's messages. However, because Bob generates a new nonce for each handshake with Alice, if the attacker tries this attack, Bob will generate a different set of keys (because his nonce will be different) and so when he goes to decrypt the data records, he'll get garbage rather than the plaintext, and the MAC check will fail.

Control Information

We have one more security problem to deal with. Remember that it's easy for the attacker to forge packets. Well, TCP connection closes are just another packet, so it's easy for the attacker to forge connection closes. That means that the attacker can mount a *truncation attack*, in which the attacker convinces one side (or both) that there is less data than there actually was. In order to fix this problem, we need to have some way for Alice to tell Bob (or vice versa) that she's done sending data. Then, if the attacker forges a TCP close, the victim can tell that something is wrong because the end of data message never arrives.

There are a number of tricks we could use to prevent this attack. For instance, we could say that a zero-length record means that you're done. However, in general it would be nice if we had some place to transmit *control messages* that aren't part of the data stream. For instance, we might want to have some way to report errors. An easy way to accomplish this is to have each record be *typed* by what kind of data it carried, regular or control. We can do this by adding a *type field* to each record, and the type field has to be MACed in order to protect it, as shown in Figure 1.12.

Length	Sequence Number	Type	Data	MAC
--------	-----------------	------	------	-----

Figure 1.12 Improved record format

The MAC would then be computed using:

$$M = \text{MAC}(M_x, \text{Sequence} \parallel \text{Type} \parallel D)$$

Of course, the type field isn't much good if we don't define some types. We'll go with something simple. If the type field is 0, that means just treat the record as if it were ordinary data. If it's 1, that means that this record is control information. The data portion of the record needs to be scanned by the protocol to figure out what to do next. We'll adopt a simple convention that all control data consists of a simple number. We'll reserve 0 to indicate that the connection is closing, and use nonzero numbers to indicate errors, as shown in Figure 1.13.

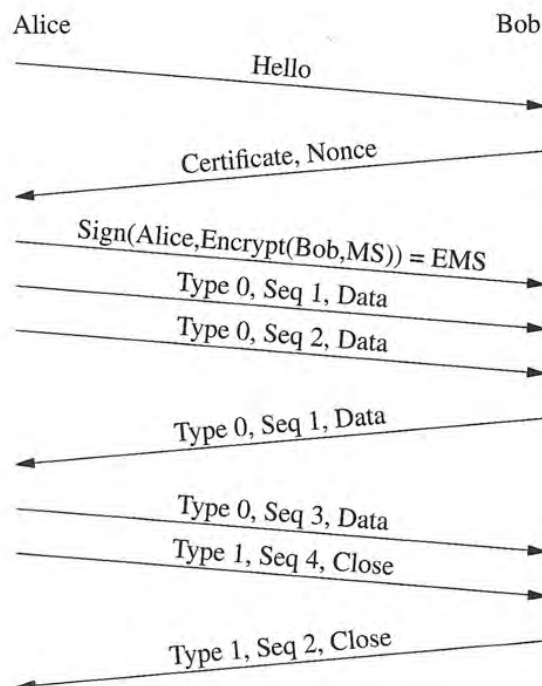


Figure 1.13 Full Toy Security Protocol (TSP) connection

Summary

As the astute reader will probably have guessed, we've now reproduced most of the essential features of SSL. We've got handshaking, key exchange, mutual authentication, and secure data transfer. What's left?

First, TSP isn't complete. We haven't specified enough detail for someone to actually implement it. Some of these details aren't that important. For instance, how long is the length field? Two bytes? Three? Variable? This governs the maximum record length, so it's not unimportant, but mainly it has to be big enough—and we all have to agree. Some of the other details are a little more tricky. Exactly what algorithms are we going to use for key exchange, authentication, etc.? In any case, we'd have to fill in a lot of detail before this protocol would even work.

TSP is skimpy on details, but more important, it is missing some desirable features. The most important thing it's missing is *negotiation*. We'd like to be able to support a number of different algorithm choices for each essential task. That way, if one algorithm is broken, we can switch over easily. We'd also like to be able to choose whether clients are authenticated or anonymous. Nevertheless, it's an interesting and instructive toy and captures most of the idea.

1.9 The Export Situation

Because so much computer software is written in the United States, the U.S. export situation has had a dramatic impact on the design of communications security systems. Export policy in the U.S. has been historically driven by the National Security Agency (NSA). NSA has responsibility for securing the government's communications and snooping on the communications of others. The actual enforcement of export policy is always left to someone else, originally the State Department and now the Bureau of Export Administration (BXA) of the Commerce Department, but all the important decisions are made by NSA.

The Stone Age

Up until September of 1998, the general idea behind export policy was simple. It was legal to export encryption technology for use in authentication, but confidentiality was severely restricted. If your product included any confidentiality at all, it had to be reviewed before export. The rules for what would pass review were never published but it was generally understood that the limits were 40-bit-strong encryption and 512-bit-strong key exchange.

Provided you met these criteria, you could get what was called *commodities jurisdiction*, which basically meant that you could apply to the Department of Commerce for a general license to export your software without having to seek approval. If you wanted to sell your software widely as a mass market product or have it downloaded over the net, this was the approval you needed. Even with this classification, you still couldn't export your software to a few embargoed countries: Cuba, Iran, Iraq, Libya, North

1.1

At the April 1995 IETF meeting in Danvers, Massachusetts, the IESG adopted what came to be known as the *Danvers Doctrine*, which stated that the IETF would design protocols that embodied good engineering principles, regardless of exportability. At the time, this doctrine implied support for DES at a minimum and over time it came to mean 3DES. Moreover, the IETF had a longstanding preference for unencumbered (i.e., free) algorithms when possible. When the Merkle-Hellman patent (covering all of public key cryptography) expired in 1998, but RSA was still patented, the IESG began pressuring working groups to adopt free public key algorithms. Finally, many IETF members felt it was good practice for protocols to have a mandatory set of options to ensure that any two implementations could talk to each other.

When the TLS working group had finished its work, in late 1997, it sent the document off to the IESG, and the IESG sent it back with instructions to add a mandatory cipher—DSS for authentication, DH for key agreement, and 3DES for encryption, thus solving the above three issues. Much discussion on the mailing list ensued, with Netscape in particular resisting mandatory algorithms in general and 3DES in particular. After a prolonged standoff between the IESG and the working group, grudging consensus was reached and the document was sent back with the appropriate changes.

Unfortunately, in the meantime, another obstacle appeared. The IETF *Public Key Infrastructure* (PKIX) working group, which was standardizing a profile for X.509 certificates, was winding up its work. TLS depended on certificates and hence on PKIX, and IETF rules forbid protocols advancing ahead of other protocols on which they depend. PKIX finalization took rather longer than expected and added an additional delay. Finally, in January of 1999, over two years late, TLS was published as RFC 2246 [Dierks1999]. As of this writing, deployment has been less than spectacular. Internet Explorer supports TLS but Netscape Navigator does not. Most toolkits do support TLS.

2.7 SSL for the Web

The primary use of SSL is to protect Web traffic using HTTP. We discuss this topic extensively in Chapter 9. Here we do so to provide a brief overview of a typical application in order to explain some of the choices that were made in SSL design.

The process is simple. In HTTP, a TCP connection is created and the client sends a request. The server responds with a document. When SSL is used, the client creates a TCP connection, establishes an SSL channel on top, and then sends the same HTTP request over the SSL channel. The server responds in a similar fashion over the SSL connection.

Because the SSL handshake looks like garbage to an ordinary HTTP server, in order for the process to work correctly the client needs some way to know that the server is prepared to accept SSL connections, as not all servers do so. A Web address (technically known as a *Uniform Resource Locator* (URL)) beginning with `https` rather than `http` is used to indicate that SSL should be used, for example

```
https://secure.example.com
```

As a consequence, the combination of HTTP running over SSL is often referred to as *HTTPS*.

If all goes well and the server and client can communicate, the user is typically rewarded with some sort of user interface indication that security is in use. In older versions of Netscape, a blue bar was displayed below the toolbar and a key appeared in the lower left-hand corner of the browser. In newer versions of Netscape, the open lock in the lower left-hand corner closes. In Internet Explorer, a lock appears in the lower right-hand corner of the screen. These are all indications that the current page was fetched using SSL.

Although nearly all modern commercial Web servers implement SSL, initially Netscape charged a significantly different price for their SSL-enabled server (*Netscape Commerce Server*) than for their ordinary server. Moreover, due to patent reasons, none of the free servers in the United States support SSL with RSA. (And since RSA is the de facto standard, this might as well mean not at all.) Even if you have a server that implements SSL, you might choose not to enable security. HTTPS puts a much higher load on the server machine than does pure HTTP, and obtaining the required certificate is fairly inconvenient. Moreover, certificates aren't cheap. Verisign, the biggest CA as of this writing, charges over \$300 U.S. for a single certificate.

In January 2000, Netcraft's survey [Netcraft2000] found about 1.5 million servers running SSL, but only about 60,000 of them were certified by a well-known third party. The rest were self-signed or issued by a private CA. A client would require special configuration to connect to these servers.

2.8 Everything over SSL

Because so many protocols run over TCP, and SSL connections act so much like TCP connections, securing a preexisting protocol by layering it over SSL is a very attractive design decision. In addition to HTTP over SSL and NNTP over SSL (often referred to as *SNEWS*), proposals soon appeared for using SSL to secure all the major Internet protocols, including SMTP [Hoffman1999a], Telnet [Boe1999], and FTP [Ford-Hutchinson2000]. Many vendors also use SSL to secure their proprietary protocols.

In order to accommodate connections from clients who do not use SSL, servers must typically be prepared to accept both secured and non-secured versions of the application protocol. All of the protocols mentioned in the previous paragraph use one of two basic strategies: *separate ports* and *upward negotiation*. Both of these strategies have advantages and disadvantages, which we'll discuss in Chapter 7.

In a separate-ports strategy, the protocol designer simply assigns a different *well-known port* to the protocol and the server implementor has the server listen both on the original port and on the new secure port. Any connections that arrive on the secure port are automatically SSL negotiated before application protocol traffic begins. HTTPS uses this strategy, as we'll discuss in Chapter 9. Figure 2.4 shows the port assignments for some commonly used protocols.

```

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

The PF

```

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

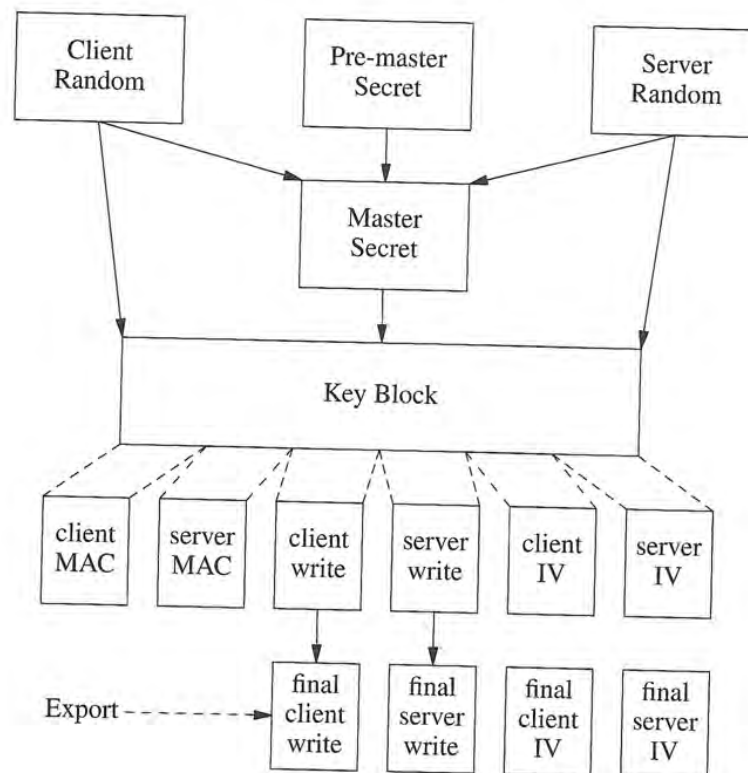


Figure 3.13 Key derivation

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.

$$\begin{aligned} \text{P_hash}(\text{secret}, \text{seed}) = & \text{HMAC_hash}(\text{secret}, \text{A}(1) + \text{seed}) + \\ & \text{HMAC_hash}(\text{secret}, \text{A}(2) + \text{seed}) + \\ & \text{HMAC_hash}(\text{secret}, \text{A}(3) + \text{seed}) + \dots \end{aligned}$$

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:

$$\begin{aligned} \text{PRF}(\text{secret}, \text{label}, \text{seed}) = & \text{P_MD5}(\text{S1}, \text{label} + \text{seed}) \oplus \\ & \text{P_SHA-1}(\text{S2}, \text{label} + \text{seed}); \end{aligned}$$

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:

$$\begin{aligned} \text{master_secret} = & \text{PRF}(\text{pre_master_secret}, \text{"master secret"}, \\ & \text{client_random} + \text{server_random}) [0..47]; \end{aligned}$$

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 initialization vector (IV) for both

client and server. First, we add up the lengths of all the values we need and produce 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_random and 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

SSLv3 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)

of the sides has been explicitly configured to support only TLS or if there is a bug. There is no value in configuring for TLS-only support.

Note that if the server only supports a version older than the client offers, it will *not* send a `protocol_version` alert. Instead, it will simply respond with its version in the `ServerHello`. If the client does not want to support the server's version choice, it will have to send an alert (presumably `protocol_version` but the SSLv3 and TLS specs don't specify). The `protocol_version` alert is always fatal. The `protocol_version` alert was added in TLS. SSLv3 implementations usually use `handshake_failure` for this purpose.

insufficient_security

The `insufficient_security` alert is used to indicate that the cipher suites offered by the client were weaker than required by the server. This alert is new with TLS. SSLv3 implementations use `handshake_failure`. Since the `insufficient_security` alert indicates failure to negotiate a cipher suite, it is always fatal.

internal_error

The `internal_error` alert is used to indicate that the transmitting implementation encountered some sort of internal error such as a memory allocation or hardware failure. This alert is new with TLS so SSLv3 implementations need to use some other error. The `internal_error` alert is always fatal.

user_cancelled

The `user_cancelled` alert is used when the user cancels the handshake for some reason. This alert is new for TLS. RFC 2246 says that it should be used as a warning, which is confusing—if the user cancels the handshake then the connection will be unsuccessful so the alert should be fatal.

no_renegotiation

As we'll see in Chapter 4, SSL allows either client or server to initiate a new handshake at any time other than during a handshake. The client initiates renegotiation by sending a new `ClientHello`. The server initiates renegotiation by sending a `HelloRequest` message. RFC 2246 says that the client can silently refuse to renegotiate or send a `no_renegotiation` alert. RFC 2246 is unclear on how the server should respond to an unwanted `ClientHello` but presumably it should send a `no_renegotiation` alert as well.

The `no_renegotiation` alert is new for TLS. SSLv3 implementations simply ignore unwanted renegotiation requests. Since most existing TLS implementations are actually upgraded SSLv3 implementations, they typically silently ignore renegotiation rather than sending the `no_renegotiation` alert. In any case, the `no_renegotiation` alert is always a warning.

Note that locking memory doesn't offer any protection from an administrator who reads sensitive data from the process while it's resident in memory. It merely protects against an administrator subsequently deciding to recover the data.

Core Dumps

On many operating systems (UNIX in particular) when an application generates a memory error, a *core* image is written to disk. This image contains the entire memory state of the process, including any secret information. Anyone who can read those files can of course recover that information. Programmers may wish to install exception handlers that detect when the application is about to core dump and zero sensitive information.

5.11 Securing the Server's Private Key

The server's private key is the key to the security of the handshake. In static keying modes, it is used to protect the `pre_master_secret` directly. In ephemeral keying modes it is used to authenticate the ephemeral key which is in turn used to protect the `pre_master_secret`. Consequently, although compromise of the server's private key is always bad, it's worse when static keying is used because this compromises all the traffic encrypted under that key.

Static Keying

The most vulnerable situation is when the server has a *static* key pair that is used to establish the `master_secret`. The most common case is the one we covered in Chapter 3. The server has a single RSA key pair with the public key in its certificate. The client uses that public key to encrypt the `master_secret` so that only the server can read it. However, an attacker who has the private key can easily decrypt the `master_secret` and perform all the attacks described previously.

In this situation, compromise of the server's static private key compromises every SSL session that was ever established with that key, because the attacker can recover the `master_secret` offline at a time of his choosing. In other words, the attacker can mount a purely passive attack. It's trivial to write a program that takes the server's private key and a recorded message trace and outputs the plaintext. In fact, the `ssldump` traces in Chapter 4 were generated using exactly this technique.

Although static keying is mostly used with RSA, SSL also has a number of static Diffie-Hellman modes in which the static DH key is in the server's certificate. These modes are equally vulnerable to this sort of attack. However, most uses of DH in SSL use ephemeral DH. The next section discusses the results of private key compromise when ephemeral keying (as described in Chapter 4) is used.

Ephemeral Keying

Consider the case in which the server's private key is not used for key establishment but only for signature. The server generates an *ephemeral* key pair which is used for key establishment and uses its static key pair to authenticate the ephemeral public key to the client.

When ephemeral keying is used, there are actually two flavors of private key compromise, each with different consequences. Either the static key (used for authentication) can be compromised or the ephemeral key (used for key establishment) can be compromised. First consider the case where the static key is compromised. Knowing the static key is of no value for decrypting traffic because no data is ever encrypted under this key. It's only encrypted using the ephemeral key. However, the static key is also of no use in recovering the ephemeral key.

Recovering a static authentication key isn't useless, however. An attacker who has the server's static authentication key can take over any connection (posing as the server) during the handshake phase. He can then either simply act as the server or mount a man-in-the-middle attack where he poses as the server to the client and the client to the server, shuttling data back and forth between them. Figure 5.2 shows this process.

Figure 5.2 shows this man-in-the-middle attack in action with some of the irrelevant messages trimmed. The attacker copies most of the handshake messages directly from client to server and vice versa, but changes the `ServerKeyExchange` message to be a signed version of his own ephemeral key rather than the server's. He signs this with the compromised private key. He then decrypts all traffic from the client and forwards it reencrypted to the server, making whatever changes he wants. There is one subtlety here in that the attacker must generate a new `Finished` message because the handshake hashes that the client sees will be different from those the server sees due to the changed ephemeral key.

So, compromise of a static authentication key allows the attacker to actively attack any given session during the handshake phase but not afterwards. It's interesting to note that the attacker cannot mount this form of attack if client authentication is used. Because the client's signature is over all of the handshake messages, the messages that the attacker sends to the client are different than the messages that the server thinks it's sending to the client. Therefore, the server's signature check will fail.

Even if the server machine is completely compromised and the server's private authentication key is recovered, this knowledge cannot be used to attack any session that's already been established and shut down. This property is called *perfect forward secrecy* (PFS). Note that static key establishment keys do not have this property. However, PFS is obtained only if the traffic keys themselves haven't been stored somewhere. For instance, if the keys are in the session cache, then the attacker could still decrypt the connection. Deleting unused or expired keying material regularly is critical for PFS.

By contrast, if the attacker compromises one of the server's ephemeral keys, he can use it only to attack connections that were protected using that key. For all such connections, the attacker can recover the `master_secret` and mount the attacks described in Section 5.9.

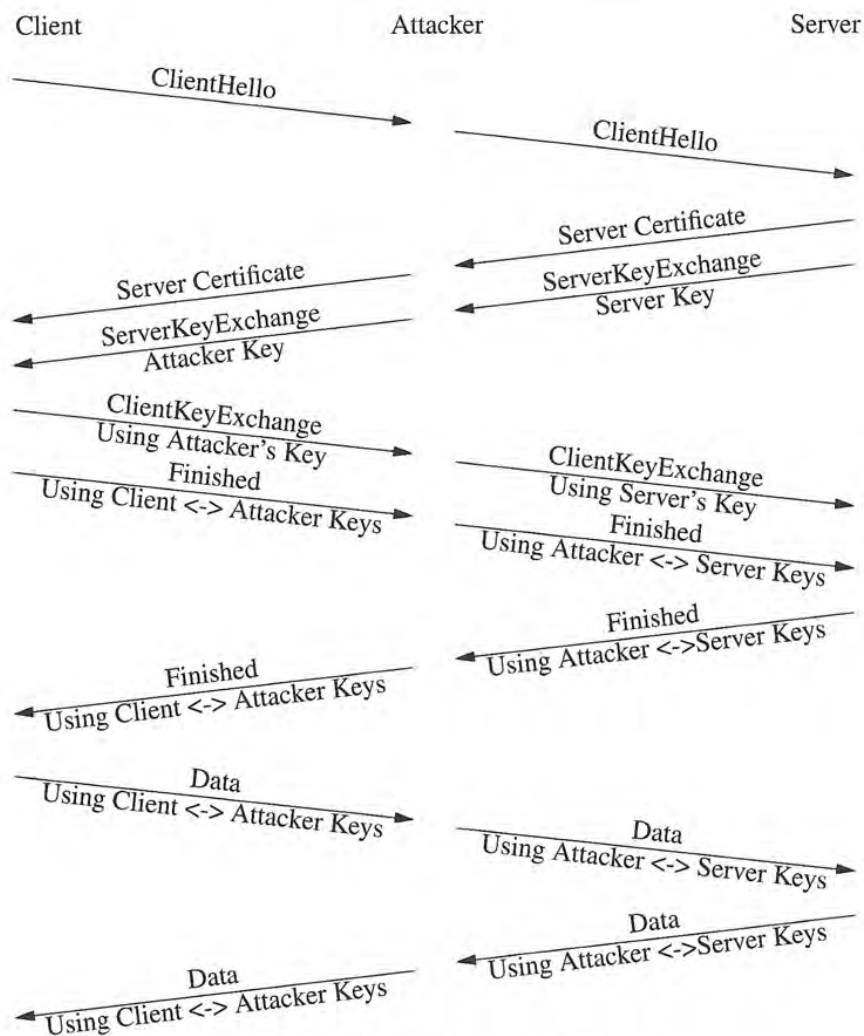


Figure 5.2 Attack using a compromised signing key

How many connections are affected depends entirely on the server's behavior. When ephemeral RSA is being used, the number of compromised connections can be very large. As we discussed in Chapter 4, generating RSA keys is expensive and servers will often generate one ephemeral RSA key when they start up and use that key until they're

sh
be
cu
ke

fr
ge
of

Private K

M
us
us
ar

de
m
m

be
ur
er

dc
ap
to
de

Password

TI
ev
tic
ve
pl
kr
ni

U.
th
ar
se
si

shut down, days or weeks later. Consequently, an enormous amount of traffic might well be compromised when one ephemeral key is compromised. Because ephemeral RSA is currently used only with exportable RSA encryption keys (<512 bits), and we know keys of this size to be at the outer edge of breakability, this is an area of some concern.

When ephemeral DH is used, it's fairly common to regenerate the ephemeral key frequently. Some implementations (OpenSSL for instance) have a mode in which they generate a new ephemeral key for every new session. When used this way, compromise of one session has no effect on any other session.

Private Key Storage

Most SSL servers run on general purpose computers. Like most server programs, they use the computer's filesystem to store their permanent data between invocations. This usually includes the server's private key. If the server machine is secure against attack and only trusted users are allowed on the system, this is a reasonably safe practice.

However, having a completely trusted server is difficult. Many organizations can't devote a machine solely to being an SSL server, and even if they could the machine might be compromised or stolen. To partially defend against these eventualities, common practice is to encrypt the private key on disk.

At this point, the astute reader will be wondering how this makes the situation any better. We've encrypted the private key, but now we need to protect the key to encrypt it under. Of course, we could encrypt that key, but then we'd need to protect the new encryption key, and so on. So, encryption alone isn't the solution to the problem.

Of course, we could force the user to memorize the key, but memorizing 128 random bytes is really a lot of work and most users won't accept that job. The usual approach is to use a password or passphrase to generate the encryption key that we use to encrypt the private key. Passwords are easily memorizable and we have a special key derivation function that we use to generate the key from the password.

Password-Based Encryption

There are a large number of ways to transform passphrases into encryption keys. However, the strong ones share a number of common features. The reason that these functions must be different from ordinary key derivation functions is that text usually has a very low *entropy*. That is to say that given part of a word or a phrase, the rest isn't completely unpredictable. (By contrast, cryptographic keys usually are constructed so that knowing part of a key provides no information about the rest of the key.) Several techniques have been designed to counter this problem.

Users are encouraged to choose longer passphrases than the keys to be derived from them. For instance, a 128-bit key can be represented in 16 characters, but passphrases are often 50 or more characters. This allows them to be both memorable and somewhat secure. Nevertheless, passphrases are probably more guessable than keys, particularly since users often choose short passwords. In order to provide some protection even if

poor passwords are chosen, password-based key derivation functions typically use two techniques called *salting* and *iteration*.

Salting includes a public random value as part of the input to the key derivation function. This value is called a *salt*. The salt is simply stored with the key file. Even if two users use the same password, they won't have the same encryption key because they will be using different salt values. This prevents two attacks. First, the attacker might make a table of all common passwords and then could break a password merely by searching that table. Second, the attacker could collect a series of password files and search them all in parallel, thus doing the key derivation stage only once. Neither of these attacks works when salting is used.

Iteration slows down the key derivation function. Password-based key derivation functions are designed to be very slow, in order to make search of the password space expensive. Users won't notice if it takes a half a second to decrypt their private key, but this is a substantial burden if each of a hundred million passwords takes a half a second to check. Typical functions are designed with a basic primitive which can be applied (iterated) an arbitrary number of times, with the output of one stage being used as the input to the next stage. Each iteration takes some time, slowing down the entire process. Again, the iteration count is stored with the encrypted file.

The best-known algorithm for password-based encryption is RSA's PKCS #5 [RSA1993d], which included both the salt and iteration features. Unfortunately, PKCS #5 could generate keys of 160 bits at most, making it useless for 3DES. When Microsoft designed the PFX key storage standard (now PKCS #12 [RSA1999a]), they designed a new function that could generate arbitrary length keys. RSA has since updated PKCS #5 to support arbitrary length keys [RSA1999b].

Figure 5.3 shows a diagram of the salting and iteration process as used in PKCS #5 version 1. The process proceeds in two stages. First, the concatenated password and salt are hashed to produce T_1 . This value is hashed repeatedly ($count - 1$ times) to produce T_n , which is the output of the function. T_n is simply equal to $Hash(T_{n-1})$. Obviously this transform cannot produce an output longer than the hash function, which is why it is not used to generate keys longer than the 160-bit output of SHA-1. The PKCS #12 and PKCS #5 version 2 transforms are similar in spirit but not identical and can produce keys of arbitrary length.

Storageless Key Recovery

Imagine that you want to run SSL on a device without any user-specific permanent storage whatsoever. There's no place to store a private key on disk, encrypted or otherwise. It's nevertheless possible for such a device to have a private key if we can derive the private key from the passphrase.

The simplest case is when using discrete log systems such as DH or DSA. Because the private key X is an arbitrary number, it's possible to simply generate it from the passphrase using the password-based KDFs we've just discussed. The group parameters

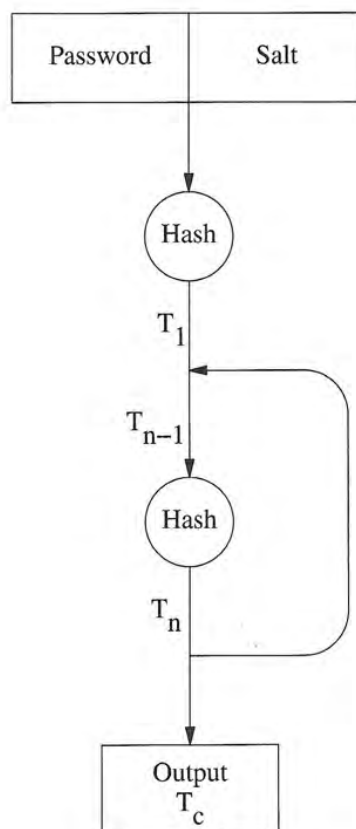


Figure 5.3 PKCS #5 transform

g , p , and q can be common to all device users and thus burned into permanent storage on the device when it's created.

This technique is less convenient for systems such as RSA where the key generation process is expensive, but it's nevertheless possible to use a passphrase to seed the PRNG that is used to generate the RSA key. As long as the passphrase is the same, the same RSA key will be generated each time.

This approach is primarily useful in situations in which a given user might use any number of different machines at any given time, such as kiosk machines or shared terminals. Because certificates can be stored in the clear on the network, the user can simply walk up to the machine and enter his passphrase.

Hardware Cryptography

Another approach to storing private keys is to use a hardware device. The device contains both permanent storage for the private key and some sort of processor capable of doing cryptographic computations. The private key is generated on the device, all private key operations are done on the device, and the private key never comes off the device.

Hardware crypto devices come in two basic flavors, permanent and removable. Permanent devices are intended to stay connected to the machine at all times, either as a card on the system bus or plugged into the back via a SCSI, parallel, or other interface. Removable devices typically are packaged as PCMCIA cards or smartcards (devices about the size of a credit card). A removable device has the obvious advantage that it can be taken away from the machine and locked up for extra security.

Either sort of device can be stolen, so the private keys are usually protected by a password or a *personal identification number* (PIN). The device won't operate correctly unless the user *logs in* with the correct PIN. Sometimes the private key is encrypted as it would be on disk and sometimes the device is simply programmed to forbid using the key unless the PIN is entered. Furthermore, the devices are typically designed to *zeroize* after some (small) number of incorrect attempts. Zeroizing means that they erase all of their sensitive internal state.

For additional security, hardware devices often incorporate a tamper resistance feature. The device is designed to detect attempts to open the case and directly access the hardware. If such an attempt is detected, the device zeroizes its memory. The idea is that even physical possession of the device is useless without the appropriate PIN. FIPS-140-1 [NIST1994b] provides standards for this sort of tamper resistance.

So, hardware devices have two primary advantages over software key files: Because the card zeroizes after some number of failed login attempts, the attacker cannot simply try all possible PINs as he can with the key file, so it's not necessary to choose a really good PIN. (This is good, because choosing good PINs is difficult.)

Second, even if the attacker does get the PIN, he needs physical access to the device. Say the attacker breaks into your server machine and guesses the PIN, but your key is in hardware. He can use the server to decrypt traffic, but once you discover the compromise, you can remove the device or change the PIN and the attack is over. With software, the attacker would have the private key and you would have to generate a completely new key pair.

Fixed devices usually are packaged (and marketed) as *cryptographic accelerators*. They're intended to be able to perform cryptographic operations faster than the host computer. As a consequence, they typically provide both increased security and a performance boost. By contrast, removable devices tend to be very slow (on the order of a few RSA operations per second) and therefore not really usable in a server environment, though they can provide convenient security and packaging for client-side keying material.

Some cryptographic accelerators have both a fixed and removable component. This provides added security because the device cannot be used without the removable key but is as fast as an ordinary fixed device.

Section 5.

Passphr

Y

al

w

tc

p

m

tc

r

v

s

ti

s

r

f

Biomet

]

i

.

Passphrase Entry

You may have noticed by now that both encrypted disk files and hardware devices usually require the operator to enter a passphrase in order to start the server. This is the weak link in the security of the entire system. It's extremely desirable for server systems to be able to restart cleanly without user intervention, but the requirement to enter passphrases to start the secure server makes this impossible.

There are only two known workable approaches: compromise security or compromise unattended reboot. Compromising unattended reboot is easy: require the operator to enter a passphrase to start the server. This means that the machine may restart correctly but an operator is needed on site to start the secure server portions.

Compromising security is similarly easy. You simply place the passphrase somewhere in the system startup scripts and arrange for it to be passed to the server when the server is started. The obvious disadvantage here is that anyone (for instance the administrator or attackers) who can read that file can read the password. Typically, such files are set to have very restrictive permissions, but most operating systems let the administrator read any file.

There's no known way to work around these two choices. Your organization simply has to choose one and accept the consequences.

Biometrics

Biometric authentication measures such as fingerprint identification and retinal scans are often used as secure access control measures. Although there is some doubt as to whether they can be securely implemented, they are nevertheless fairly widely used for access to secure environments and computers.

However, biometrics are singularly inappropriate for protecting private keying material. The measures we've talked about so far all depend on encrypting the private key based on some piece of data the user inputs. This piece of data must be identical each time because it is used to create an encryption key and encryption algorithms are specifically designed so that tiny errors in the key cause large errors in decryption.

However, biometrics cannot be easily used for this purpose because the characteristic being measured is not identical every time. Therefore, it's difficult if not impossible to recreate the same encryption key from the biometric each time. Biometric systems therefore have to keep the key in the clear and simply use the biometric to control access. This is inferior if the device containing the key is ever compromised, because the attacker doesn't need to decrypt the key. If the key is stored on disk somewhere, this is catastrophic.

There are two exceptions to this general rule. First, if the key is stored in trusted tamper-resistant hardware, then a biometric is safe because the hardware will zeroize the key before allowing it to be recovered. This requires having a lot of faith in your tamper resistance! Second, if a passphrase is used in conjunction with a biometric, then the key can still be encrypted using the passphrase and biometrics can be used as a secondary piece of access control. In this case, biometrics add valuable security.

Bottom Line

Understanding your threat model is critical to understanding which technique to use. For instance, if you have strong physical security for your system, you may not need to use as strong security to protect the keys. By contrast, if you're worried that your machine will be stolen or accessed by an unauthorized user, it's critical to protect the keys in hardware in order to limit the cost of compromise. This section gives you the information to evaluate various techniques, but you must choose the one that fits your security model.

5.12 Random Number Generation

The security of an SSL implementation depends completely on the quality of the random numbers it generates. Moreover, both the client and the server need to have strong random number generation. Exactly what is meant by "strong" random numbers isn't that obvious, but it's fairly obvious what it doesn't mean: The "random" number generators that most programming languages provide don't do the job. Special tools are required.

There are two general approaches to generating random data: you can have true randomness provided by some random or nearly random physical process, or you can have pseudo-randomness, which uses some algorithm to produce a stream of random-seeming numbers. In general, good hardware random number generators aren't available on most machines, so programmers settle for algorithmic *pseudo-random number generators* (PRNGs).

Whatever strategy we choose, we want to have at least the following properties.

The stream should be unbiased. The number of zero and one bit should be on average equal.

Output should be decorrelated. Knowing the value of the output of some section of the byte stream should not help you know the value of any other section.

Pseudo-Random Number Generators

The general idea with a PRNG is that you have some function that produces a stream of unpredictable bytes. In theory, this stream should be indistinguishable from a stream of bytes generated by a hardware device, such as flipping a coin.

Of course, this isn't the case because all PRNGs will eventually repeat the stream of bytes. The output of a PRNG is a function of the PRNG's internal state. Because their internal state can be no larger than the memory of the computer they are running on, they must repeat after outputting at most that many outputs. In practice, the internal state of the PRNG merely needs to be sufficiently large that an essentially infinite number of bytes can be generated without the state repeating. 256 bits of internal state is easily plenty to achieve this goal.

Hardware

Th
de
pro
ste
be
rar

arc
un

Typically, PRNGs are based on digest or encryption algorithms. A good example is described in [Kelsey1999]. There are a number of well-known PRNGs available, and we won't discuss the details here. However, they all have one thing in common—they need to be *seeded* with some amount of secret data. That data is used to set up their internal state. Knowing the seed data amounts to the same thing as knowing the internal state of the PRNG and allows the attacker to predict the bytestream completely. The major problem with using a PRNG is to seed it correctly.

In general, PRNGs are constructed so that they can be seeded with a large amount of low-quality random data and they will *distill* the randomness out of it. For instance, consider some data like the price of a stock from day to day. If the stock is at 85 on Monday, it's very likely that it will be somewhere between 80 and 90 on Tuesday, so the 10s digit isn't that random, but the 1s place is somewhat random and the fractions place is very random. A good PRNG can be seeded with this sort of data and collect all the good randomness out. This makes the job of collecting seed data somewhat easier.

A typical strategy for collecting seed data is to look at some fairly volatile piece of system state, such as network activity, the process table, or the screen, and distill that input into strong randomness. On an active system, this is probably good enough for generating keys to protect single sessions. Several operating systems (Windows 2000, *BSD, and Linux, at least) provide a kernel device that collects system events to continuously seed a PRNG in the background.

For high-security applications, such as private key generation, the implementations often require user input and measure the time between keystrokes or mouse movements. RFC 1750 [Eastlake1994] provides some guidance on collecting seed data.

Although most SSL implementations provide some way of seeding their PRNG from system data, they usually expect the application to provide most of the seed data and provide their own seeds only as a last-ditch effort. Even if you're using a toolkit, you may very well have to seed the PRNG yourself.

It's critical to note that a strong PRNG produces random-looking output even if it's poorly seeded. There are a number of statistical tests that are of some use in determining whether the PRNG you are using is strong. They are of little use in determining whether it has been well seeded.

Hardware Random Number Generators

The other popular strategy is to use a hardware source of randomness. Certain physical devices seem to act randomly at the quantum mechanical level, and this can be used to produce random data. The most obvious example is radioactive decay, which produces a steady stream of signals at random (but predictable on average) intervals. The time between signals (or their intensity over a given time window) can be used to produce random numbers.

Although radioactive decay is random, it's not unbiased, because the output clusters around a mean value. Some processing is usually needed to make sure that the output is unbiased.

Imagine that you have a random number generator that is random but biased in a simple way, such as an unbalanced coin that returns 60% heads and 40% tails. John von Neumann [Neumann1951] proposed a simple way to unbiased it. Collect the output in pairs. Any pair that is the same (e.g., {heads, heads} or {tails, tails}) throw away. Any pair that is different, output the first one (e.g., {heads, tails} becomes heads.) The choice of first or second is arbitrary. It's easy to convince yourself that even if the generator is biased, the trick produces a balanced stream of output, because the probability of the sequence {heads, tails} is the same as the probability of the sequence {tails, heads}.

The easiest way to do this is simply to use the output of a hardware random number generator to seed a PRNG. This overcomes the bias problem as well as ensuring that you don't have to wait for the hardware device to do its thing whenever you want random data.

Recently, Intel Corporation added a hardware-based random number generation feature to their Pentium III processors. The RNG works largely as described above. A hardware random number device based on thermal noise is built into the chip. The data is post-processed through a mixing function based on SHA-1. Initial evaluations [Kocher1999] indicate that the system is secure, provided, of course, that the machine running it has not been compromised by an attacker who is forging RNG output.

5.13 Certificate Chain Verification

We've already established that an unverifiable certificate is little better than no certificate at all. However, merely being able to construct a certificate chain to some root isn't enough to verify a certificate. A number of other checks are required in order to be sure that you're communicating with the entity you think you're communicating with.

Server Identity

As we discussed earlier, the certificate presented by the server must contain an identity which is somehow connected to the expected identity of the server. The expected identity can't be carried over the wire, but must already be known to the client at the time the connection is initiated.

In most cases, when a client wants to connect to an SSL server, the destination is specified by a DNS domain name (see [Mockapetris1987a, Mockapetris1987b]) such as `foo.example.com` or by a *Uniform Resource Locator* (URL) [Berners-Lee1994] that contains a domain name. Thus, the expected identity of the server is the domain name and that must match the identity in the certificate.

Unfortunately, the *distinguished names* (DNs) in certificates aren't really that compatible with domain names. Recall from Chapter 1 that distinguished names are structured as a series of attribute-value pairs whereas domain names are essentially a single string. Moreover, there's no DN attribute for domain names. As a consequence, most CAs use the Common Name attribute to represent the domain name. This approach requires both client and server to be aware of the convention to work correctly.

The end result of all this tuning is unfortunately still not as fast as pure C code, but it's not that far off, either. PureTLS with the GoNative Provider module is roughly half as fast as OpenSSL, depending on your algorithm choices.

6.19 SSL Servers under Load

Unsurprisingly, under heavy load, the cryptographic computations necessary for SSL become the bottleneck for server performance. Thus, a server that is capable of handling a given load can quickly become overloaded if those transactions are transitioned over to SSL. In order to simulate the impact of a large number of clients, we can use a *load generator*. We have a client machine (or machines) which connects to the server and makes requests. Each machine can generate a large number of requests at a given time, thus simulating a large number of users connecting to a given site. In this case, our client is a Pentium II/400 running FreeBSD 3.4.

We could perform this experiment using any SSL-enabled protocol, but for convenience we'll use HTTP over SSL. The traffic mix we've chosen simulates the behavior of a Web site when a large number of people decide to request a page at more or less the same time. In their paper on SSL performance, Abbott and Keung [Abbott1988] report that this is the scenario of most concern to their customers.

To produce this traffic mix, we arrange that each client connects to the server at a different *arrival time*. The arrival times are generated using a Poisson process with a mean of 3 seconds. Thus, all of the clients connect somewhere between 0 and 10 seconds after the start of the simulation.

Figure 6.16 shows the results of such a simulation on an ordinary HTTP server. The server is Apache 1.3.12 with mod_ssl 2.6.5 running under Linux 2.2.14-5.0 (Red Hat) on a Pentium III/650. The bar graph labelled "arrivals" represents the number of simulated client connections within each 1-second window. Thus, 23 connections were initiated in the first second of the simulation. The line graph labelled "queue" is the number of clients that were active during this time window. Thus, a client that connects at 1.5 s and disconnects at 2.5 s contributes to the queue during the 1–2 and 2–3 time windows.

When we examine Figure 6.16, we can see that the queue size closely tracks client arrivals. This is happening because the server is serving each request more or less as it comes in. This is a good indication that the server is handling the load successfully.

Compare this result to Figure 6.17, which shows the server behavior with 550 simulated HTTP/SSL clients. The cipher suite is TLS_RSA_WITH_3DES_EDE_CBC_SHA and the client has been set never to resume sessions, so every handshake requires an RSA operation. Initially, the queue size closely tracks the client arrival times, but as the number of clients increases during the simulation the queue size starts to increase dramatically. Note that although our client is significantly slower than our server, it nevertheless is capable of completely saturating the server machine. This is a nice demonstration of the asymmetry of SSL.

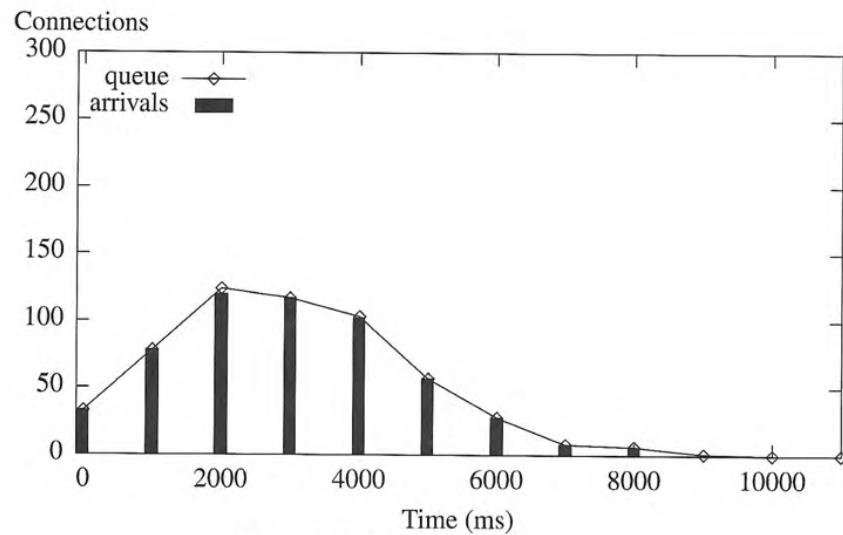


Figure 6.16 550 simulated HTTP clients

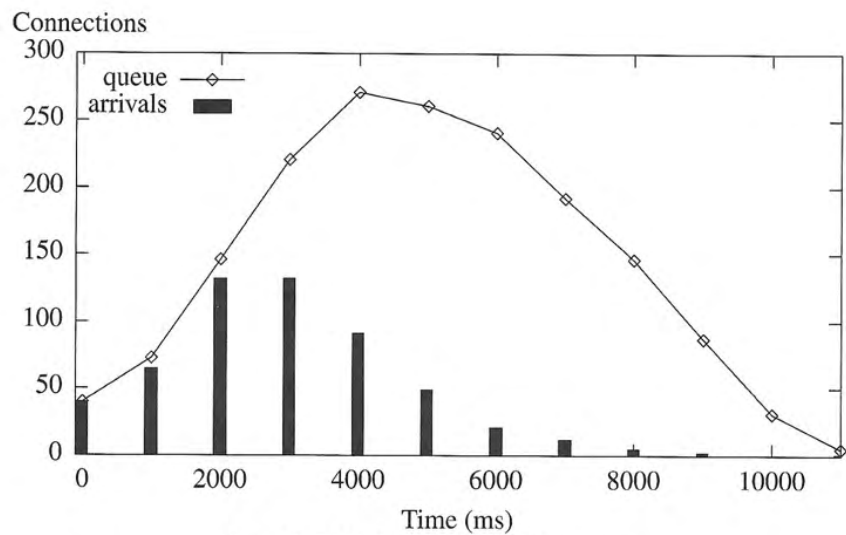


Figure 6.17 550 simulated HTTP/SSL clients

What's happening here is that the server CPU is saturated by the SSL handshakes it is already performing. Thus, more clients are arriving than are being successfully served, creating a backlog of unserved clients. As the number of new arrivals decreases, the server is able to work through the backlog of queued but unserved connections and so the queue size slowly decreases. However, the server is only able to clear the queue because the number of arrivals trails off.

The server would not be able to handle the peak load shown in Figure 6.17 on a continuous basis. This is demonstrated by Figure 6.18, which shows the server under a continuous load of approximately 75 connections a second. Note that the size of the queue grows linearly with time, indicating that the server is not keeping up with load.

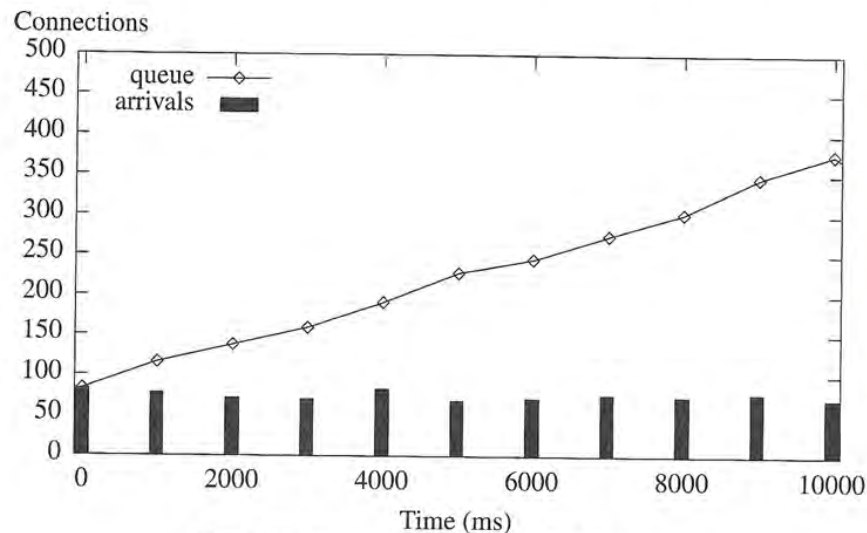


Figure 6.18 A server under continuous high load

6.20 Hardware Acceleration

Because the bottleneck in SSL performance is server CPU, it's natural to consider accelerating the SSL server by offloading the expensive crypto operations to a separate processor. Hardware acceleration not only increases the number of connections that a server can handle but also frees the server CPU to perform other tasks, such as interpreting the CGI scripts that are used to run many Web sites.

Typically, hardware accelerators are packaged as boards that may be plugged into the server machine, usually via the PCI bus on Intel architecture machines. Accelerators can also be attached to the server machine via an Ethernet connection or the SCSI bus. Accelerators are available from several manufacturers but the best known are Rainbow (<http://www.rainbow.com/>) and nCipher (<http://www.ncipher.com/>).

the network buffers on either side are in a well-defined state so that SSL data is not inadvertently processed as application protocol data.

Upward negotiation can come at a performance cost as well. When secure and insecure protocols run over separate ports, then the first packet of data is the ClientHello. However, if upward negotiation is employed, then the client and server must first establish that SSL is permitted. This can take up to one round-trip time to establish (it might be less than one round-trip if you piggyback the SSL ClientHello on the first protocol message).

Finally, although packet filtering firewalls do not need to be modified to support an upward negotiation strategy, the effect on application-layer firewall proxies is substantial. Because the application-layer proxy must be able to recognize the upward negotiation commands and get out of the way, this often requires not simply reconfiguration of the proxy but a change to the code as well. This can be a substantial deployment barrier.

7.12 Downgrade Attacks

Both methods of protocol selection are susceptible to an active downgrade attack. In this attack, the attacker convinces the client or the server that an SSL connection is not possible, thus forcing the parties to fall back and communicate via an insecure channel if they are to communicate at all. It's important to note that none of these attacks work by downgrading the SSL handshake itself. That's prevented by the digests in the Finished message. Rather, they work by making it appear as if it's impossible to negotiate an SSL connection. Thus, no SSL protocol measures can protect against them.

Separate Ports

The simplest attack to mount on a separate ports negotiation is simply to make it appear that the server isn't listening on the appropriate port at all. This is trivial to do for any attacker who can inject packets into the network. The attacker simply waits for the client to send the TCP SYN to open the connection and then forges an RST packet in response. There's no way for the client to detect that the RST came from the attacker rather than the legitimate server, so the TCP stack returns an error to the client code.

Under normal circumstances this would be a simple denial-of-service attack and of minimal concern. However, the client's (or user's) behavior can make it something far worse. The problem is illustrated by many Web pages for secure servers. Such servers want to be able to support SSL but also don't want to deny non-SSL transactions. For instance, amazon.com's Web server contains the following text:

If you received an error message when you tried to use our secure server, sign in using our standard server. If you select the secure server, the information you enter will be encrypted.

If the user follows these directions literally and clicks on the standard server link, an attacker can trivially force any connection to an insecure mode by simulating an error

when the user tries the secure link. The situation can be made even worse if clients automatically fall back to insecure modes. In either case, the attacker has achieved much more than a denial-of-service attack. It's an active confidentiality and integrity attack.

Naturally, this isn't the only way that an attacker can make it appear that an SSL connection cannot be created. Other options would be to forge a ClientHello with no cipher suites or a server alert claiming that the negotiation was impossible.

Upward Negotiation

Downgrade attack on a protocol that uses an upward negotiation strategy is a little different from one using a separate ports strategy, but the theory is the same. The attacker forges a message from the server (or the client) indicating that it is unwilling to use SSL. This means that the attacker must be able to take over the TCP connection, but this isn't that difficult. Figure 7.2 shows the flow of protocol messages here.

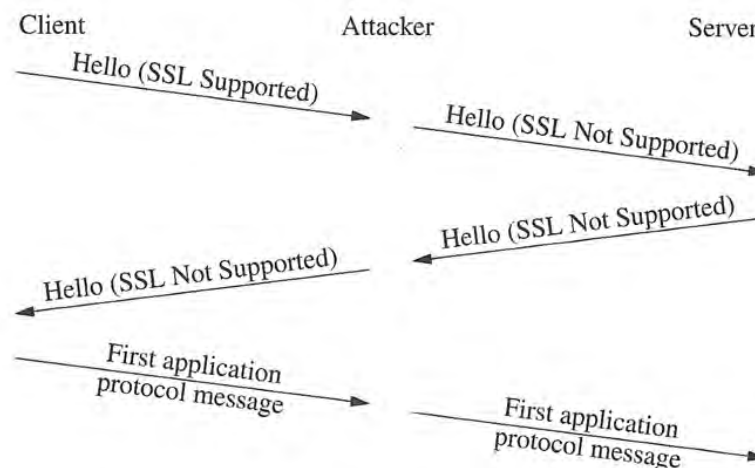


Figure 7.2 A downgrade attack on upward negotiation

As with the working upward negotiation example shown in Figure 7.1, the client attempts to form a TCP connection to the server. However, this time it forms a connection to the attacker. In Figure 7.2 we show the attacker connecting to the server posing as the client, but in many cases the attacker can just pose as the server. The client requests SSL but the attacker refuses. What happens next depends on how the client is configured.

In many cases, the client will be configured to proceed without SSL if it can't negotiate it. If all goes well, we'll *opportunistically* negotiate cryptography, but we won't worry if we can't negotiate it. In an environment where some but not all servers speak

SSL and the client has no prior knowledge about any server, this is the only sensible and interoperable way to configure it.

However, when it's configured this way, the client is completely open to this downgrade attack. As we show in Figure 7.2, it simply proceeds with an ordinary transaction. Thus, opportunistic cryptography provides excellent protection against passive attack but little protection against active attack. One partial solution to this problem (recommended in RFC 2487 [Hoffman1999a]) is to remember servers with which you have previously negotiated secure connections and insist on secure connections to them. This doesn't help when you're first encountering a new server, but it will protect against this downgrade attack against servers you've already contacted.

Even if the server requires SSL, the attacker can often mount a downgrade attack by posing as the server to the client and proxying the client's requests. This will work as long as the client doesn't demand SSL. The only case in which the server can enforce SSL successfully is if it requires SSL and client authentication. In that case the attacker will not be able to pose as the client. However, this requires all clients to have certificates as well as to support SSL. The result will be that the vast majority of clients will not be able to connect, which is often unacceptable.

Countermeasures

As we've seen, both protocol selection strategies can be vulnerable to downgrade attacks. With a separate ports strategy, all it takes is a little user inattention for security to be completely lost. The fact that legitimate servers encourage manual downgrading as a response to error makes this attack even easier.

When an upward negotiation strategy is being used, the situation is even worse. Under attack, the primary virtue of upward negotiation—automatic upgrade—becomes a weakness—automatic downgrade. Because references do not need to contain an indication that security is required in order to interoperate, it's quite common not to introduce such references at all. As a consequence, there's no way to automatically enforce security, leaving the protocol completely open to a downgrade attack.

There are two countermeasures required for this sort of attack, one technical and one social. The technical measure is straightforward; references to servers must include the fact that it is a secure site and some minimal expectations about what sorts of connection you can negotiate. The fact that SSL protects its negotiation provides some assistance here. If you can connect to the site and verify its certificate, then you can be sure that you have not been subject to a downgrade attack.

The social problem is harder. Users and programmers must be taught not to make insecure connections when secure connections fail. If you see a reference that indicates security, it's almost certainly not safe to reference it insecurely. This sort of behavior is difficult to teach users, but without it systems are ridiculously vulnerable.

IF

7.13 Reference Integrity

The goal of reference integrity is to ensure that the server that you're connecting to is the same one that the user (or the program acting for some user) intended to connect to. This is a difficult task for several reasons.

First, the identities that servers present (distinguished names in certificates) aren't very user-friendly. More important, users aren't very motivated to pay attention to the identity of the servers they're connecting to, so the process has to be automated. Finally, many attempts to make software more user-friendly have resulted in representations of identity that are inherently difficult for software to parse. For instance, it's very easy for software to match domain names. However, software is increasingly hiding the domain names in favor of "Internet keywords," the name of the Web page, or a user's personal name. These name forms are much more difficult to map to domain names.

The overriding requirement for reference integrity is that there be a chain of trust between the identity that the application layer expects and the application layer that the certificate presented by the server. Optimally, this would consist of a single link: the identity in the certificate would match the application-layer identity. Unfortunately, this is only possible in some cases.

Failing that, the application-layer identity must be securely linkable to some other identity form. This identity form must in turn be linkable to a third identity form. This process continues until the identity can be compared to the server's certificate. Naturally, if this chain is at all long, it must be mechanically verifiable; users simply will not take the time to verify it. The major alternate forms we need to be concerned with are *IP addresses* (i.e., raw network addresses), *alternate DNS names*, and *human readable names*.

IP Addresses

As we discussed in Chapter 5, the requirement for a chain of verifiable identities is why certificates generally should not contain IP addresses. In general, DNS does not provide a secure way of mapping names to IP addresses. Secure DNS [Eastlake1999] provides some promise of fixing this problem but deployment has so far been underwhelming.

As a special case, if application-layer identities are expressed in terms of IP addresses, it's acceptable for certificates to contain IP addresses, because they can be securely compared. However, in general this is bad practice for several reasons. First, IP addresses are often obtained by dereferencing DNS names using `nslookup` or some similar tool. As we've just said, that's insecure. Second, they're subject to renumbering both when service providers change and when configurations change, so they may not be as persistent as domain names. It's difficult for users to know when such renumberings are legitimate and when they are not, so they are easily tricked into accepting spoofed connections.

Alternate DNS Names

Mapping domain names to IP addresses is not the only situation in which DNS insecurity comes into play. For a number of reasons, it's common for a single DNS name to map to a number of other DNS names. The two most common cases are CNAME and MX records. CNAMEs are usually used to provide a layer of indirection between a published name and a machine's actual name. MX records are used for e-mail routing, and we'll discuss their special problems in Chapter 10 where we discuss SMTP.

Two of the most common uses for CNAME records are simple aliasing and load balancing. Imagine that you run a very small company which is big enough to want its own Web site but not big enough to run it yourself. Instead, you farm that job out to your ISP. Nevertheless, you'd like your customers to be able to refer to your company's Web site by your company name, like so: `www.example.com`. In actuality, though, this machine is run by your ISP and it already has a name: `web1.isp.com`. A CNAME provides an alias between these two names.

CNAME stands for *canonical name*. The idea here is that `web1.isp.com` is the canonical name of the machine and the CNAME record contains the canonical name.

Now when customers enter `www.example.com` into their browser, it automatically detects that it really needs to talk to `web1.isp.com` and so looks up its address and connects to it. If the ISP decides to move the Web site, it simply changes the target of the CNAME. Better yet, if you decide to host the Web site yourself, you just change the CNAME to point to the machine you're hosting it on, for instance `server.example.com`.

Now, imagine that your load gets so high that your single machine can't handle it any more. CNAMEs help here too because you can arrange for the name server to return a different CNAME every time, thus sharing the load between multiple servers.

Whenever we have this form of indirection, we need to carefully consider which name should appear in the certificate. From a security perspective the answer is clear but from an operational perspective it's difficult. Because DNS lookups cannot be trusted, it's not possible to create a chain of trust from the name `www.example.com` to any of these names pointed to by CNAMEs. Thus, the address that appears in the certificate *must* be `www.example.com`.

Unfortunately, using the original name (the CNAME) in the certificate is operationally problematic. In the case where the ISP is running your Web server, this would require them to obtain multiple certificates (one for each customer) or a single certificate with multiple aliases (multiple Common Names or `subjectAltNames`). In the case where you're running multiple servers, you'll need to share your certificate (and private key!) between all of those servers, thus increasing your security risk.

In the Web context, a common fix for this problem is to have an opening page that is accessible via HTTP only and have that point to a secure server using a URL that matches the secure server's one certificate. So, in this example the link that actually takes you to the secure server might be a link to `https://secure1.isp.com`. Of course, this sort of mismatch between URLs and the user's expectations has its own security issues which we'll take up partially in the next section and more in Chapter 9 when we discuss HTTP over SSL.

Human Readable Names

The final form of identity to consider is human readable names. Many application protocols and clients provide references to users that are in themselves representative but are nearly meaningless to the software. For instance, it's very common for e-mail clients to display user's names instead of their e-mail addresses. Similarly, users who view data in Web browsers often look at the title of the document or the content of a link rather than at the underlying URL.

Since this data is not securely bound to the server's certificate, it forms an inadequate basis for trust. Occasionally, an implementation will have access to a trustable database that maps these names onto network names. (Even Secure DNS would not do this, however.) Then and only then can these names be trusted. Otherwise, users need to be presented with the opportunity to view the identity that was verified in untranslated form. Only then can full security be provided.

7.14 Username/Password Authentication

As we stated previously, authenticating users (and clients) presents a difficult problem. From a security perspective, it would be ideal to simply deploy certificates for every client we wished to authenticate. However, this presents deployment problems in many situations, so it's important to have other options available.

Username/Password

The most obvious (and commonly used) approach is simple passwords. As we stated earlier, transmitting passwords over SSL removes the major problem with passwords—passive sniffing attacks—but it still leaves open some other more difficult (but still possible) forms of attack.

There are three major types of attack worth considering. First, note that the security of the channel depends on the client forming a secure connection to the right server. This means that the client must carefully check the server's certificate. If this check is not performed the channel is subject to a man-in-the-middle attack and the attacker can recover the user's password.

A second variety of attack is a *password guessing* attack. As we said in Chapter 5, users often choose very poor passwords (see [Klein1990]) such as their username, first name, date of birth, etc. Such passwords are easily guessed. Therefore, an attacker can simply initiate a connection to the server and try each of these easily guessed passwords in sequence. Such an attack might not succeed with any given user, but if there is a large user base, the probability that at least one user will choose a bad password is high. There are a number of somewhat effective countermeasures against password guessing, including proactive password checking, limited tries, and slowing down each password check (to make guessing more time consuming), which we will not discuss here.

A final weakness with passwords is that they're transferable. When you authenticate to a server you give it your password. If you've used the same password on multiple machines (many users do) then any one of these servers can impersonate you to another

HTTP over SSL

9.1 Introduction

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

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

9.2 Securing the Web

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

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

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

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

Basic Technologies

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

HTTP

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

HTML

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

URLs

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

Practical Considerations

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

Secu

9.3

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 POSTs may have a message body whereas GETs 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

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

9.10

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.

9.11

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.

Conn

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)
I 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.

```
1      #include "common.h" common.c
2
3      BIO *bio_err=0;
4      static char *pass;
5      static int password_cb(char *buf,int num,int rwflag,void *userdata);
6      static void sigpipe_handle(int x);
7
8      /* A simple error and exit routine*/
9      int err_exit(string)
10         char *string;
11     {
12         fprintf(stderr,"%s\n",string);
13         exit(0);
14     }
15
16     /* Print SSL errors and exit*/
17     int berr_exit(string)
18         char *string;
19     {
20         BIO_printf(bio_err,"%s\n",string);
21         ERR_print_errors(bio_err);
22         exit(0);
23     }
24
25     /*The password code is not thread safe*/
26     static int password_cb(char *buf,int num,int rwflag,void *userdata)
27     {
28         if(num<strlen(pass)+1)
29             return(0);
30
31         strcpy(buf,pass);
32         return(strlen(pass));
33     }
34
35     static void sigpipe_handle(int x){
36     }
37
38     SSL_CTX *initialize_ctx(keyfile,password)
39         char *keyfile;
40         char *password;
41     {
42         SSL_METHOD *meth;
43         SSL_CTX *ctx;
```

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.

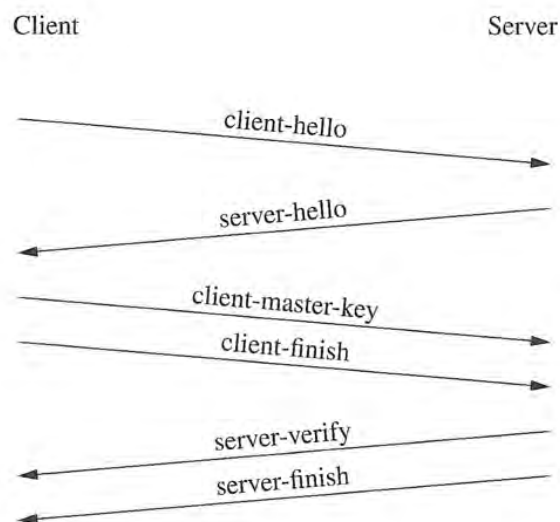


Figure B.1 SSLv2 handshake

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.

Clie

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.

Data

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.

B.3

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 individually 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

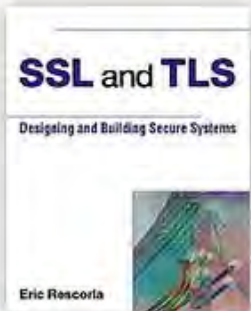
SSLv2 was missing a number of features that seemed desirable. None of these features are absolutely necessary, but their absence was inconvenient and they were added when SSLv3 was designed.

ATTACHMENT A-3



Items ▾ Search for books, articles, and more

Home Libraries Topics Lists About For Librarians





SSL and TLS : designing and building secure systems

★★★★☆ [3 reviews](#)

Author: [Eric Rescorla](#)

Summary: "This book not only describes how SSL/TLS is supposed to behave but also uses the author's free ssldump diagnostic tool to show

[Show more ▾](#)

 **Print Book**, English, ©2001

Edition: [View all formats and editions](#)

Publisher: Addison-Wesley, Boston, ©2001

Physical Description:	xxv, 499 pages : illustrations ; 24 cm
ISBN:	9780201615982, 0201615983
OCLC Number / Unique Identifier:	44509390
Subjects:	Computer network protocols Computer networks Security measures Protocoles de réseaux d'ordinateurs Réseaux d'ordinateurs Sécurité Mesures SSL TLS Web Sécurité Mesures World Wide Web Security measures Show less ^

Contents:

Security concepts
Introduction to SSL
Basic SSL
Advanced SSL
SSL security
SSL performance
Designing with SSL
Coding with SSL
HTTP over SSL
SMTP over TLS
Contrasting
approaches

[Show less ^](#)

 **8 editions** in 274 libraries

Featured libraries All libraries

Showing 209 libraries near [Washington, District of Columbia](#)

☐ Only public libraries



Academic

George Washington University

 [1.2 miles](#) from your current location.

Melvin Gelman Libr, 2130 H St NW, Washington, DC,
20052-2521, United States

Borrow 



Regional or
National

Library of Congress

 [2.2 miles](#) from your current location.

101 Independence Avenue, SE, Washington, DC,
20540, United States

Borrow 

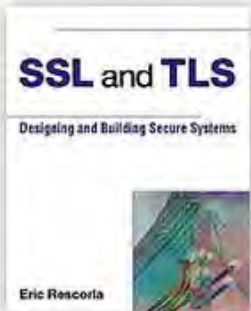


ATTACHMENT A-4



Items ▾ Search for books, articles, and more

Home Libraries Topics Lists About For Librarians





SSL and TLS : designing and building secure systems

★★★★☆ [3 reviews](#)

Author: [Eric Rescorla](#)

Summary: "This book not only describes how SSL/TLS is supposed to behave but also uses the author's free ssldump diagnostic tool to show

[Show more ▾](#)

 **Print Book**, English, ©2001

Edition: [View all formats and editions](#)

Publisher: Addison-Wesley, Boston, ©2001

Physical Description:	xxv, 499 pages : illustrations ; 24 cm
ISBN:	9780201615982, 0201615983
OCLC Number / Unique Identifier:	44509390
Subjects:	Computer network protocols Computer networks Security measures Protocoles de réseaux d'ordinateurs Réseaux d'ordinateurs Sécurité Mesures SSL TLS Web Sécurité Mesures World Wide Web Security measures Show less ^

Screen capture created by J L Mullins from *WorldCat* on February 12, 2024 at URL:
<https://search.worldcat.org/title/44509390>

Contents:

Security concepts
Introduction to SSL
Basic SSL
Advanced SSL
SSL security
SSL performance
Designing with SSL
Coding with SSL
HTTP over SSL
SMTP over TLS
Contrasting
approaches

[Show less ^](#)

Find a Copy at a Library

Filter by: [Any format](#) [Any edition](#) [Distance within 200+ mi](#)

8 editions in 274 libraries

Featured libraries

All libraries

Showing 209 libraries near [Virginia, United States](#) ☐ Only public libraries



Academic

Virginia Commonwealth University

VCU Libraries

 [66 miles](#) from your current location.
P.O. Box 842033, Richmond, VA, 23284-2033, United States

Borrow 

 Favorite Library



Academic

Virginia State University

 [69 miles](#) from your current location.
1 Hayden Drive, Petersburg, VA, 23806, United States

Borrow 

 Favorite Library



Academic

James Madison University Libraries

 [70 miles](#) from your current location.
1251 Carrier Dr., Harrisonburg, VA, 22807, United States

Borrow 

 Favorite Library



Academic

Duke University Libraries

Duke University Library; Perkins Library

 [100 miles](#) from your current location.
411 Chapel Drive, PO Box 90188, Durham, NC, 27708, United States

Borrow 

 Favorite Library



Academic

William And Mary

 [107 miles](#) from your current location.
400 Landrum Drive, Williamsburg, VA, 23186-0004, United States

Borrow 

 Favorite Library

ATTACHMENT A-5

LIBRARY

CATALOG

←

<

1 of 1

>

BOOK

SSL and TLS : designing and building secure systems

Full Record

MARC Tags

Personal name

Rescorla, Eric, 1972-

Main title

SSL and TLS : designing and building secure systems / Eric Rescorla.

Published/Created

Boston : Addison-Wesley, c2001.

Request this Item

 LC Find It

More Information

>

LCCN Permalink	https://lcn.loc.gov/00057593
Description	xxv, 499 p. : ill. ; 24 cm.
ISBN	0201615983 (pbk. : alk. paper)
LC classification	TK5105.59 .R47 2001
Variant title	Secure sockets layer and Transport layer security

LC Subjects	Computer networks--Security measures. World Wide Web--Security measures. Computer network protocols.
Browse by shelf order	TK5105.59
Notes	Includes bibliographical references (p. 465-474) and index.
LCCN	00057593
Dewey class no.	005.8
Type of material	Book

Item Availability

>

CALL NUMBER	TK5105.59 .R47 2001 Copy 1
Request in	Jefferson or Adams Building Reading Rooms
Status	Not Charged

CALL NUMBER	TK5105.59 .R47 2001 FT MEADE Copy 2
Request in	Jefferson or Adams Building Reading Rooms - STORED OFFSITE
Status	Not Charged

ATTACHMENT A-6

LIBRARY | CATALOG

← 1 of 1 →

BOOK

SSL and TLS : designing and building secure systems

[Full Record](#)[MARC Tags](#)

000	01125cam a22002774a 4500
001	12082347
005	20010206085635.0
008	000627s2001 maua b 001 0 eng
906	— a 7 b cbc c orignew d 1 e ocip f 20 g y-gencatlg
925	0_ a acquire b 2 shelf copies x policy default
955	— a to ASCD pc21 06-27-00; jf03 06-27-00; jf08 06-27-00 to SL; jf12 to Dewey 06-28-00; aa07 06-28-00; CIP ver. pb18 to BCCD 12-28-00; copy 2 added jf16 to BCCD 02-06-01
010	— a 00057593
020	— a 0201615983 (pbk. : alk. paper)
040	— a DLC c DLC d DLC
042	— a pcc
050	00 a TK5105.59 b .R47 2001
082	00 a 005.8 2 21
100	1_ a Rescorla, Eric, d 1972-
245	10 a SSL and TLS : b designing and building secure systems / c Eric Rescorla.
246	3_ a Secure sockets layer and Transport layer security
260	— a Boston : b Addison-Wesley, c c2001.
300	— a xxv, 499 p. : b ill. ; c 24 cm.
504	— a Includes bibliographical references (p. 465-474) and index.
650	_0 a Computer networks x Security measures.

650 _0 |a World Wide Web |x Security measures.

650 _0 |a Computer network protocols.

Request this Item

 LC Find It

Item Availability

>

CALL NUMBER	TK5105.59 .R47 2001 Copy 1
Request in	Jefferson or Adams Building Reading Rooms
Status	Not Charged

CALL NUMBER	TK5105.59 .R47 2001 FT MEADE Copy 2
Request in	Jefferson or Adams Building Reading Rooms - STORED OFFSITE
Status	Not Charged

ATTACHMENT A-7

Screen capture created by J L Mullins from College of William and Mary Libraries OPAC on February 1, 2024 at URL:

https://wm.primo.exlibrisgroup.com/discovery/sourceRecord?vid=01COWM_INST:01COWM_WM_NEW_UI&docId=alma991015025589703196&recordOwner=01COWM_INST

WILLIAM & MARY LIBRARIES

BOOK

SSL and TLS : designing and building secure systems

Rescorla, Eric, 1972-

c2001; Boston : Addison-Wesley

DETAILS

Title	SSL and TLS : designing and building secure systems
Author	Rescorla, Eric, 1972- >
Subject	Computer network protocols > Computer networks -- Security measures > World Wide Web -- Security measures >
Description	Includes bibliographical references (p. 465-474) and index.
Other title	Secure sockets layer and Transport layer security
Publisher	Boston : Addison-Wesley
Creation Date	c2001
Format	xxv, 499 p. : ill. ; 24 cm.
Language	English
Source	Library Catalog
Identifier	LC : 00057593 ISBN : 0201615983 (pbk. : alk. paper) OCLC : (OCoLC)44509390
MMS ID	991015025589703196
Call Number	TK5105.59 .R47 2001

ATTACHMENT A-8

https://wm.primo.exlibrisgroup.com/discovery/sourceRecord?vid=01COWM_INST:01COWM_WM_NEWUI&docId=alma991015025589703196&recordOwn_er=01COWM_INST

01060cam a22003014a 4500
001 991015025589703196
005 20231213184733.0
008 000627s2001 maua b 001 0 eng
010 ##\$a 00057593
020 ##\$a0201615983 (pbk. : alk. paper)
035 ##\$a(OCOLC)44509390
040 ##\$aDLC \$cDLC
042 ##\$apcc
049 ##\$aVMM
050 00\$aTK5105.59 \$b.R47 2001
082 00\$a005.8 \$221
100 1\$aRescorla, Eric, \$d1972-
245 10\$aSSL and TLS : \$bdesigning and building secure systems / \$cEric Rescorla.
246 3\$aSecure sockets layer and Transport layer security
260 ##\$aBoston : \$bAddison-Wesley, \$cc2001.
300 ##\$xxv, 499 p. : \$bill. ; \$c24 cm.
504 ##\$aIncludes bibliographical references (p. 465-474) and index.
596 ##\$a1
650 #0\$aComputer networks \$xSecurity measures.
650 #0\$aWorld Wide Web \$xSecurity measures.
650 #0\$aComputer network protocols.
994 ##\$aE0 \$bVMM
999 ##\$aTK5105.59 .R47 2001 \$wLC \$c1 \$11001546875 \$d5/3/2008 \$e4/30/2008 \$1STACKS \$mSHEM \$n2 \$p\$39.95 \$rNY \$sY \$tBOOK \$u4/30/2001 \$xOCLC

ASK US

FEEDBACK

ATTACHMENT A-9



BibFormats

(<https://www.oclc.org/bibformats/en/home.html>)

9xx Fields

9xx Introduction

9xx fields

9xx fields are not part of the standard MARC 21 format. OCLC has defined these 9xx fields for use by the Library of Congress and for internal OCLC use: 936, 938, 956, 987, 989, and 994. Fields 901-907, 910, and 945-949 have been defined by OCLC for your local use and will pass OCLC validation. The remaining 9xx fields may also be locally defined by your library, but may not pass OCLC validation.

OCLC services

OCLC services use some 9xx fields for processing. OCLC does not retain these fields in the WorldCat record. For example, field 951 is used for agent specific data for cataloging agent authorizations. This manual does not include descriptions of these fields.

The WorldCat Cataloging Partners, and other services, also use some 9xx fields to supply nonbibliographic information. Because WorldCat Cataloging Partners 9xx fields are defined by agreement between a library and a vendor, this manual does not include descriptions of these fields. OCLC removes WorldCat Cataloging Partners-defined 9xx fields from WorldCat records. You may still see 9xx fields that are not defined in WorldCat records. These fields will need to be removed before replacing the WorldCat record.

MARC 21 9xx fields are OCLC-defined and therefore not included in *MARC 21 Format for Bibliographic Data*.

This page last updated: 2024-03-14 (<https://www.oclc.org/bibformats/en/revisionhistory.html>) · [Report an error in WorldCat](https://www.oclc.org/oclc-forms/en/worldcat-bibliographic-authority-record-change.html) (<https://www.oclc.org/oclc-forms/en/worldcat-bibliographic-authority-record-change.html>) · [Submit a suggestion for this page](https://help.oclc.org/Librarian_Toolbox/Contact_OCLC_Support) (https://help.oclc.org/Librarian_Toolbox/Contact_OCLC_Support) · [Get help with general cataloging questions](https://help.oclc.org/WorldCat/Cataloging_documentation) (https://help.oclc.org/WorldCat/Cataloging_documentation)

[_ \(http://www.facebook.com/pages/OCLC/20530435726\)](http://www.facebook.com/pages/OCLC/20530435726)[_ \(https://www.youtube.com/OCLCvideo\)](https://www.youtube.com/OCLCvideo)[_ \(https://www.linkedin.com/company/oclc\)](https://www.linkedin.com/company/oclc)[_ \(https://www.instagram.com/oclc_global\)](https://www.instagram.com/oclc_global)[_ \(https://blog.oclc.org/next/\)](https://blog.oclc.org/next/)[_ \(https://staging.prod.oclc.org/en/rss.html\)](https://staging.prod.oclc.org/en/rss.html)

© 2024 OCLC (<https://policies.oclc.org/en/copyright.html>) Domestic and international trademarks and/or service marks of OCLC, Inc. and its affiliates (<https://policies.oclc.org/en/copyright/trademarks.html>)

This site uses cookies. By continuing to browse the site, you are agreeing to our use of cookies. Find out more about OCLC's cookie notice. (<https://policies.oclc.org/en/privacy/cookie-statement.html>)

ATTACHMENT A-10

650 - Subject Added Entry-Topical Term (R)

MARC 21 Bibliographic - Full

July 2022

First Indicator

Level of subject

- # - No information provided
- 0 - No level specified
- 1 - Primary
- 2 - Secondary

Second Indicator

Thesaurus

- 0 - Library of Congress Subject Headings
- 1 - Library of Congress Children's and Young Adults' Subject Headings
- 2 - Medical Subject Headings
- 3 - National Agricultural Library subject authority file
- 4 - Source not specified
- 5 - Canadian Subject Headings
- 6 - Répertoire de vedettes-matière
- 7 - Source specified in subfield \$2

Subfield Codes

Main term portion

- \$a - Topical term or geographic name entry element (NR)
- \$b - Topical term following geographic name entry element (NR)
- \$c - Location of event (NR)
- \$d - Active dates (NR)
- \$e - Relator term (R)
- \$g - Miscellaneous information (R)
- \$4 - Relationship (R)

Control subfields

- \$0 - Authority record control number or standard number (R)
- \$1 - Real World Object URI (R)
- \$2 - Source of heading or term (NR)
- \$3 - Materials specified (NR)
- \$6 - Linkage (NR)
- \$7 - Data provenance (R)
- \$8 - Field link and sequence number (R)

Subject subdivision portion

- \$v - Form subdivision (R)
- \$x - General subdivision (R)
- \$y - Chronological subdivision (R)
- \$z - Geographic subdivision (R)

FIELD DEFINITION AND SCOPE

Subject added entry in which the entry element is a topical term.

Topical subject added entries may consist of general subject terms including names of events or objects. Subject added entries are assigned to a bibliographic record to provide access according to generally accepted thesaurus-building rules (e.g., *Library of Congress Subject Headings* (LCSH), *Medical Subject Headings* (MeSH)). Field 650 may be used by any institution assigning subject headings based on the lists and authority files identified in the second indicator position or in subfield \$2 (Source of heading or term).

A title (e.g., Bible and atheism), a geographic name (e.g., Iran in the Koran), or the name of a corporate body (e.g., Catholic Church and humanism) used in a phrase subject heading are also recorded in field 650.

GUIDELINES FOR APPLYING CONTENT DESIGNATORS

■ INDICATORS

First Indicator - Level of subject

Used to distinguish primary and secondary descriptors.

- No information provided

0 - No level specified

Level of the subject term could be determined but is not specified.

650 00\$aFlour industry\$vPeriodicals.

1 - Primary

Main focus or subject content of the material.

650 17\$aCareer Exploration.\$2ericd

650 #7\$aCareer Exploration.\$2ericd

650 17\$aCooks.\$2ericd

2 - Secondary

Less important aspect of the content of the material.

650 27\$aFood Service.\$2ericd

650 27\$aJunior High Schools.\$2ericd

650 27\$aSimulation.\$2ericd

[Primary and secondary subject content identified.]

Second Indicator - Thesaurus

Subject heading system or thesaurus used in constructing the subject heading.

0 - Library of Congress Subject Headings

Subject added entry conforms to and is appropriate for use in the *Library of Congress Subject Headings* (LCSH) and the Name authority files that are maintained by the Library of Congress.

1 - Library of Congress Children's and Young Adults' Subject Headings

Subject added entry conforms to the Library of Congress *Children's and Young Adults' Subject Headings* used for the Library of Congress Children's and Young Adults' Cataloging (CYAC) Program and is appropriate for use according to CYAC guidelines.

2 - Medical Subject Headings

Subject added entry conforms to and is appropriate for use in the National Library of Medicine authority files.

3 - National Agricultural Library subject authority file

Subject added entry conforms to and is appropriate for use in the National Agricultural Library subject authority file.

4 - Source not specified

Subject added entry conforms to a controlled list that cannot be identified by second indicator values 0-3, 5-6 or by a code in subfield \$2. Field 653 (Index Term-Uncontrolled) is

used to record terms that are not derived from controlled subject heading lists.

5 - Canadian Subject Headings

Subject added entry conforms to and is appropriate for use in the *Canadian Subject Headings* that is maintained by the Library and Archives Canada.

6 - Répertoire de vedettes-matière

Subject added entry conforms to the *Répertoire de vedettes-matière* that is maintained by the Bibliothèque de l'Université Laval and/or name authority files maintained by libraries in Canada using French as the language of description.

7 - Source specified in subfield \$2

Subject added entry conforms to a set of subject heading system/thesaurus building rules. The identifying code is given in subfield \$2.

■ SUBFIELD CODES

\$a - Topical term or geographic name entry element

Topical subject or a geographic name used as an entry element for a topical term. Parenthetical qualifying information associated with the term is not separately subfield coded.

650 #0\$aAmish.

650 #0\$aKalmyk cattle.

650 #0\$aAstronauts.

650 #0\$aEgypt in the Bible.

650 #0\$aBASIC (Computer program language)

650 #0\$aBull Run, 2d Battle, 1862.

650 #0\$aConcertos (String orchestra)

650 #0\$aVocal music\$zFrance\$y18th century.

650 #0\$aDentistry\$vJuvenile films.

650 #5\$aCanadian wit and humor (English)

\$b - Topical term following geographic name entry element

Topical term that is entered under a geographic name contained in subfield \$a. This construction is not used in AACR2 formulated records.

650 #0\$aCaracas.\$bBolivar Statue.

\$c - Location of event

\$d - Active dates

Time period during which an event occurred.

\$e - Relator term

Specifies the relationship between the topical heading and the described materials, e.g., depicted.

650 #0\$aSeabiscuit (Race horse),\$edepicted.
[Photograph of Seabiscuit, the race horse.]

650 #0\$aUnicorns,\$edepicted.
[Bayeux Tapestry, showing a unicorn.]

\$g - Miscellaneous information

Data element that is not more appropriately contained in another defined subfield.

\$v - Form subdivision

Form subdivision that designates a specific kind or genre of material as defined by the thesaurus being used. Subfield \$v is appropriate only when a form subject subdivision is added to a main term.

650 #0\$aScuba diving\$vPeriodicals.

650 #0\$aVomiting\$xTreatment\$vHandbooks, manuals, etc.

\$x - General subdivision

Subject subdivision that is not more appropriately contained in subfields \$v (Form subdivision), \$y (Chronological subdivision), or \$z (Geographic subdivision). Subfield \$x is appropriate only when a general topical subdivision is added to a main term.

650 #0\$aRacetracks (Horse-racing)\$zUnited States\$xHistory.

650 #0\$aNumismatics\$xCollectors and collecting.

\$y - Chronological subdivision

Subject subdivision that represents a period of time. Subfield \$y is appropriate only when a chronological subject subdivision is added to a main term.

650 #0\$aMusic\$y500-1400.

\$z - Geographic subdivision

Geographic subject subdivision. Subfield \$z is appropriate in field 650 only when a geographic subject subdivision is added to a main term.

650 #0\$aWorld War, 1939-1945\$xCampaigns\$zTunisia.

650 #0\$aReal property\$zMississippi\$zTippah County\$vMaps.

\$0 - Authority record control number or standard number

See description of this subfield in Appendix A: [Control Subfields](#).

\$1 - Real World Object URI

See description of this subfield in Appendix A: [Control Subfields](#).

\$2 - Source of heading or term

MARC code that identifies the source list from which the subject added entry was assigned. It is used only when the second indicator position contains value 7 (Source specified in subfield \$2). Code from: [Subject Heading and Term Source Codes](#).

650 #7\$aEducational buildings\$zWashington (D.C.)\$y1890-1910.\$2letgm

650 17\$aAcoustic measurement.\$2test

\$3 - Materials specified

Part of the described materials to which the field applies.

\$4 - Relationship

Code or URI that specifies the relationship from the entity described in the record to the entity referenced in the field. A source of relationship codes is: [MARC Code List for Relators](#).

\$6 - Linkage

See description of this subfield in Appendix A: [Control Subfields](#).

\$7 - Data provenance

See description of this subfield in Appendix J: [Data Provenance Subfields](#).

\$8 - Field link and sequence number

See description of this subfield in Appendix A: [Control Subfields](#).

INPUT CONVENTIONS

Ambiguous Headings - See [Appendix E](#).

Punctuation - Field 650 ends with a mark of punctuation or a closing parenthesis. If the final subfields are subfield \$2 or \$3, the mark of punctuation precedes those subfields.

650 17\$aCareer Exploration.**\$2**ericd

650 #0\$aBallads, English**\$z**Hudson River Valley (N.Y. and N.J.)

Main term portion followed by a subject subdivision does not end with a mark of punctuation unless the main term portion ends with an abbreviation, initial/letter, or open date.

650 #0\$aRain and rainfall**\$z**Washington (State)**\$z**Seattle**\$v**Maps.

Spacing - No spaces are used in initialisms, acronyms, or abbreviations.

650 #0\$aMARC formats.

650 #0\$aC.O.D. shipments.

Display Constant

- *[dash associated with the content of subfield \$v, \$x, \$y, or \$z]*

Dash (-) that precedes a subject subdivision in an extended subject heading is not carried in the machine-readable record. It may be system generated as a display constant associated with the content of subfield \$v, \$x, \$y, and \$z.

Content designated field :

650 #0\$aNuclear energy**\$x**History.

Display example:

Nuclear energy-History.

Initial Article - Initial articles (e.g., The) occurring at the beginning of topical subject added entry fields are usually omitted in most languages (except when the intent is to file on the article).

Any diacritics and/or special characters occurring at the beginning of fields are retained. Note that such characters are usually ignored for purposes of sorting and filing.

CONTENT DESIGNATOR HISTORY

\$b - Topical term following geographic name as entry element [OBSOLETE, 1981]

Subfield was restored for use in retrospective record conversion in 1987.

\$g - Miscellaneous information [NEW, 2014]

\$0 - Authority record control number [NEW, 2007]

\$0 - Authority record control number [REDEFINED, 2010]

\$1 - Real World Object URI [NEW, 2017]

\$4 - Relator code [NEW, 2005]

\$4 - Relator code [RENAMED, 2017] [REDESCRIBED, 2017]

Subfield \$4 was renamed and redescribed to allow for the recording of relationship URIs in addition to MARC and non-MARC codes.

\$7 - Data provenance [NEW, 2022]

