Designing Secure Client/Server Applications using Secure Sockets Layer (SSL)

- The Secure Sockets Layer (SSL) protocol for Internet security (developed by Netscape Communications to ensure private and authenticated communications) is an open platform put into the public domain for the Internet community.

- SSL provides data encryption, server authentication, message integrity, and client authentication for a TCP/IP connection.

- SSL is a protocol that provides a secure channel to protect the data being exchanged between two machines. The secure channel is transparent, which means that it passes the data through unchanged.

- The data is encrypted between client and server while it is in transit. At the other end the data is decrypted and the original plaintext is available again.

- This transparency allows nearly any protocol that can be run over TCP to be run over SSL with only minimal modification making it a very convenient tool for use in networked TCP/IP applications.

- SSL has gone through a number of revisions, beginning with version 1 and culminating in its adoption by the IETF as the **Transport Layer Security** (TLS) standard.

- SSL was originally designed for the World Wide Web environment. Netscape's intent was to provide a single solution for all their communications security problems, including Web, mail, and news traffic.

- SSL had to work well with the main protocol used by the Web, Hypertext Transfer Protocol (HTTP).

- When SSLv2 was first designed in 1994, the main security issue was how to pass information from the client to the server (such as credit card numbers) without disclosing it to a third party (attacker).

- This requires that the client (Web browser) be certain that it is sending the credit card information to the correct server (the Web server).

- On the other hand, there was no need for the server to know who the client is because the credit card number is all the identity information required for billing purposes.

- Thus, the second major design goal: **server authentication**. Optionally, SSL also provided **client authentication**.

- It was very important that the protocol be able to handle this situation automatically, with a minimum of inconvenience to the user. This design goal is best referred to as spontaneity.

- Because Netscape wanted to use SSL as a unified security solution, it also had to work correctly for other protocols besides HTTP.

- Since the most popular Internet protocols ran over TCP connections, a protocol that provided a secure, transparent channel could be used to provide security for all such protocols.

- This property, **transparency**, is probably the single largest reason for SSL's success. It made it relatively simple to provide some security for almost any protocol merely by running it over SSL.

- Immediately after introducing SSL, Netscape started using it to protect not only HTTP traffic but also Usenet news traffic over **Network News Transfer Protocol** (NNTP).

- Unfortunately, not all protocols require the same security properties, and so the results of "just" running a protocol over SSL are often undesirable, either from a security or a performance perspective.

SSL Encryption

- The SSL protocol supports the use of a variety of different cryptographic algorithms, or **ciphers,** for use in operations such as authenticating the server and client to each other, transmitting certificates, and establishing session keys.

- Clients and servers may support different **cipher suites,** or sets of ciphers, depending on factors such as the version of SSL they support, company policies regarding acceptable encryption strength, and government restrictions on export of SSL-enabled software.

- As part of its other functions, the SSL handshake protocol determines how the server and client negotiate which cipher suites they will use to authenticate each other, to transmit certificates, and to establish session keys.

- The cipher suite descriptions that follow refer to these algorithms:

    - **DES.** Data Encryption Standard, an encryption algorithm used by the U.S. Government.

    - **DSA.** Digital Signature Algorithm, part of the digital authentication standard used by the U.S. Government.

    - **KEA.** Key Exchange Algorithm, an algorithm used for key exchange by the U.S. Government.

    - **MD5.** Message Digest algorithm developed by Rivest.

    - **RC2 and RC4.** Rivest encryption ciphers developed for RSA Data Security.

    - **RSA.** A public-key algorithm for both encryption and authentication. Developed by Rivest, Shamir, and Adleman.

    - **RSA key exchange.** A key-exchange algorithm for SSL based on the RSA algorithm.

    - **SHA-1.** Secure Hash Algorithm, a hash function used by the U.S. Government.

- **SKIPJACK.** A classified symmetric-key algorithm implemented in FORTEZZA-compliant hardware used by the U.S. Government. (For more information, see FORTEZZA Cipher Suites.)

- **Triple-DES.** DES applied three times.

- Key-exchange algorithms like KEA and RSA key exchange govern the way in which the server and client determine the symmetric keys they will both use during an SSL session.

- The most commonly used SSL cipher suites use RSA key exchange. The SSL 2.0 and SSL 3.0 protocols support overlapping sets of cipher suites. Administrators can enable or disable any of the supported cipher suites for both clients and servers.

- When a particular client and server exchange information during the SSL handshake, they identify the strongest enabled cipher suites they have in common and use those for the SSL session.

- Decisions about which cipher suites a particular organization decides to enable depend on trade-offs among the sensitivity of the data involved, the speed of the cipher, and the applicability of export rules.

SSL and the TCP/IP Protocol Suite

- All versions of SSL share the same basic approach: they provide a secure channel between two communicating programs over which arbitrary application data can be sent.

- In that sense an SSL connection acts very like a "secured" TCP connection. As the name Secure Sockets Layer implies, SSL connections are intended to act just like sockets connected by TCP connections.

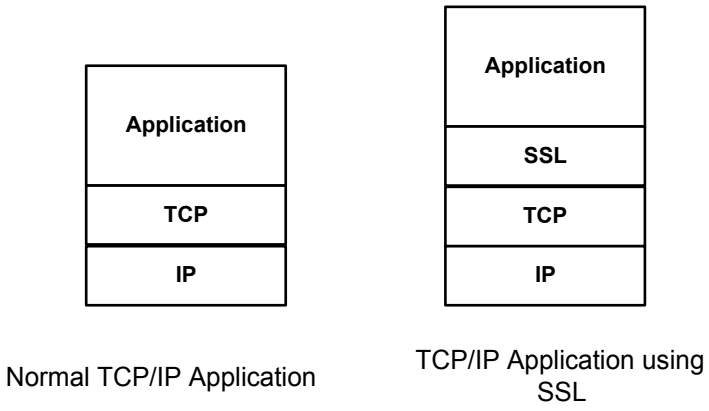
- Making the SSL semantics mimic TCP semantics makes the application programmer's life a lot easier.

- To that end most SSL implementations provide an API modeled after the most popular networking API, **Berkeley sockets**.

- The table shown provides an example of some calls from a typical SSL API (OpenSSL) and the corresponding Berkeley Sockets API calls.

Sockets API	OpenSSL
int socket (int, int, int)	SSL *SSL_new(SSL_CTX *)
int connect (int, const struct sockaddr *, int)	int SSL_connect(SSL *)
ssize_t write (int, const void *, size_t)	int SSL_write(SSL*, char *, int)
ssize_t read (int, void *, size_t)	int SSL_read(SSL *, char *, int)

- Ideally, an application programmer would be able to replace all sockets calls with SSL calls and get security more or less automatically.
- It would even be possible to secure an application merely by re-linking with a library that provided secure versions of the ordinary sockets call.
- Unfortunately, although this is possible to some extent, SSL semantics do not precisely match TCP semantics, which can lead to problems.
- The diagram shown illustrates how SSL fits within the TCP/IP protocol stack. It is just below the application layer and just above the TCP layer.



- As mentioned earlier, the primary use of SSL is to protect Web traffic using HTTP. 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 random characters (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 (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.
- After the initial exchange the server and client can start to communicate. The user is typically presented 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 most 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.

- Due to patent reasons, none of the free servers in the US 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 burden on the server machine than does pure HTTP, and obtaining the required certificate can be fairly inconvenient. Moreover, certificates aren't cheap.

- There are several commercial, well-known third party Certificate Authority (CA) organizations that will issue certificates. Businesses can use commercial CA's or use self-signed or private CA issued certificates.

Port Numbers for SSL Protocols

- 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.

- 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 SSL protocols use one of two basic strategies: **separate ports** and **upward negotiation**.

- In a **separate ports** strategy, the protocol designer simply assigns a different well-known port to the protocol and the server is designed to 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.

- In a **upward negotiation** strategy, the application protocol is modified to support a message indicating that one side would like to upgrade to SSL.

- If the other side agrees, an SSL handshake starts. Once the handshake is completed, application messages resume over the new SSL channel. SMTP over TLS uses this strategy.

- The table shown provides some of the currently used port assignments for applications using SSL protocols.

Service Name	Assigned Port/Protocol	Application
Ftps-data	989/TCP, UDP	ftp protocol, data, over TLS/SSL
Ftps	990/TCP, UDP	ftp protocol, control, over TLS/SSL
nntps	563/TCP, UDP	nntp protocol over TLS/SSL
imaps	993/TCP, UDP	IMAP4 over SSL
Pop3s	995/TCP, UDP	POP3 over SSL
https	443/TCP, UDP	HTTP over SSL
ldaps	636/TCP, UDP	LDAP over SSL

SSL Implementations

- The OpenSSL Project at http://www.openssl.org/ provides a high-quality, free, source implementation of SSLv2, SSLv3, and TLS available for free downloading.

- OpenSSL is based on Eric Young's SSLeay library, which was first released in 1995. The code is written in more or less ANSI C and has been widely ported.

- OpenSSL is licensed under a BSD-style license and is therefore free for commercial and noncommercial use.

- Until very recently, due to patent reasons, OpenSSL could not be used inside the United States in RSA mode. However, since RSA's patent expired in September 2000, it is now legal in the U.S. as well.

- The following vendors sell C/C++ SSL/TLS toolkits.

    - Certicom (http://www.certicom.com/)
    - Netscape Communications (http://home.netscape.com/)
    - RSA Security (http://www.rsasecurity.com/)
    - SPYRUS/Terisa Systems (http://www.spyrus.com/)

SSL Overview

- A connection is divided into two phases; the **handshake** and **data transfer** phases.
- The **handshake** phase **authenticates the server** and establishes the **cryptographic keys,** which are used to protect the data to be transmitted.
- The handshake must be completed before any application data can be transmitted. Once this has been done, the **data** is broken up and transmitted as a series of **protected records**.

The Handshake Phase

- An SSL handshake serves a threefold purpose. First, the client and the server need to agree on a set of algorithms that will be used to protect the data.
- Second, they need to establish a set of cryptographic keys that will be used by those algorithms.
- Third, the handshake may optionally authenticate the client.
- The overall process may be summarized a series of steps:
    - The client sends the server a list of the algorithms it is willing to support, along with a random number used as input to the key generation process.
    - The server chooses a cipher out of that list and sends it back along with a certificate containing the server's public key.

      The certificate also provides the server's identity for authentication purposes and the server Supplies a random number that is used as part of the key generation process.
    - The client verifies the server's certificate and extracts the server's public key.

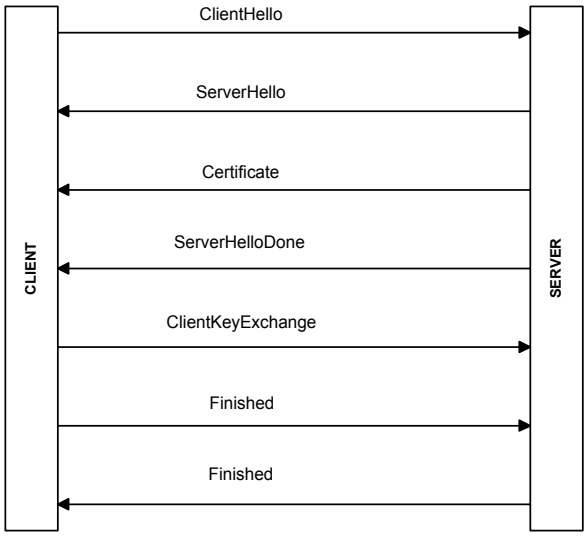
      The client then generates a random secret string called the **pre_master_secret** and encrypts it using the server's public key.

      It sends the encrypted public key to the server.
    - The client and server independently compute the encryption and **MAC** (Message Authentication Code) **keys** from the **pre_master_secret** and the client and server's random values.
    - The client sends a MAC of all the handshake messages to the server.
    - The server sends a MAC of all the handshake messages to the client.

- The first goal of the handshake was to agree on a set of algorithms and steps 1 and 2 accomplish this goal.
- The client gets an opportunity to tell the server which algorithms it supports and the server chooses an algorithm.
- When the client receives the message the server sent in step 2, it also knows the algorithm, so both sides now know what algorithm to use.
- The second goal was to establish a set of cryptographic keys. This is accomplished in steps 2 and 3.
- In step 2 the server provides the client with its certificate, which allows the client to transmit a secret to the server.
- Following step 3, the client and the server both share the **pre_master_secret**. The **client** has the pre_master_secret because it **generated** it, and the **server** has the pre_master_secret because it **decrypted** it.
- Note that step 3 is the key step in the whole handshake. All the data that is going to be protected depends on the security of the pre_master_secret.
- The client uses the server's public key (extracted from the certificate) to encrypt a shared key, and the server uses its private key to decrypt the shared key.
- The rest of the handshake is mainly devoted to ensuring that this exchange can happen safely.
- In step 4 the client and the server then separately use the same **key_derivation_function** (KDF) to generate the master-secret. The master-secret is used to generate the cryptographic keys, again using the KDF.
- Steps 5 and 6 serve to protect the handshake itself from tampering. Consider a scenario where an attacker wishes to control the algorithms used by the client and the server.
- It's quite common for the client to offer a range of algorithms, some weak and some strong, so that it can communicate with servers that only support weak algorithms.
- The attacker could delete all the strong algorithms from the client's offer in step 1 and thus force the server to choose a weak algorithm.
- The MAC exchange in steps 5 and 6 stops this, because the client's MAC will be computed over the original messages and the server's MAC will be computed over the messages the attacker modified, and they won't match when checked.
- Because the client- and server-provided random numbers are inputs to the key generation process, the handshake is secure from a replay attack.
- These messages are the first messages encrypted under the newly encrypted algorithms and keys.
- So, at the end of this process, the client and the server have agreed on the cryptographic algorithms to use and have a set of keys to use with those algorithms.

- More importantly, they can be sure that an attacker hasn't interfered with the handshake and that the negotiation reflects their authentic preferences.
- Each of the aforementioned steps is accomplished by one or more handshake messages.
- The correlation between the **steps** in a handshake phase and the **handshake messages** may be summarized as follows:
- **Step 1:**
    - Corresponds to a single SSL handshake message, **ClientHello**.
- **Step 2:**
    - Corresponds to a series of SSL handshake messages.
    - The first message the server sends is the **ServerHello**, which contains its algorithm preferences.
    - Next it sends its certificate in the **Certificate** message.
    - Finally, it sends the **ServerHelloDone** message, which indicates that this phase of the handshake is done.
    - The reason that the **ServerHelloDone** is needed is that some of the more complicated handshake variants involve other messages being sent after the **Certificate** message.
    - When the client receives the **ServerHelloDone** message, it knows that no such other messages will be arriving and so it can proceed with its part of the handshake.
- **Step 3:**
    - Corresponds to the **ClientKeyExchange** message.
- **Steps 5 and 6:**
    - Corresponds to the **Finished** message.
    - The **Finished** message is the first message that is protected using the just negotiated algorithms.
    - To protect the handshake from external tampering the content of the message is a MAC of all the previous handshake messages.
    - However, since the **Finished** message is protected using the negotiated algorithms, the message itself is also MAC'ed with the newly negotiated MAC keys.

- The message exchange is summarized in the diagram shown.

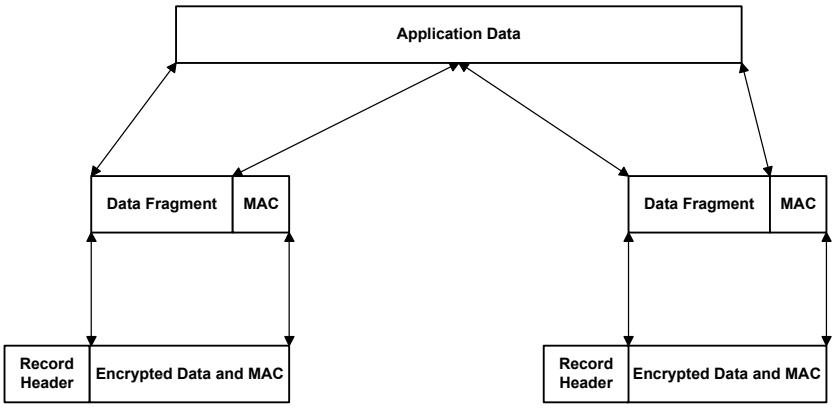


SSL Handshake Messages

The Data Transfer Phase

- Following a successful handshake, the client and server can now exchange encrypted and authenticated data.
- The actual data transfer is accomplished using the **SSL Record Protocol**.
- The SSL Record Protocol works by breaking up the data stream to be transmitted into a series of **fragments**, each of which is independently protected and transmitted.
- On the receiving end, each record may be independently decrypted and verified. This approach allows data to be transferred from one side of the connection to the other as soon ready and processed as soon as it is received.
- Each fragment is protected (encrypted) before it is transmitted.
- To provide integrity protection, a MAC is computed over the data. The MAC will be transmitted along with each fragment and subsequently verified upon receipt at the receiver.
- The MAC is then appended to the fragment and the concatenated data and MAC are encrypted to form the **encrypted payload**.
- Finally, a **header** is attached to the payload. The concatenated header and encrypted are referred to as a **record**. A **record** is what is actually **transmitted**.

- The next diagram illustrates the fragmentation and transmission process.



The Record Header

- The main function of the record header is to provide information that is necessary for the receiving implementation to interpret the record.

- In practice, this means three pieces of information: the **content type**, the **length**, and the **SSL version**.

- The **length** field allows the receiver to know how many more bytes to read off the wire before it can process the message.

- The **version** number is simply a redundant check to ensure that each side agrees on the version.

- The **content** type field identifies the type of the message. It allows implementations to distinguish management traffic from data that is intended for the higher-level application.

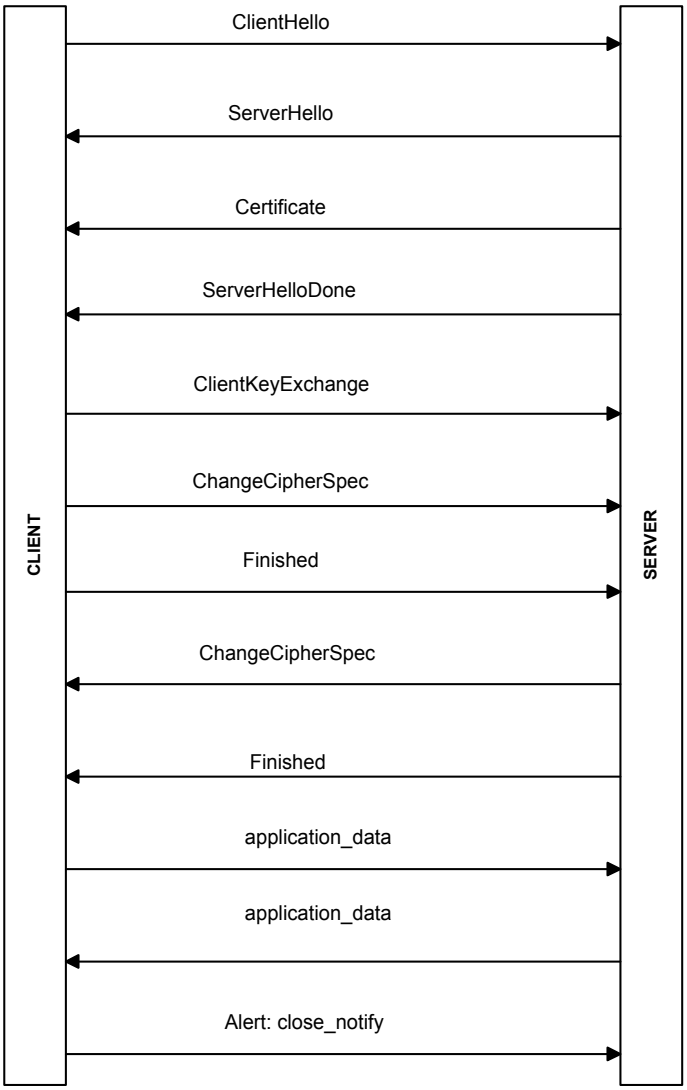
Content Types

- SSL supports four content types: **application_data**, **alert**, **handshake**, and **change_cipher_spec**.
- All data sent and received by software that uses SSL is sent as **application_data**.
- The other three content types are used for **management traffic**, such as performing the **handshake** and **signaling** errors.
- The **alert** content type is primarily used for signaling various types of error. Most alerts signal things that went wrong in the handshake, but some indicate errors that occurred when trying to decrypt or authenticate records.
- The other use of alert messages is to signal that the connection is about to close.
- The handshake content type is used to carry handshake messages.
- Since no cryptographic keys have been established, these initial messages aren't encrypted or authenticated, but otherwise the processing is the same.
- The **change_cipher_spec** message has a special purpose. It indicates a change in the encryption and authentication of records.
- Once a handshake has negotiated a new set of keys, the **change_cipher_spec** record is sent to indicate that those new keys will now be used.

Connection Summary

- The next diagram illustrates all the messages exchanged between a client and a server during a complete SSL session.
- The first message sent, **ClientHello**, contains the client's proposed cryptographic parameters, including the ciphers it's prepared to use.
- It also contains a random value to be used in key generation.
- The **server** responds with **three messages**:
- **First Server Message**
    - It sends the **ServerHello** which selects a cipher and a compression algorithm. This message contains a random value from the server.
    - The reason why SSL includes compression is that encrypted data is random for all practical purposes and thus incompressible.
    - Thus, **link-level compression** such as is found in most modems doesn't work on SSL data.
    - Therefore if we want to compress the data before transmission, it must be done **prior to encryption**.
    - Unfortunately, due to intellectual property considerations, neither SSLv3 nor TLS defined any compression algorithms. Thus, compression is almost never used with SSL.
    - As an exception, OpenSSL does support compression, so if both sides are based on OpenSSL and suitably configured, compression can be used.
- **Second Server Message**
    - The server sends the **Certificate** message, which contains the server's public key, in this case an RSA key.
- **Third Server Message**
    - Server sends the **ServerHelloDone**, which indicates that no more messages will be sent in this phase of the handshake.
- The client now sends a **ClientKeyExchange** message, which contains a randomly generated key encrypted under the server's RSA key.
- This is followed by a **ChangeCipherSpec** message from the client, which indicates that all messages that the client sends afterwards will be encrypted using the just-negotiated cipher.

- The client then sends a **Finished** message containing a check value for the entire connection so that the server can verify that the ciphers have been securely negotiated.
- Once the **server** has received the client's Finished message, it sends its own **ChangeCipherSpec** and **Finished** messages and the connection is ready to exchange application data.
- The SSLv3 specification allows one party to start sending application data as soon as that party's **Finished** message is received.
- However, this creates a security risk for client data if the connection has been compromised.
- If an attacker has changed the handshake to negotiate a weaker cipher suite, the client can't detect this until it receives the server's **Finished** message.
- To fix this, in TLS the **client** must **wait** until it has received the **server's Finished** message before it can send its first byte of data.
- Note that this isn't a security problem if the server chose the most secure of the client's ciphers, but rather than trying to detect this case, it's safer to simply wait for the other server's Finished message.
- The next two messages are **application data** sent by the client and server, respectively.
- Finally, the client shuts down the connection, but first it sends a **close_notify alert** to indicate that the connection is about to close.
- The **close_notify** is followed by a **TCP FIN** packet. The server responds with its own **TCP FIN** packet.
- Note that the server is not required to send a close notify.



Complete SSL Message Sequence

SSL Performance

- One of the most common criticisms made about SSL is that it is slow.
- Depending on the protocols being used, the server hardware, and the network environment, SSL connections can be anywhere between 2 and 100 times slower than ordinary TCP connections.
- In a critical business environment this performance cost translates directly into increased operational costs.
- Computationally intensive security protocols necessarily mean that more server architecture must be scaled upward to maintain an acceptable level of quality of service.
- Usually this means moving from a single-server architecture to a multiple-server architecture. This entails arranging a data-sharing mechanism between servers.
- Given that cryptography is computationally expensive, performance degradation is unavoidable.
- However it is possible to minimize SSL's performance impact with careful systems design and development.

Amdahl's Law

- This is a relationship put forward by Gene Amdahl in 1967 and is applied in research involving massively-parallel processing.
- Generally speaking is states that even when the fraction of serial work in a given problem is small, say ***s***, the maximum speedup obtainable from even an infinite number of parallel processors is only $\mathbf{1/s}$.
- If ***N*** is the number of processors, ***s*** is the amount of time spent (by a serial processor) on serial parts of a program and ***p*** is the amount of time spent (by a serial processor) on parts of the program that can be done in parallel, then Amdahl's law says that speedup is given by:

$$Speedup = \frac{(s+p)}{s+\frac{p}{N}} = \frac{1}{\left(s+\frac{p}{N}\right)}$$

- Where total time $(s + p) = 1$ for algebraic simplicity.

- For more details see “Amdahl, G.M. Validity of the single-processor approach to achieving large scale computing capabilities”. **AFIPS Conference Proceedings** vol. 30 (Atlantic City, N.J., Apr. 18-20). AFIPS Press, Reston, Va., 1967, pp. 483-485.

- Many researchers ([Hennessey 1996]) have applied this law to derive the most basic rule of performance tuning and optimization of software applications.

- Roughly speaking, the speedup resulting from an optimization is equal to the improvement of the speedup multiplied by the fraction of the CPU time spent in the optimized code:

$$Speedup_{overall} = \frac{Execution\_Time_{old}}{Execution\_Time_{new}} = \frac{1}{\left(1 - Fraction_{optimized}\right) + \left(\frac{Fraction_{optimized}}{Speedup_{optimized}}\right)}$$

- Therefore, it is of crucial importance to determine which sections of the system consume the most time, because they represent the prime targets for optimization.

- The objective then is to try and improve the performance of these sections of the system, either by making them run faster or execute that code less often.

- As a corollary of the above, even very expensive operations aren't worth speeding up unless they are executed frequently.

- In the case of a network protocol, this means that we are primarily concerned with operations that happen with every transaction or protocol message.

- Time consumed by system startup and shutdown can be safely ignored because it is insignificant over the life of a server.

The 90/10 Rule

- The applicability of Amdahl's law is not of much use if all parts of a system consume equal amounts of time.

- This is due to the fact we would have to optimize a significant fraction of the system in order to see any improvement.

- Fortunately most of a system's time is spent performing a few tasks. A useful rule of thumb, the **90/10 rule** states that 90% of the execution time of a program is spent in 10% of the code.

- Combining the 90/10 rule with Amdahl's law, we can see that optimizing that 10% section of the system would significantly enhance the performance of the system.

I/O Overhead

- Systems that perform extensive Input/Output in general, and network protocols in particular, add another dimension to code optimizing.

- Consider the problem of transferring a file over a network between two hosts. On a modem link the factor that limits the performance of this transaction isn't the load placed on the CPU by the transmitting and receiving programs but rather the performance of the network between the two machines.

- Provided that the machines at either end meet some minimal standard of processing power the speed of transmission is constant because it's limited by the speed of the communications channel.

- In this case, then, it might be worth expending some CPU power to compress the data before (or during) transmission in order to reduce transmission time.

- Typical text files can be compressed by a factor of between two and four, so, compressing while encrypting will reduce the transmission time and raise the effective data transmission rate.

- The bottleneck is still the network: The machine can easily compress data faster than it can be transmitted over the network. In fact, compression is so much faster that nearly all commercial modems automatically compress the data as they transmit it.

- Now, consider the same two machines connected by Fast Ethernet. Such a network can transmit roughly 100 Mbps.

- In this case, then, data can be transmitted much faster than it can be compressed. The bottleneck is the CPU rather than the network.

- Thus depending on the type of network, it is possible to trade off I/O bandwidth for CPU and vice versa.

Latency versus Throughput

"Never under estimate the bandwidth of a station wagon full of tapes."

- **Latency** is the amount of time it takes to process a given transaction from beginning to end.
- **Throughput** is the number of total transactions you can maintain over a period of time.
- The following table illustrates some latency versus throughput combinations.

	Low Latency	**High Latency**
Low Throughput	Modems	Pager
High Throughput	LANs	Satellite

Servers versus Clients

- As a general rule, network systems have many clients and a few servers.
- In such systems, slowdown at the client side generally manifests as increased latency, whereas slowdown at the server side manifests as both increased latency and decreased throughput.
- Consider an example of a single network file server that services a large number of clients equally. The server has a T1 Internet connection (capable of carrying roughly 1.5 Mb/s) and each client is served by a 56 Kbps modem.
- The server can service approximately 50 clients at a time and keep their network connections fully utilized. A 1 MByte file will take roughly 2.5 minutes to get completely transferred.
- Now suppose that the number of clients is doubled. The server tries to service them all equally, so they now get only half as much data as before.
- This doubles the latency to about 5 minutes. Notice, however, that although it takes twice as long to serve each client, twice as many clients are being serviced at the same time, so throughput is unaffected.
- Now consider the impact of halving the speed of the server's Internet connection. If we keep the number of clients the same, then we can serve them at only half the previous rate, doubling latency and halving throughput.
- This sort of effect applies equally when the bottleneck is CPU instead of I/O. Slow clients leave server resources available for other clients. Slow servers simply slow down the system as a whole.

- Note that CPU and bandwidth aren't the only scarce resources for a server. Each client consumes some server resources, especially memory, so when the number of clients gets really large, the server can be overloaded just from trying to serve them all, even if there is plenty of CPU and channel bandwidth to go around (think DoS attacks).

Cryptography Overhead

- As mentioned earlier, the primary reason why SSL is slow is that cryptography, particularly the big number operations required for public key cryptography, is extremely CPU intensive.
- The vast majority of the time on both client and server is spent on cryptographic processing. In fact, the vast majority of time is spent on a single operation: RSA decryption.
- The two phases of an SSL connection, handshake and data transfer, are worth considering separately.
- Handshake happens only once per connection, but is comparatively expensive.
- Each individual data record is comparatively cheap, but for connections involving large amounts of data the cost of the data transfer phase eventually dominates the cost of the handshake.
- Nevertheless, the cost of data transfer is mostly due to cryptography.
- The costs of the handshake differ substantially between client and server, but the data transfer phase is symmetrical.

Server Handshake

- The performance profile for the server side of the handshake is comparatively simple.
- Over half of the CPU time goes to a single operation: the RSA private decryption of the **pre_master_Secret**.
- This is followed by the computation of the master-secret and the key schedules from the pre-master secret.

Client Handshake

- The client side performance is far less straightforward. The client side operations are much faster than the server side operations.
- Thus, in many cases the client spends most of its time simply waiting for server messages.
- Once this waiting time is factored out, it can be shown that the two most expensive operations are verifying the server's certificates and encrypting the **pre_master_secret**.
- Although both of these involve RSA operations, these are public key operations and therefore are roughly an order of magnitude faster than the private key operations the server executes.
- As a consequence, the client's major computational burdens are twofold: RSA computations and the ASN.1 decoding of the server's certificate.

The Handshake Bottleneck

- In this case, the server's performance limitations control the performance of the system as a whole.
- This situation would change somewhat if client authentication were used. In that case, the client would perform an RSA private key operation more or less equivalent to that performed by the server and this would become the dominant performance cost on the client.
- The server would incur almost no additional performance cost.
- It should be noted that this particular division of labor really only applies to RSA. DSA operations are much more symmetrical between client and server.
- Research data has shown that when DSS/DHE cipher suites are used, the computational load is similarly symmetrical between client and server.
- However, in most production systems, the server is still the bottleneck. This is mainly due to the fact that client authentication and DSS/DH are rarely used.
- In addition, as discussed earlier, a server is accessed by many clients, whereas a client typically will have only a small number of connections open.
- Thus, even a very fast server will quickly achieve a load that slows it down below the speed of clients.

Data Transfer

- In the data transfer phase, there are two relevant cryptographic operations, record encryption and the record MAC. These two operations account for the majority of the cost of data transfer.
- The relative importance of these depends on which precise algorithms are chosen and the implementation details.
- If a fast encryption algorithm (such as RC4) is chosen, then the HMAC will be the dominant cost. If a slow encryption algorithm such as 3DES is selected, then encryption and MAC process will share the cost somewhat evenly.
- Record assembly and disassembly account for an insignificant amount of time.

Handshake Algorithms and Keys

- The performance characteristics of an SSL connection depend substantially on the algorithms being used.
- In the handshake phase, this can dramatically affect the cost of the handshake, as well as changing the relative loads on the client and server.

RSA versus DSA

- When deploying SSL systems, the choice of signature algorithm often depends on intellectual property or compatibility concerns.
- However, in cases where performance is the dominant concern, RSA is the algorithm of choice over DSS/DH.
- In general, RSA is much faster than DSA for verification and comparable for signature.
- This cost however is incurred primarily by the client, so it affects latency, but not the throughput.
- The DSA cipher suites are nearly always run in ephemeral mode. This requires substantial extra work on both sides to generate, sign, and verify the ephemeral DH keys.
- The end result is a dramatic reduction in throughput. In practice, DSA cipher suites tend to be between two and ten times slower than RSA cipher suites of comparable strength.

Fast or Strong Encryption

- The performance of public key algorithms declines dramatically with key size.
- RSA with 1024 bits is roughly four times slower than RSA with 512 bits.
- DSA performance is similarly reduced by longer keys. As a consequence, choosing your private key length is a compromise between security and handshake performance.
- Generally speaking, 512 bits is almost certainly too short for valuable data. However, 768-bit keys are probably strong enough for most commercial transactions and are dramatically faster than 1024-bit keys.

Ephemeral RSA

- When using RSA with a strong (>512) bit key and an export cipher, the standard requires applications to use a 512-bit ephemeral RSA key.
- In essence, this adds a single 512-bit RSA operation to each side (a public one on the client side and a private one on the server side). This has some negative effect on throughput and latency.
- Note that both sides must still perform all the operations they would have performed if ephemeral RSA were not being used.
- The slower these operations, the smaller the proportional effect will be of using ephemeral RSA.
- For 1024-bit keys, expect to see between a 10% and 25% slowdown. Note that this is not a security/performance tradeoff, as 512-bit ephemeral RSA is both weaker and slower than 1024-bit static RSA.
- 512-bit keys are much weaker than 1024-bit keys, and because the same 512-bit key is used for a large number of transactions, there is no compensating benefit of perfect forward secrecy.

Bulk Data Transfer

Algorithm Choice

- Recall that cryptography consumes the bulk of the time in record processing. Therefore the choice of MAC and encryption algorithms affects the performance of the system.
- Use RC4 for maximum speed and 3DES for maximum security.
- RC4 is about ten times faster than 3DES.
- Digest choice is less flexible as MD5 is being phased out.
- SHA-1 is fast enough in most real world scenarios, so it's a generally good choice. On modern systems, RC4 with SHA-1 is fast enough to saturate most networks.

Optimal Record Size

- A significant amount of the record computation is insensitive to the size of the data being transmitted.
- There is substantial fixed overhead, both from the cryptographic algorithms and from writing to the network.
- As a consequence, transmitting the data in small records results in bad performance. Up to a certain point, increasing the size of the record increases performance.
- On one test system, this point appears to be roughly 1024 bytes. Beyond this point, the advantage of using even larger blocks is small.
- Thus it pays to buffer it until you can encrypt fairly large records.

Some Basic SSL Performance Rules

- The general idea behind getting good performance is to perform as few operations as possible and necessary.
- This means minimizing the number of time-consuming operations and, whenever possible, optimizing those sections of code.

- **Asymmetric algorithm choice.**
    - Use RSA.
- **Private key size.**
    - Use the shortest private keys with which you feel safe. 768 is good enough for most applications.
- **Symmetric algorithm choice**
    - Use RC4 for best performance, 3DES for best security. Export versions compromise security without improving performance.
- **Digest algorithm choice**
    - MD5 only improves performance about 40% over SHA-1. In most cases SHA- I will be fast enough. Stick with SHA- I for security.
- **Record size**
    - Send data in the largest chunks possible.

Programming with SSL

- As mentioned earlier, we will be using the API available with the OpenSSL toolkit.
- The TCP client/server examples presented earlier have been modified to use the SSL calls already described.

Context Initialization

- The first task both programs will perform is to set up a context that will be used by the system.
- Most implementations do this by having a context object, which the program needs to initialize.
- This context object is then used to create a new connection object for each new SSL connection.
- These connection objects are then used to perform SSL handshakes, reads, and writes.
- This approach has two advantages. First, the context object allows many structures to be initialized only once, saving time.
- Rather than reloading this material for every connection, we simply load it into the context object at program startup. When we wish to create a new connection, we can simply point that connection to the context object.
- The second advantage of having a single context object is that it allows multiple SSL connections to share data.

- The initializing for both the client and the server program is done in the **initialize_ctx()** function.
- Before we can even create an SSL context we need to initialize the OpenSSL library itself using **SSL_library_init()**.
- OpenSSL calls its SSL context variable an **SSL_CTX** (for SSL context). The **meth** argument allows us to set the SSL version that this context is prepared to negotiate.
- This technique allows us to link in only those sections of code that are required. Each method references the various functions and object files needed to implement it.

Client Initialization

- In general, a client will have to perform the following initialization tasks.

- **Load CAs**
    - The client needs to have a list of the CAs that it trusts to sign server certificates. Typically this list is stored in a file on disk somewhere and the client needs to load it.

- **Load client keying material**
    - If the client has keys that it is using for client authentication, it will need to load them and their associated certificates as well.
    - In our case these are provided in a file.

- **Seed random number generator**
    - This can be any source of random seed data. For instance, screen or mouse data is often a good source of randomness.
    - In our examples the system messages file is used as a source of random data and fed to the to the toolkit.

- **Set allowable cipher suites**
    - Most toolkits have some set of cipher suites that they are prepared to negotiate by default.
    - The exact number of operations required to perform these actions depends on the particular toolkit you're using.
    - For instance, SPYRUS's TLSGold loads all of its keying material from a single file in one operation whereas OpenSSL requires a number of different loading operations.

Server Initialization

- Server initialization is a superset of client initialization.
- Although it's technically possible for a server that does not require client authentication not to have a list of CAs it trusts, it's good practice to check one's own certificates when initializing.
- This protects against data corruption and user error. Thus, loading a CA list is a good idea for servers as well.

- **Set DH group**
    - In ephemeral DH mode the server needs to be loaded with the DH group to use.
    - Since DH group generation is very slow it is desirable to do this at startup time.

- **Set ephemeral RSA keys**
    - Ephemeral RSA is a requirement when using long RSA keys and supporting export cipher suites.
    - Toolkits will generate one automatically as needed, but loading it as part of the initializing avoids incurring a performance cost later.

Client Connect

- Once the client has initialized the SSL context, it's ready to connect to the server.
- SSL toolkit APIs closely resemble the Berkeley Sockets API with a series of SSL calls analogous to sockets calls.
- OpenSSL requires us to create a TCP connection between client and server in the usual way and then use the TCP socket to create (and connect) an SSL socket.
- Separating the TCP connection from the SSL connection has both advantages and disadvantages.
- The primary disadvantage is that it adds to the programming complexity.
- Also, separating the TCP connect and the SSL handshake means that the SSL API doesn't know the DNS name or port that the server is expected to have.
- This means that such a toolkit cannot automatically check the server's certificate against the DNS name. The application code must perform this check.

- The main advantage is that by requiring the application to do the TCP connect provides flexibility.

- Because an upward negotiation strategy requires that the application protocol have access to the socket in order to negotiate the transition to SSL, this sort of API is required if you want to do upward negotiation.

Client SSL Handshake

- The first thing the client application does is to construct the SSL_CTX object usingi n **initialize_ctx()**. No further initialization is required on the client.

- Once the TCP connection to the server is established, we need to turn on SSL. OpenSSL uses the SSL object to represent an SSL connection.

- The most important thing to notice about this stage is that the SSL object is not directly attached to the socket.

- First, we create a **BIO object** using the socket and then attach the SSL object to the BIO.

- OpenSSL uses **B IO objects** to provide a **layer of abstraction** for I/O. As long as the object meets the BIO interface, it doesn't matter what the underlying I/O device is.

- This abstraction layer provides even more flexibility than simply separating the TCP connect and the SSL handshake: It is entirely possible with OpenSSL to do SSL handshakes on devices that aren't sockets at all provided that we have an appropriate BIO object.

- A practical use of this would be to support some protocol that can't be accessed via sockets. For instance, we could run SSL over a serial line.

- Once the SSL connection is established, the certificate chain is checked using the function **check_cert_chain()** as shown.

- Note that it does not check the certificate chain length here. Instead, the length check is done automatically by OpenSSL as long as we set the maximum chain length using **SSL_CTX_set_verify_depth()**.

Server Accept

- The code for the server side of the SSL connection is quite similar to that of the client side.

- As with the C client accept, OpenSSL requires the application to perform the TCP accept the normal way.

- With the TCP socket created, the **BIO** and **SSL** objects are created exactly as before.

Simple I/O Handling

- Once the SSL connection is established, the two sides start exchanging data over it. In principle this is very similar to sending data over TCP sockets.
- In practice however, SSL introduces some subtleties that need to be kept in mind. The examples provided illustrate the simplest way of doing this.
- This can be extended to more complex implementations using threads and multiplexed I/O.
- The simplest example of SSL I/O is on the server side. The server's task is simply to read data from the client in a loop and echo it back.
- If there's no data available to write to the client, do nothing. This is done using the standard TCP read/write loop with calls to **SSL_read()** and **SSL_write()** replacing calls to **read()** and **write()**.
- The read part is accomplished by allocating a buffer of appropriate size and calling **SSL_read()**.
- Note that buffer size is not really that important here. The semantics of SSL_read are that it returns the data that is available, even if it's less than the requested amount.
- On the other hand, if no data is available, then the call to read blocks until some data is available in the socket.
- The choice of **BUFSIZE**, then, is basically a performance tradeoff. The tradeoff is quite different here than when reading from normal sockets.
- In that case of normal sockets, each call to **read()** requires a context switch into the kernel. As you are aware, context switches are expensive and most programmers try to use large buffers to reduce them.
- However, when we're using SSL the number of calls to **read()** - and hence context switches - is largely determined by the number of records the data was written in rather than the number of calls to **SSL_read()**.
- For example, if the client wrote a 1000-byte record and we call **SSL_read()** in chunks of 1 byte, then the first call to SSL_read will result in the whole record being read in and the rest of the calls will just read it out of the SSL buffer.
- Thus, the choice of buffer size is less significant when we're using SSL than with normal sockets.
- Also note that if the data is written in a series of small records, you may want to read all of them at once with a single call to read.
- OpenSSL provides a flag **SSL_CTRL_SET_READ_AHEAD** that turns on this behavior.
- Instead of **errno**, OpenSSL provides the **SSL_get_error()** call. This call lets us examine the return value and figure out whether an error occurred and what it was.

- The data just read is echoed back using **SSL_write()**. By default (as per OpenSSL) the call to **SSL_write()** should always write all the data before returning.

Closing the connection

- We call **SSL_shutdown()** to send the **close_notify** and then free the SSL structure and close the socket.

- The function **destroy_ctx()** is used to free the context object.