Efficiently handling SSL transactions is one cornerstone of your IT security infrastructure. Do you know how the protocol actually works?

Wesley Chou

Inside SSL: The Secure Sockets Layer Protocol

As enterprises conduct more and more of their business activities online, the need for security becomes more crucial. Organizations must implement protocols to address a variety of security-related tasks, including

- protecting sensitive data,
- confirming data integrity,
- authenticating senders, and
- safeguarding Web sites against unauthorized access and attacks.

Which of these issues a particular security protocol addresses—and how well—dictates the role it should play in an IT infrastructure. For example, a secure virtual private network (VPN) would require a protocol that provides strict access control and data protection. On the other hand, an e-business would want to allow easy access to its site but still provide data security and authentication for e-commerce transactions. For this level of need, one security protocol—SSL, the secure sockets layer—has been widely implemented and is now the de facto standard for providing secure e-commerce transactions over the Web.

Inside

THE BIRTH OF SSL

When Netscape developed its first Web browser, it realized that secure connections between clients and servers would be essential for the Internet's success as a business tool. Because there was no way to guarantee the security of the network through which Web traffic passed, the best way to protect data was to provide encryption and decryption at a connection's endpoints.

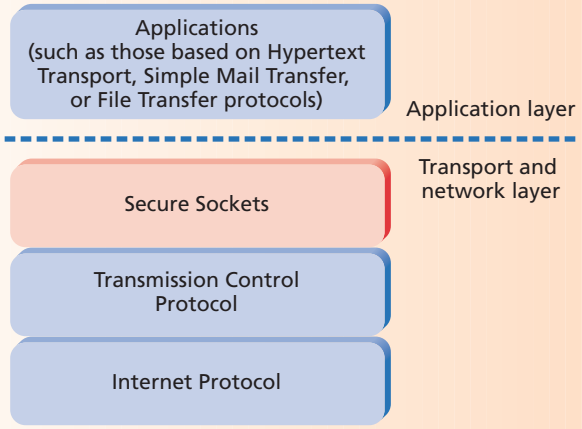
Netscape could have incorporated this encryption directly into its browser application, but this would not have provided a unified solution that non-HTTP applications could also use. A generic, application-independent form of security was necessary. Consequently, Netscape developed SSL to sit on top of TCP (Transmission Control Protocol), thus providing a TCP-like interface to upper-layer applications, as Figure 1 shows.

In theory, application developers could take advantage of the new layer by replacing all traditional TCP socket calls with the new SSL calls. The specific details of how SSL encrypted and decrypted the data were relatively transparent. As other vendors—most notably, Microsoft—began developing their own transport security protocols, the Internet Engineering Task Force (IETF) intervened to define a standard for an encryption-layer protocol. With the input of multiple vendors, the IETF created TLS, the Transport Layer Security standard. The "SSL and TLS Resources" sidebar lists Web sites that detail these protocols.

The first version of TLS is based on the last version of SSL, 3.0—in fact, TLS 1.0 is often called

1520-9202/02/$17.00 © 2002 IEEE

Figure 1. SSL operates above the transport layer to provide application-independent security.



SSL and TLS Resources

- **SSL specification,** http://www.netscape.com/eng/ssl3/ssl-toc.html
- **Request for comments on TLS,** http://www.ietf.org/rfc/rfc2246.txt
- **Open-source tool kit for implementing SSL/TLS,** http://www.openssl.org

Encryption Algorithms Online

- **AES,** http://csrc.nist.gov/publications/fips/fips197/fips-197.pdf
- **DES,** http://csrc.nist.gov/publications/fips/fips463/ fips46-3.pdf
- **Diffie-Hellman,** http://www.ietf.org/rfc/rfc2631.txt
- **RSA,** http://www.rsasecurity.com

SSL 3.1. Although TLS and SSL have subtle implementation differences, application developers usually notice very little difference, and end users should see no difference at all. TLS 1.0 and SSL 3.0 are not, however, interoperable. The most significant difference is that TLS requires certain encryption algorithms that SSL does not. A TLS server must "back down" to SSL 3.0 to interoperate with SSL-3.0 clients.

CRYPTOGRAPHY 101

Before I go into the details of SSL's operation, a brief cryptography primer will be useful to define terms and clarify relevant principles.

In general, encryption involves using a mathematical algorithm that takes two inputs—the plain data stream (*plaintext*) and a prespecified number (*key*)—and as output produces an encrypted data stream (*ciphertext*). Decryption involves using an algorithm that takes the ciphertext and a key as inputs and provides the plaintext as output.

The key and the algorithm used for encryption aren't necessarily the same key and algorithm used for decryption. Without the decryption key, the ciphertext stream should be difficult to decipher. For the best encryption algorithms, the most efficient way to break the code is to perform a brute-force search, attempting every possibly key combination. If the key is sufficiently large, this search is essentially impossible for an ordinary individual.

Two distinct types of encryption algorithms exist: *secret key* (also known as private or symmetric key) and *public key,* sometimes called asymmetric key. The "Encryption Algorithms Online" sidebar lists addresses for information about several of the most widely used algorithms.

Secret-key encryption

With secret-key algorithms, both the sender and the recipient use the same key to encrypt and decrypt the data, as Figure 2 shows. Examples of popular secret-key encryption algorithms include DES (Data Encryption Standard), Triple DES, AES (Advanced Encryption Standard), and RC4. An inherent problem with secret-key encryption algorithms is how to reliably and securely distribute the keys. It is a classic catch-22: Both the sender and receiver must have keys before a secure data transfer can occur, yet key distribution must occur over a secure data channel. If the key distribution is unsecure, anyone listening on the line can learn the key and decrypt successive messages.

Because secret-key algorithms depend on secure key transfer, the actual mathematical operations the protocols use to encrypt and decrypt the data are somewhat straightforward computationally. DES, for example, mainly involves the use of basic arithmetic, direct table lookups, and rearrangements of bit locations.

Public-key encryption

Public-key encryption involves two separate keys: public and private, with each public key corresponding to a specific private key. Normally, the sender uses the public key to encrypt the message, and the recipient uses the private key to decrypt it, as Figure 3 shows. Knowledge of the public key does not allow any unintended recipients or eavesdroppers to decrypt the message. Consequently, a private-key holder can freely publish the associated public key. This way, anyone can use the public key to encrypt

Figure 2. Secret-key encryption.

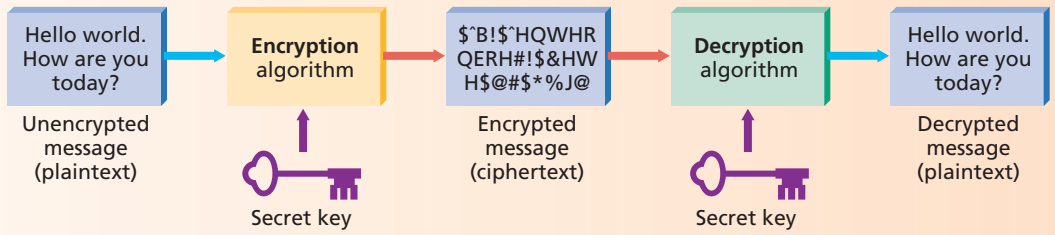
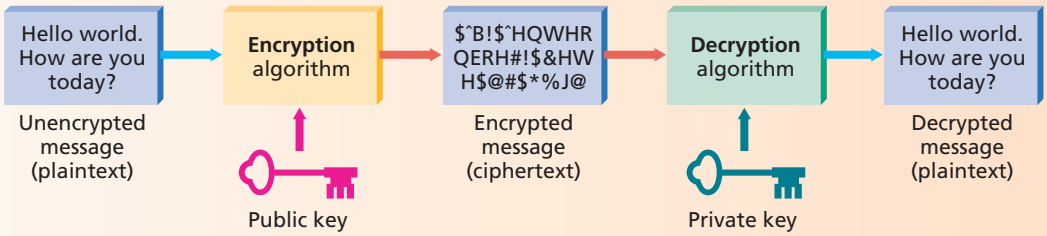


Figure 3. Public-key encryption.



a message and send it to the private-key owner. Only the holder of the private key can decrypt the message.

Unfortunately, decrypting a message that was encrypted with a public-key algorithm is quite CPU intensive. For example, one of the most popular public-key encryption algorithms, RSA (Rivest-Shamir-Adelman), defines keys that it then uses as exponents and modulus during cryptographic computations. When used as an exponent, even a relatively small, 512-bit key requires a significant number of CPU instructions.

Another interesting feature of public-key encryption is that a sender can use the private key to encrypt a message that the receiver can only decrypt with the associated public key. This characteristic is useful for message authentication. For example, a bank can create a *digital signature* by encrypting some short message with its private key. Any bank customer can then use the bank's public key to decrypt the message and verify that the message did, in fact, come from the bank.

Private, public, or both?

Secret-key encryption algorithms often involve a deterministic number of additions and shifts. These algorithms frequently use the key to assist in bit manipulation or to make the data in the ciphertext stream appear more random. In other words, while an increase in the secret key's size might increase the ciphertext's randomness, it does not necessarily increase the number of instructions required.

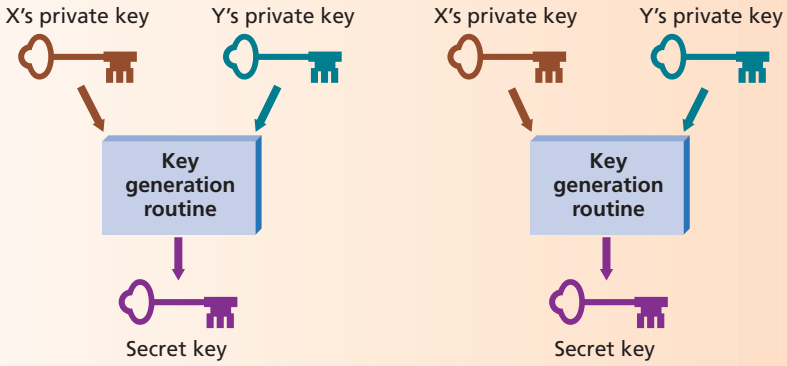
Public-key encryption, however, often uses the key as an exponent, so a large key can have a direct effect on the number of calculations required to encrypt or decrypt a data block. Thus, although public-key algorithms do not face the key distribution difficulty that secret-key algorithms do, they require significantly more computing power.

To capitalize on the strengths of both types of algorithm, security protocols often use public-key algorithms to transmit secret keys. In other words, a data transfer encrypted with a public-key algorithm will contain a secret key as a payload. Once distributed, secret keys are the basis for successive data transfers.

One slight variant of this approach uses a public-key *agreement algorithm* rather than a *distribution algorithm*. With a public-key agreement algorithm, one entity does not actually transmit data about the secret key to the other. Instead, the two entities exchange public keys and then independently generate a secret key. One of the most popular forms of public-key agreement is the Diffie-Hellman algorithm, illustrated in Figure 4.

Although SSL supports the Diffie-Hellman protocol, the majority of SSL transactions do not use this public-key agreement approach. Instead, they use the RSA public-key algorithm to distribute secret-key parameters. In contrast, implementations of IPSec—the protocol suite designed to add security directly to the IP layer—generally do use the Diffie-Hellman protocol as a public-key agreement algorithm with which to determine secret-key parameters.

Figure 4. Diffie-Hellman public-key agreement algorithm.



Major Certificate Authorities

- Entrust, http://www.entrust.com
- GeoTrust, http://www.geotrust.com
- Thawte, http://www.thawte.com
- Verisign, http://www.verisign.com

HOW SSL SECURES A TRANSACTION

SSL consists of two phases: handshake and data transfer. During the handshake phase, the client and server use a public-key encryption algorithm to determine secret-key parameters. During the data transfer phase, both sides use the secret key to encrypt and decrypt successive data transmissions.

The client initiates an SSL handshake connection by first transmitting a Hello message. This message contains a list of the secret-key algorithms, called *cipher specs*, that the client supports. The server responds with a similar Hello message, selecting its preferred cipher spec. Following the Hello message, the server sends a *certificate* that contains its public key.

Essentially, a certificate is a set of data that validates the server's identity. It contains the server's ID, its public key, and several other parameters. A trusted third party, known as a *certificate authority* (CA), generates the certificate and verifies its authenticity. To obtain a certificate, a server must use secure channels to send its public key to a CA. The CA generates the certificate, which contains its own ID, the server's ID, the server's public key, and other information. The CA then uses a message digest algorithm to create a certificate *fingerprint*. Conceptually, a message digest is similar to a checksum in that it takes a given stream of data and produces a deterministic, fixed-length output (the fingerprint). The CA then encrypts the fingerprint with its private key to create the certificate signature.

To validate a server's certificate, a client first uses the CA's public key to decipher the signature and read the precalculated fingerprint. Then the client independently computes the certificate's fingerprint. If the two fingerprints don't match, the certificate has been tampered with.

Of course, to decipher the signature, the client must have previously and reliably obtained the CA's public key. The client maintains a list of trusted CAs and their public keys. When the client receives a server's certificate, it verifies that the CA signing the certificate belongs to its list of trusted CAs. Fortunately, there are relatively few CAs, so they can easily publish their public keys in hard copy or broadcast them to many public Web sites (see the "Major Certificate Authorities" sidebar). In fact, many browser applications have the public keys of major CAs compiled directly into their code.

Once the client has authenticated the server (the server can also request a certificate from the client), the two use a public-key algorithm to determine secret-key information. After each side has indicated its readiness to begin using the secret key, the two sides complete the handshake phase with Finished messages, and the connection enters the data transfer phase.

During the data transfer phase, both sides break up their outgoing messages into fragments and append to them *message authentication codes* (MACs). The MAC is an encrypted message digest (fingerprint) computed from the message contents. For security purposes, the MAC key is different from the secret key; it is also computed during the handshake phase. When transmitting, the client or server combines the data fragment, MAC, and a record header and encrypts them with the secret key to produce the completed SSL packet. When receiving, the client or server decrypts the packet, computes the MAC, and compares the computed MAC to the received MAC.

Figure 5 shows an example of the SSL handshake, using the RSA public-key algorithm for key exchange.

Another SSL feature is the concept of *session resumption*. SSL's original designers knew that public-key algorithms are computationally expensive. A client making several new connections with the same server in a short time would heavily burden the server and thus experience a noticeable response time delay. However, if the client and server establish a unique session ID at the beginning of a new session, any successive connections within a given time frame can simply refer to that ID and use the same secret key, as illustrated in Figure 6.

Of course, there is an inherent security risk in reusing a session ID for any length of time. Consequently, if desired, the client can renegotiate a new session ID and a new secret key for a given session. Microsoft's Internet Explorer, for example, renegotiates a new session ID every two minutes.

Although this renegotiation feature provides extra security, it renders content switching of SSL connections impossible in configurations that don't include a specific SSL accelerator as part of the content switch. (I'll go into the details of SSL accelerators on content switches in Part 2 of this article.)

SSL TASK OUTLINE

To summarize, the tasks that each SSL entity—client or server—must be able to perform fall into three categories: handshake processing, bulk-data operations, and administration. Of these task categories, handshake processing takes the most computing power, followed by bulk-data encryption and decryption, and then administration.

Handshake processing

Handshake processing breaks down into several distinct categories.

Message exchange. In message exchange, both the client and the server must recognize the defined messages, such as Hello and Finished.

Public-key computations. The client must use public-key algorithms to either distribute or generate secret-key parameters. In addition, the client must use a CA's public key to validate a server's certificate. On the other end, the server must use public-key algorithms to either decrypt or generate secret-key parameters. If the server requires client authentication, it must also use a CA's public key to validate the client's certificate.

Random-number generation. De-pending on the type of public-key encryption algorithm used, both the client and server might need to generate random numbers to reduce the predictability of secret keys.

Handshake authentication. To provide handshake authentication, both the client and server generate a message digest based on, among other inputs, all messages transmitted during the handshake phase. Each side will then encrypt and send this message digest as part of the Finished message. The recipient of the Finished message must verify the message digest's authenticity by performing an independent message digest calculation that includes all previously received handshake messages. If the two message digests do not match, one or more of the handshake messages has been tampered with.

Bulk-data operations

Bulk-data operations are performed in actual data that must be securely transferred between the client and server.

Encryption/decryption. During the data transfer phase, both the client and server use the secret key to encrypt and decrypt data.

Figure 5. SSL handshake example.

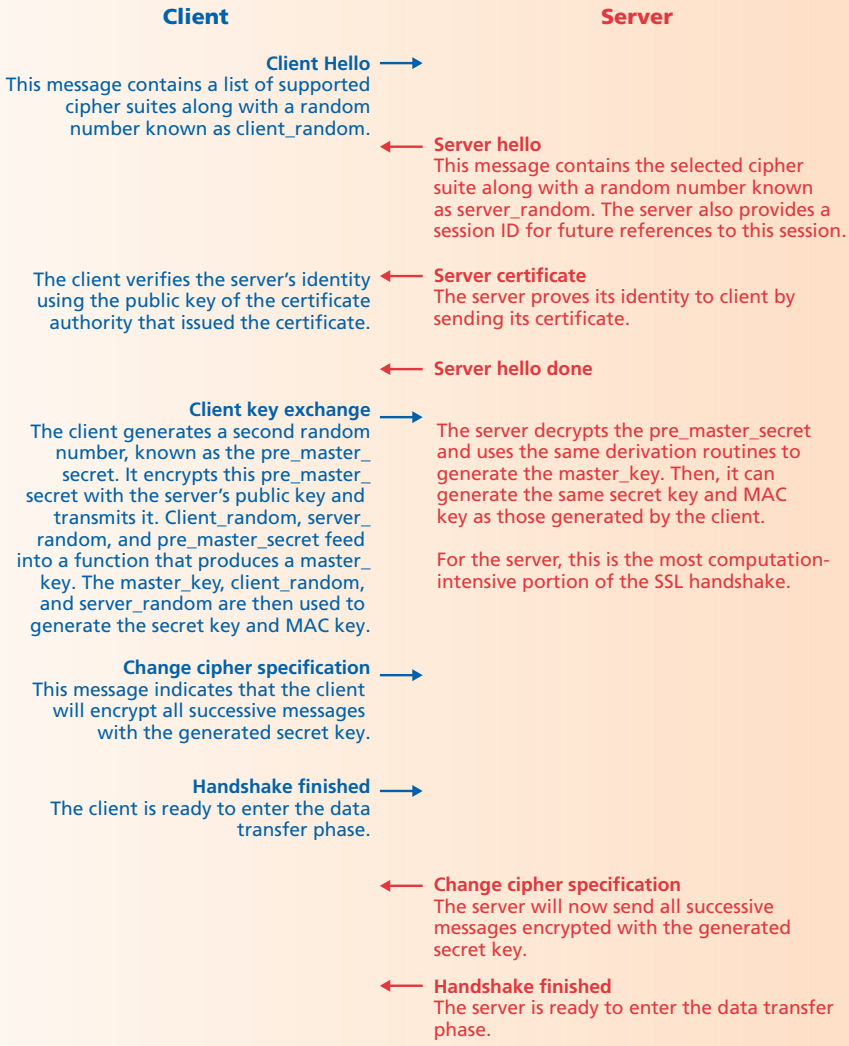
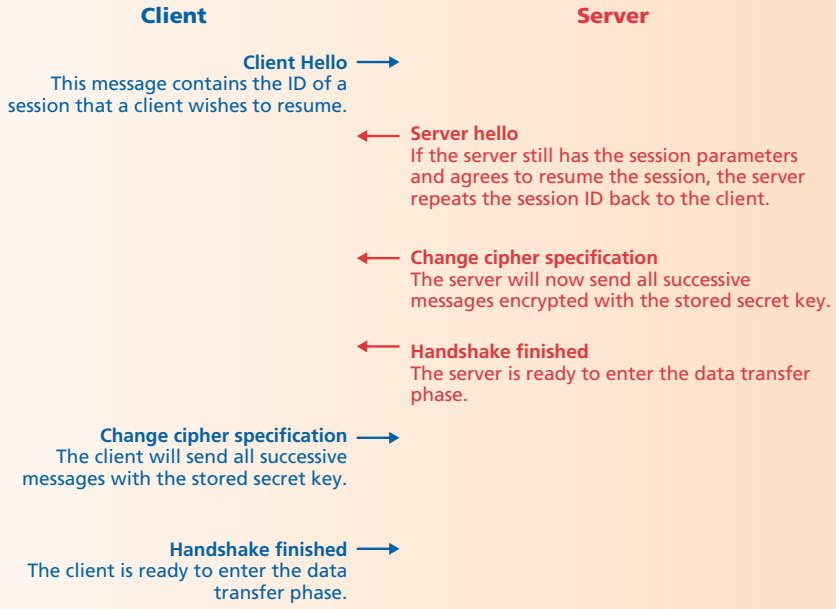


Figure 6. SSL session resumption handshake example.



Message authentication. For every SSL data record transmitted, the sender must calculate and add a MAC. For every SSL data record received, the recipient must verify the MAC.

Administration

Adminstrative components require the least amount of computation.

Certificate and key maintenance. If the client plans to access a site that requires client authentication, it must maintain its certificate and the associated private key. If not, the client does not need to maintain a certificate. The server must always maintain its certificate and the associated private key.

Session ID storage. Both the client and server must maintain a cache of session IDs and associated secret keys to use during a session resumption handshake.

Now that you understand the basics of how SSL works, you can begin to analyze the costs and benefits of the various SSL implementation strategies. A key consideration, of course, is how to handle the cryptographic computation involved.

Because a single Web server services multiple Web clients, the SSL server entity causes much more of a bottleneck during SSL transactions than the client. For this reason, Part 2 of this article focuses on the various options available for implementing an SSL server entity: SSL functions can take place on the Web server with software alone, on the Web server with the assistance of an adapter card, or at the edge content switch with the assistance of an SSL accelerator. You'll see that the alternatives to the software-only implementation can ease your server's computing burden, letting your enterprise handle large numbers of SSL transactions more efficiently. ■

Wesley Chou *is a software engineer at Cisco Systems. Contact him at wschou@cisco.com.*

For further information on this or any other computing topic, visit our Digital Library at http://computer.org/publications/dlib.