How TLS/SSL Works

In this section

TLS/SSL authenticates and secures data transfers by using certificate-based authentication and symmetric encryption keys. This section discusses how the RFC-standard TLS protocol is used in the Windows Server 2003 operating system.

This section is divided into five subsections:

- **Schannel SSP Architecture** illustrates how the Microsoft Security Support Provider Interface (SSPI) in Windows Server 2003 provides a mechanism of generic routines that can access the Secure Channel (Schannel) Security Support Provider (SSP).
- **TLS/SSL Architecture** discusses the Handshake and Record Layer and related sub-protocols of TLS/SSL, as well as the Schannel session cache.
- **TLS/SSL Protocol Processes and Interactions** illustrates full handshake protocol, application data flow, resuming a secure session, and renegotiation with what is included in TLS messages. Some processes — such as certificate mapping, and how internally renegotiations are handled — are specific to Windows systems and might or might not differ in other implementations of the TLS protocol.
- **Network Ports Used by the TLS/SSL** tabulates networks ports that are used for TLS/SSL
- **Related Information** lists links of related information.

The Windows Server 2003 operating system can use three related security protocols to provide authentication and secure communications over the Internet:

- Transport Layer Security (TLS) 1.0
- Secure Sockets Layer 3.0
- Secure Sockets Layer (SSL) 2.0

All three protocols provide authentication through the use of certificates and secure communication through a variety of possible cipher suites. The generic term *cipher suite* refers to a combination of protocols such as key exchange, bulk encryption, and message integrity. Because authentication relies on digital certificates, certification authorities (CAs) like Verisign are an important part of Secure Channel (Schannel). A CA is a mutually trusted third party that confirms the identity of a certificate requestor (usually a user or computer), and then issues the requestor a certificate. The certificate binds the requestor's identity to a public key. CAs also renew and revoke certificates as necessary. For example, if a client is presented with a server's certificate, the client computer might try to match the server's CA against the client's list of trusted CAs. If the issuing CA is trusted, the client will verify that the certificate is authentic and has not been tampered with. Microsoft Internet Explorer and Internet Information Services (IIS) make use of these protocols, and preferably TLS, for Secure Hypertext Transfer Protocol (HTTPS).

This document focuses on TLS, because TLS is replacing SSL and PCT. TLS is standardized in RFC 2246 IETF RFC database.

Note

- Throughout this document *TLS/SSL* refers to the common protocol features of Transport Layer Security and Secure Sockets Layer. *TLS* refers to only Transport Layer Security, and *SSL* refers to only Secure Sockets Layer.

Back to Top

Schannel SSP Architecture

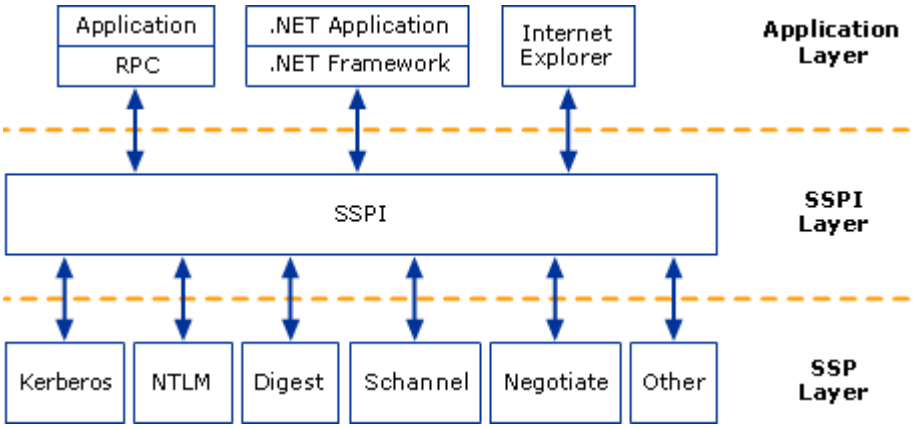
The Windows Server 2003 operating system implements the TLS/SSL protocols as a Security Support Provider SSP, a dynamic-link library (DLL) called Schannel that is supplied with the operating system. Which SSP is used depends on the capabilities of the computer on the other side of the connection and the configuration of the individual application that is being used.

The Microsoft Security Support Provider Interface (SSPI) is the foundation for authentication in Windows Server 2003. That is, applications and infrastructure services that require authentication use SSPI to provide it.

The SSPI is the implementation of the Generic Security Service API (GSSAPI) in Windows Server 2003. For more information about GSSAPI, see RFC 2743 and RFC 2744 in the IETF RFC database.

The default SSPs in Windows Server 2003 — Kerberos, NTLM, Digest, Schannel, and Negotiate authentication protocols — are incorporated into the SSPI in the form of DLLs. Additional SSPs can be incorporated if they can interoperate with the SSPI.

SSPI Architecture



In the Windows Server 2003 operating system, SSPI provides a mechanism that carries authentication tokens over the existing communication channel between the client and server. When two parties need to be authenticated so that they can communicate more securely, the requests for authentication are routed to the SSPI, which completes the authentication process, regardless of the network protocol currently in use. The SSPI returns transparent binary large objects, and then these are passed between the applications, at which point they can be passed to the SSPI layer on that side. Thus, the SSPI enables an application to use various security models available on a computer or network without changing the interface to the security system.

The following table describes the SSP components that are plugged into the SSPI. Each of the protocols in the table is used in different ways in Windows Server 2003 to promote more secure communication in an insecure network environment.

SSP Layer Components

Component	Description
Kerberos V5 authentication	An industry-standard protocol that is used with either a password or a smart card for interactive logon. It is also the preferred authentication method for services in Windows 2000 and Windows Server 2003.
NTLM authentication	A challenge-response protocol that is used to provide compatibility with versions of Windows earlier than Windows 2000.
Digest authentication	An industry standard that is used in Windows Server 2003 for Lightweight Directory Access Protocol (LDAP) and Web authentication. Digest transmits credentials across the network as a Message Digest 5 (MD5) hash or message digest.
Schannel	An SSP that implements SSL and TLS. Schannel is used for applications used in cross-organization environments, such as Web-based server authentication, in which a user attempts to access a secure Web server or corporate access using VPN.
Negotiate	An SSP that can be used to negotiate a specific authentication protocol. When an application calls into SSPI to log on to a network, it can specify an SSP to process the request. If the application specifies Negotiate, Negotiate analyzes the request and picks the best SSP to handle the request, based on customer-configured security policy.

Secure Channel SSP

You can use the Secure Channel (Schannel) SSP for access to Web-enabled services, such as e-mail or personal information served on Web pages.

The Schannel SSP uses public key certificates to authenticate parties. It includes four authentication protocols in its suite. When authenticating parties, it will select one of the four protocols in the following order of preference:

- TLS version 1.0
- SSL version 3.0
- PCT. PCT is turned off by default in Windows Server 2003. PCT has been superseded by Secure Sockets Layer 3.0 and the TLS protocol. The Schannel SSP supports PCT 1.0 for backward compatibility only, but this protocol might not be available in future releases.
- SSL version 2.0

Schannel then selects the most preferred authentication protocol that both parties can support. For example, if a server supports all four Schannel protocols and the client supports only SSL 3.0 and PCT, Schannel uses SSL 3.0 for authentication.

Back to Top

TLS/SSL Architecture

The Schannel authentication protocol suite is based on public key cryptography. The Schannel suite includes Transport Layer Security (TLS), Secure Sockets Layer (SSL) version 3.0, SSL version 2.0, and Private Communications Transport (PCT). All Schannel protocols are based on a client/server model. An Schannel client sends a message to a server, and the server responds with the information needed to authenticate itself. The client and server perform an additional exchange of session keys, and the authentication dialogue ends. When authentication is completed, secure communication can begin between the server and the client using the secret keys established during the authentication process.

Schannel does not require server keys to be stored on domain controllers or in a database, such as Active Directory. Clients, however, must be able to confirm the validity of credentials with a trusted authority. Schannel validates the credentials with the root CA's certificates, which are loaded when you install Windows Server 2003. Therefore, users do not need to establish accounts before authenticating and creating a secure connection with a server.

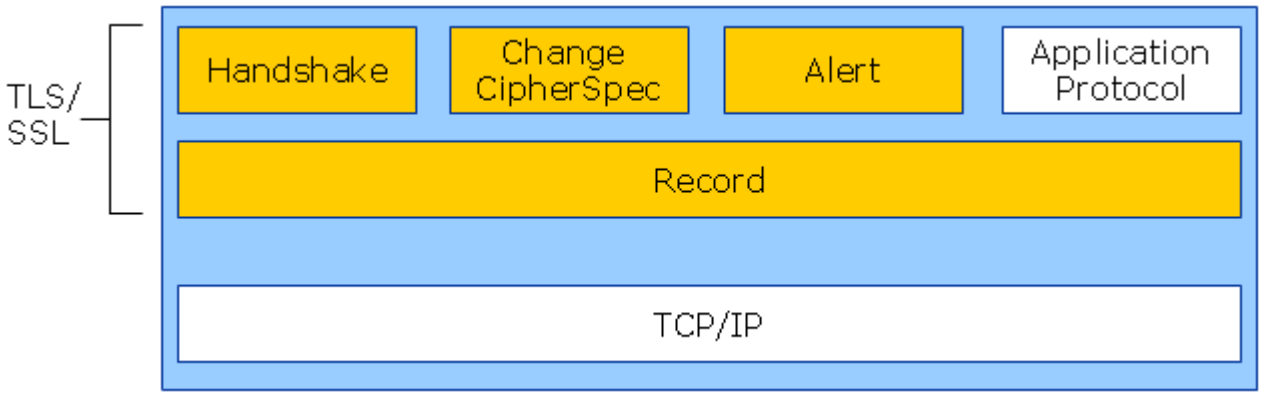
The TLS/SSL security protocol is layered between the application protocol layer and the TCP/IP layer, where it can secure and send application data to the transport layer. Because it works between the application layer and the transport layer, TLS/SSL can support multiple application layer protocols.

TLS/SSL assumes that a connection-oriented transport, typically TCP, is in use. The protocol allows client/server applications to detect the following security risks:

- Message tampering
- Message interception
- Message forgery

The TLS/SSL protocol can be divided into two layers. The first layer consists of the application protocol and the three Handshake sub-protocols: the Handshake Protocol, the Change Cipher Spec Protocol, and the Alert Protocol. The second layer is the Record Protocol. The following figure illustrates the various layers and their components.

TLS/SSL Protocol Layers



The Handshake Protocols

The Handshake protocols of the TLS/SSL protocol are responsible for establishing or resuming secure sessions. The main goals of this layer are to:

- Negotiate cipher suites and compression algorithms.
- Authenticate the server to the client and, optionally, authenticate the client to the server through certificates and public or private keys.
- Exchange random numbers and a pre-master secret. Together with some further data, these values will be used to create the shared secret key that the Record Layer will use to hash and encrypt application data. The shared secret key is called the Master Secret.

The three Handshake sub-protocols are:

- **Handshake**. This sub-protocol is used to negotiate session information between the client and the server. The session information consists of a session ID, peer certificates, the cipher specification to be used, the compression algorithm to be used, and a shared secret that is used to generate keys.
- **Change Cipher Spec**. This sub-protocol changes the keying material that is used for encryption between the client and server. The keying material is raw data that is used to create keys for cryptographic use. The Change Cipher Spec sub-protocol consists of a single message to tell other party in the TLS/SSL session, who is known as the peer, that the sender wants to change to a new set of keys. The key is computed from the information that is exchanged by the Handshake sub-protocol.
- **Alert**. This sub-protocol uses messages to indicate a change in status or an error condition to the peer. There are a wide variety of alerts to notify the peer of both normal and error conditions. A full list can be found in RFC 2246, "The TLS Protocol Version 1.0." Alerts are commonly sent when the connection is closed, an invalid message is received, a message cannot be decrypted, or the user cancels the operation.

Handshake Protocol Functions

The Handshake protocol provides a number of very important security functions. It performs a set of exchanges that starts authentication and negotiates the encryption, hash, and compression algorithms.

Authentication

A certificate is a digital form of identification that is usually issued by a certification authority (CA) and contains identification information, a validity period, a public key, a serial number, and the digital signature of the issuer. For authentication purposes, the Handshake Protocol uses an X.509 certificate to provide strong evidence to a second party that helps prove the identity of the party that holds the certificate and the corresponding private key.

A CA is a mutually trusted third party that confirms the identity of a certificate requestor (usually a user or computer), and then issues the requestor a certificate. The certificate binds the requestor's identity to a public key. CAs also renew and revoke certificates as necessary. For example, if a client is presented with a server's certificate, the client computer might try to match the server's CA against the client's list of trusted CAs. If the issuing CA is trusted, the client will verify that the certificate is authentic and has not been tampered with. Finally, the client will accept the certificate as proof of identity of the server.

Encryption

There are two main types of encryption: *symmetric key* (also known as *shared secret key*) and *asymmetric key* (also known as *public key* or *public-private key*). TLS/SSL uses symmetric key for bulk encryption and public key for authentication and key exchange.

- **Symmetric Key (also known as Private Key)**. In symmetric key encryption, the same key is used to encrypt and decrypt the message. If two parties want to exchange encrypted messages securely, they must both possess a copy of the same symmetric key. Symmetric key cryptography is often used for encrypting large amounts of data because it is computationally faster than asymmetric cryptography. Typical algorithms include Data Encryption Standard (DES), Triple DES (3-DES), RC2, RC4, and Advanced Encryption Standard (AES).
- **Asymmetric Key, also known as Public Key**. Asymmetric key encryption uses a pair of keys that have been derived simultaneously through a complex mathematical process. One of the keys is made public, typically by asking a CA to publish the public key in a certificate for the certificate-holder (also called the subject). The private key is kept secret by the subject and never revealed to anyone. The keys work together, with one being used to perform the inverse operation of the other: If the public key is used to encrypt data, only the private key of the pair can decrypt it; if the private key is used to encrypt, the public key must be used to decrypt. This relationship allows a public key encryption scheme to do two important things. First, anyone can obtain the public key for a subject and use it to encrypt data that only the user with the private key can decrypt. Second, if a subject encrypts data using its private key, anyone can decrypt the data by using the corresponding public key. This is the foundation for digital signatures. The most common algorithm is Rivest, Shamir, and Adleman (RSA).

TLS/SSL uses public key encryption to authenticate the server to the client and, optionally, the client to the server. Public key cryptography is also used to establish a session key. The session key is used in symmetric algorithms to

encrypt the bulk of the data with the faster, less processor-intensive symmetric key encryption.

Hash Algorithms

During the handshake process, the client and server agree on the hash algorithm. A hash is a one-way mapping of values to a smaller set of representative values, so that the size of the resulting hash is smaller than the original message and the hash is unique to the original data. A hash is similar to a fingerprint: a fingerprint is unique to the individual and is much smaller than the original person. Hashing is used to establish data integrity during transport. Two common hash algorithms are Message Digest 5 (MD5) and Standard Hash Algorithm 1 (SHA-1). MD5 produces a 128-bit hash value and SHA-1 produces a 160-bit value.

Note

- Data can be encrypted and decrypted, but you cannot reverse engineer a hash. Hashing is a one-way process. Running the process backward does not recreate the original data. This is why a new hash is computed and then compared to the sent hash.

The hash algorithm includes a value that is used to check the integrity of the transmitted data. This value is established by using either a message authentication code (MAC) or a hashed message authentication code (HMAC). The MAC uses a mapping function to represent the message data as a fixed-length, preferably smaller, value and then hashes the message. The MAC ensures that the data has not been modified during transmission. The difference between a MAC and a digital signature is that a digital signature is also an authentication method. SSL uses a MAC.

The HMAC is similar to the MAC but uses a hash algorithm in combination with a shared secret key. The shared secret key is appended to the data to be hashed. This makes the hash more secure because both parties must have the same shared secret key to prove the data is authentic. TLS uses an HMAC. For more information about HMAC, see RFC 1024 in the IETF RFC Database.

The Change Cipher Spec Protocol

The Change Cipher Spec Protocol signals a transition of the cipher suite to be used on the connection between the client and server. This protocol is composed of a single message which is encrypted and compressed with the current cipher suite. This message consists of a single byte with the value 1. Message after this will be encrypted and compressed using the new cipher suite.

The Alert Protocol

The Alert Protocol includes event-driven alert messages that can be sent from either party. Following an alert message, the session is either ended or the recipient is given the choice of whether or not to end the session. The alerts are defined in the TLS specification in RFC 2246. The following table lists the possible alert messages and their descriptions. Schannel SSP will only generate these alert messages at the request of the application.

Event-Driven Alert Messages from the Alert Sub-protocol

Alert Message	Description
close_notify	Notifies the recipient that the sender will not send any more messages on this connection.
unexpected_message	Received an inappropriate message This alert should never be observed in communication between proper implementations. This message is always fatal.
bad_record_mac	Received a record with an incorrect MAC. This message is always fatal.
decryption_failed	Decryption of a TLSCiphertext record is decrypted in an invalid way: either it was not an even multiple of the block length or its padding values, when checked, were not correct. This message is always fatal.
record_overflow	Received a TLSCiphertext record which had a length more than 2^14+2048 bytes, or a record decrypted to a TLSCompressed record with more than 2^14+1024 bytes. This message is always fatal.
decompression_failure	Received improper input, such as data that would expand to excessive length, from the decompression function. This message is always fatal.
handshake_failure	Indicates that the sender was unable to negotiate an acceptable set of security parameters given the options available. This is a fatal error.
bad_certificate	There is a problem with the certificate, for example, a certificate is corrupt, or a certificate contains signatures that cannot be verified.
unsupported_certificate	Received an unsupported certificate type.
certificate_revoked	Received a certificate that was revoked by its signer.

certificate_expired	Received a certificate has expired or is not currently valid.
certificate_unknown	An unspecified issue took place while processing the certificate that made it unacceptable.
illegal_parameter	Violated security parameters, such as a field in the handshake was out of range or inconsistent with other fields. This is always fatal.
unknown_ca	Received a valid certificate chain or partial chain, but the certificate was not accepted because the CA certificate could not be located or could not be matched with a known, trusted CA. This message is always fatal.
access_denied	Received a valid certificate, but when access control was applied, the sender did not proceed with negotiation. This message is always fatal.
decode_error	A message could not be decoded because some field was out of the specified range or the length of the message was incorrect. This message is always fatal.
decrypt_error	Failed handshake cryptographic operation, including being unable to correctly verify a signature, decrypt a key exchange, or validate a finished message.
export_restriction	Detected a negotiation that was not in compliance with export restrictions; for example, attempting to transfer a 1024 bit ephemeral RSA key for the RSA_EXPORT handshake method. This message is always fatal.
protocol_version	The protocol version the client attempted to negotiate is recognized, but not supported. For example, old protocol versions might be avoided for security reasons. This message is always fatal.
insufficient_security	Failed negotiation specifically because the server requires ciphers more secure than those supported by the client. Returned instead of handshake_failure. This message is always fatal.
internal_error	An internal error unrelated to the peer or the correctness of the protocol makes it impossible to continue, such as a memory allocation failure. The error is not related to protocol. This message is always fatal.
user_canceled	Cancelled handshake for a reason that is unrelated to a protocol failure. If the user cancels an operation after the handshake is complete, just closing the connection by sending a close_notify is more appropriate. This alert should be followed by a close_notify. This message is generally a warning.
no_renegotiation	Sent by the client in response to a hello request or sent by the server in response to a client hello after initial handshaking. Either of these would normally lead to renegotiation; when that is not appropriate, the recipient should respond with this alert; at that point, the original requester can decide whether to proceed with the connection. One case where this would be appropriate would be where a server has spawned a process to satisfy a request; the process might receive security parameters (key length, authentication, and so on) at startup and it might be difficult to communicate changes to these parameters after that point. This message is always a warning.

The Record Layer

As specified by RFC 2246, the Record Layer might have four functions:

- It fragments the data coming from the application into manageable blocks (and reassemble incoming data to pass up to the application). Schannel SSP does not support fragmentation at the Record Layer.
- It compresses the data and decompresses incoming data. Schannel SSP does not support compression at the Record Layer.
- It applies a Message Authentication Code (MAC), or hash/digest, to the data and uses the MAC to verify incoming data.
- It encrypts the hashed data and decrypts incoming data.

Record Protocol Functions

The Record Protocol receives and encrypts data from the application layer and delivers it to the Transport Layer. Then it takes the data, fragments it to a size appropriate to the cryptographic algorithm, optionally compresses it (or, for data received, decompresses it), applies a MAC or HMAC and then encrypts (or decrypts) the data using the information negotiated during the Handshake Protocol. HMAC is supported only by TLS.

The resulting encrypted data is then passed to the transport layer.

Record Layer Encryption States

The record layer changes encryption states during the session setup. That is, different encryption methods are used at different stages of the process.

- **No Encryption Used**. Prior to the negotiating a cipher suite and deciding which security methods to use, the state is Null. No message authentication or encryption is performed.
- **Public Key Encryption Used** (Uses public-private key pair). Once a cipher suite is negotiated and certificates are exchanged, The record layer hashes outgoing data with the appropriate MAC (typically both MD5 and secure hash algorithm (SHA-1), as required for RSA signing) and encrypts it with the sender's private key. Incoming data is decrypted with the sender's public key.
- **Symmetric Key Encryption Used** (Uses shared session key rather than public-private key pair). Near the end of the Handshake sequence, the handshake protocol exchanges data that allows both the client and the server to compute a Master Secret and then to derive the Client Write MAC Secret, Server Write MAC Secret, Client Write Key, and Server Write Key from it.

  When these keys are available, the client and server send a message to Change Cipher Spec. At that point, the client and server stop using the public-private key pair and start using the shared session keys derived from the Master Secret.

Back to Top

TLS/SSL Processes and Interactions

This section provides a detailed explanation of the TLS/SSL protocol, specifically the handshake protocol, its associated messages and alerts, and the record protocol.

The first three operations listed below take place using the Handshake Protocol. The parties negotiate which cipher suite they will use, authenticate, exchange keys, and exchange application data.

Handshake and Cipher Suite Negotiation

First, the client and server contact each other and choose a common cipher suite. The suite will include:

- A key exchange method, which determines how the shared master key will be exchanged.
- A bulk encryption method which determines how application data will be encrypted.
- A MAC, which determines how application data will be hashed and signed to prove integrity.

Authentication

The server always authenticates its identity to the client. However, the client might not need to authenticate with the server, depending on the application. The authentication method (primarily which digital certificate format will be used) depends on the negotiated cipher suite.

Key Exchange

After choosing a cipher suite, the client and server exchange a key (or the information needed to create a key) that they will use for data encryption, which again depends on the negotiated cipher suite's requirements. The key exchange operation requires the following things:

- The random values are created called the Client Random and the Server Random.
- The client generates a random Pre-Master Secret.

The client must transmit the Pre-Master Secret securely to the server.

- With RSA key exchange, the Pre-Master Secret is encrypted with the server's public key. The client gets this key from the server's certificate.
- With EDH key exchange, the pre-master secret is the result of the EDH operation. In this case the ephemeral DH key is signed by the server's private key.

Both client and server use the Pre-Master Secret to create a shared Master Secret. This is done by hashing the pre-master secret together with the ClientRandom and ServerRandom values. The Master Secret is used to create four keys that are shared by the client and server:

- **Client Write MAC Secret.** This key is added to client message hashes. The client uses the key to create the

initial hash. The server uses it to authenticate client messages.

- **Server Write MAC Secret.** This key is added to server message hashes. The server uses the key to create the initial hash. The client uses it to authenticate server messages.
- **Client Write Key**. The client uses this key to encrypt messages. The server uses the key to decrypt client messages.
- **Server Write Key.** The server uses this key to encrypt messages. The client uses the key to decrypt server messages**.**

The last operation takes place at the Record Layer using the Record Protocol.

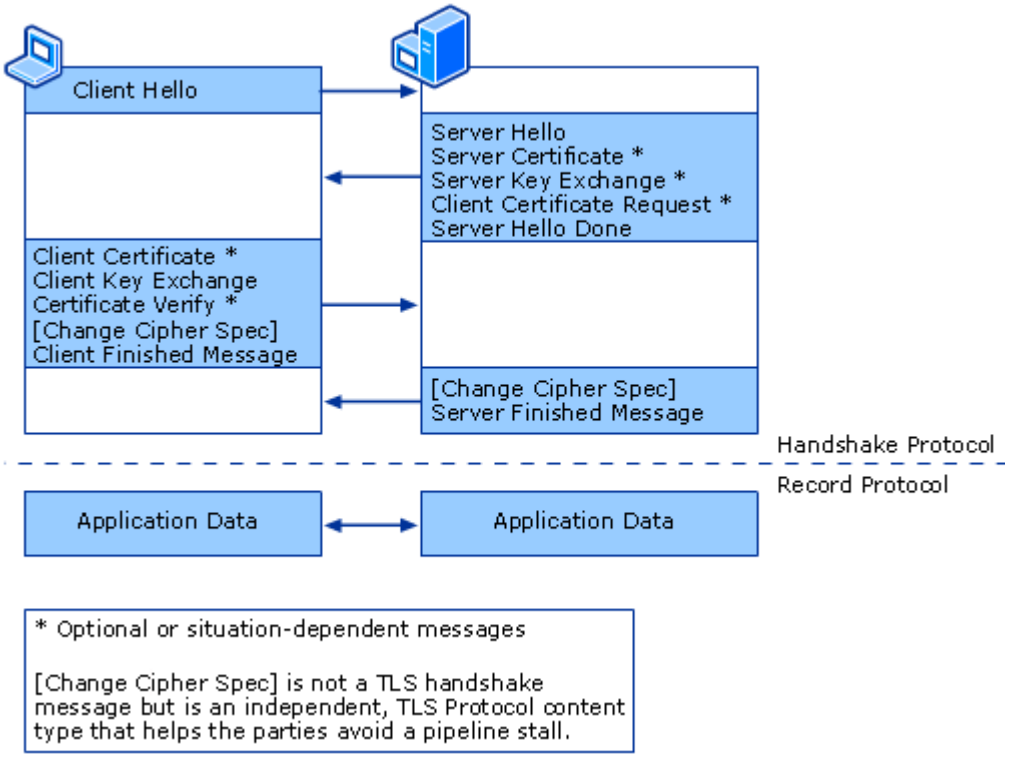
Application Data Exchange

The client application and the server application communicate with each other. All data is encrypted using the negotiated bulk encryption method.

TLS/SSL Message Sequence Overview

The process of authenticating and establishing an encrypted channel with a server using the Schannel authentication protocol involves the following steps:

Full TLS Handshake



The Handshake protocols manage this process of exchanging key generation material, and it uses the Handshake Protocol.

Once hashing and encryption keys are ready for use, the Record Layer takes over. It uses the Record Protocol to secure application data, using the keys created in the Handshake process.

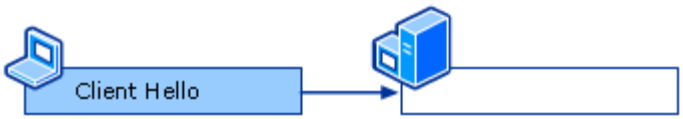
The Full Handshake Protocol

The handshake protocol is a series of sequenced messages that negotiate the security parameters of a data transfer session.

Client Hello Messages

The Client Hello is typically the first message in the TLS/SSL session setup sequence. The protocol allows the server to request a hello, but this is application-specific and not part of the normal sequence.

Client Hello Messages



Client Hello Message

The client initiates a session by sending a Client Hello message to the server. The Client Hello message contains:

- **Version Number**. The version number of the highest version that the client supports. This is sent by the client to the server. Version 2 is used for SSL 2.0, version 3 for SSL 3.0, and version 3.1 for TLS. Although the IETF RFC for TLS is TLS version 1.0, the protocol uses 3.1 in the version field to indicate that it is a later version, with more functionality than SSL 3.0.
- **Client Random**. A 4-byte number that consists of the client's date and time, plus a 28-byte cryptographically-generated pseudorandom number. This is used in the calculation of the Master Secret from which the encryption keys are derived.
- **(Optional) Session Identification**. A byte string used to identify an active or resumable session state. Enables the client to resume a previous session. Resuming a previous session can be useful, because creating a new session requires processor-intensive public key operations that can be avoided by resuming an existing session with its established session keys. Previous session information, identified by the SessionID, is stored in the respective client and server session caches. This is empty if the client is not resuming a session.
- **Cipher Suite**. The list of cipher suites available on the client. An example of a cipher suite is TLS_RSA_WITH_DES_CBC_SHA, where TLS is the protocol version, RSA is the algorithm that will be used for the key exchange, DES_CBC is the encryption algorithm (using a 56-bit key in CBC mode), and SHA-1 is the hash function.

  Note
  - Microsoft applications typically specify RSA as the key exchange algorithm, RC4 as the encryption method, and MD5 as the Message Authentication Codes.
- **Compression Algorithm**. The client's supported compression algorithms. Compression is optional and is one of the four operations that the Record Protocol might perform on all data after the Hello sequence is finished.

Server Session Cache Allows Session Resume

The Client Hello is used to either start a new session or to resume an existing session. If the Client Hello includes a Session ID, the client is attempting to resume a session.

The server maintains a session cache to allow fast resumption of recent sessions, similar to a ticket cache in Kerberos. Resuming a session does not require certificate exchanges, so it is much quicker than the normal Hello sequence.

The application determines whether clients are allowed to resume sessions and how long idle session IDs are valid.

The server uses CryptoAPI to manage both the session ID and the certificate cache.

Note
- The Client Hello can be initiated at any time during an existing session and is not limited to just session initialization. The server application might request a new Hello periodically to request client authentication based on the resource that is requested. Either the client application or the server might request a new handshake to refresh encryption keys.

TLS 1.0 Cipher Suites

Schannel supports the cipher suites in the following table for TLS 1.0. In this table, the following acronyms are used:

CBC = cipher block chaining

DES = Data Encryption Standard

DHE = Ephemeral Diffie-Hellman

DSS = Digital Signature Standard

The suites are listed in order of preference.

TLS 1.0 Cipher Suites

Key Exchange	Cipher	Hash
RSA	RC4 128	MD5
RSA	RC4 128	SHA-1
RSA	3DES EDE CBC	SHA-1
DHE DSS	3DES EDE CBC	SHA-1
RSA	DES CBC	SHA-1
DHE DSS	DES CBC	SHA-1
RSA Export 1024	RC4 56	SHA-1
RSA Export 1024	DES CBC	SHA-1
DHE DSS Export 1024	DES CBC	SHA-1
RSA Export	RC4 40	MD5
RSA Export	RC2 CBC 40	MD5
RSA	None	MD5
RSA	None	SHA-1

SSL 3.0 Cipher Suites

Schannel supports the cipher suites listed under TLS 1.0 Cipher Suites.

SSL 2.0 Cipher Suites

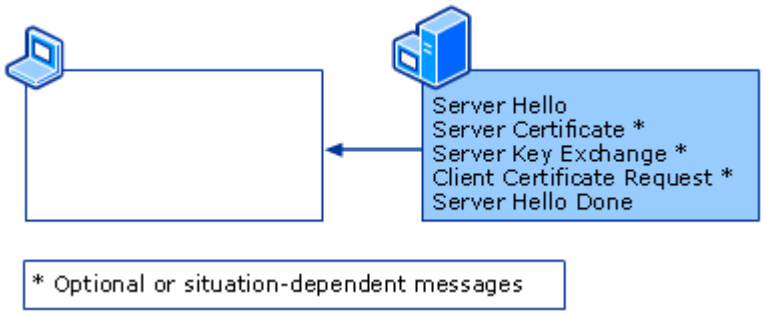
Schannel supports the following cipher suites for SSL 2.0. The suites are listed in order from most secure to least secure.

SSL 2.0 Cipher Suites

Cipher	Hash
RC4 128	MD5
DES 192 EDE3 CBC	MD5
RC2 CBC 128 CBC	MD5
DES 64 CBC	MD5
RC4 128 Export 40	MD5

Server Responses to Client Hello

Server Responses to Client Hello



Server Hello Message

The server responds with a Server Hello message. The Server Hello message includes:

- **Version Number**. The server sends the highest version number that is supported by both sides. This is the protocol version that will be used during the connection.
- **Server Random[32]**. ServerRandom[32] is a 4-byte representation of the server's date and time plus a 28-byte, cryptographically-generated, pseudorandom number. This number, along with the Client Random, is

used by both the client and the server to generate the Master Secret from which the encryption keys will be derived.

- **Session Identification** (if any). This can be one of three choices.
  - **New session ID.** The client did not indicate a session to resume, so a new ID is generated. A new session ID is also generated when the client indicates a session to resume, but the server can't or won't resume that session.
  - **Resumed Session ID**. The ID is the same as indicated in the Client Hello. The client indicated a specific session ID to resume and the server is willing to resume that session.
  - **Null**. This is a new session, but the server is not willing to resume it at a later time, so no ID is returned.
- **Cipher Suite**. The server chooses the strongest cipher that both the client and server support. If there are no cipher suites that both parties support, the session is ended with a *handshake failure* alert.
- **Compression Algorithm**. If used, specifies the compression algorithm to use.

Server Certificate Message

The server sends its certificate to the client. The server certificate contains the server's public key. The client uses this key to authenticate the server and to encrypt the Premaster Secret. The Server Certificate message includes:

- **The server's certificate list**. The first certificate in the list is the server's X.509v3 certificate that contains the server's public key.
- **Other validating certificates**. All other validating certificates, up to but not including the root certificate from the CA, signed by the CA.

The client also checks the name of the server in the certificate to verify that it matches the name the client used to connect. For example, if the user types http://www.contoso.com as the URL in the browser, the certificate contains a subject name of www.contoso.com or *.contoso.com. Internet Explorer warns the user if these names do not match; other client applications typically fail the connection directly.

(Optional) Server Key Exchange Message

The server creates and sends a temporary key to the client. This key can be used by the client to encrypt the Client Key Exchange message later in the process. The step is only required when the server's certificate does not contain a public key that is suitable for key exchange or when the cipher suite mandates the use of an ephemeral key for the key exchange operation. This message is also used when the server uses a DSS certificate. In this case, the server public key is not suitable for key exchange, and so an ephemeral DH key is created by the server and sent in the Server Key Exchange message.

In these situations, the server must create and send a temporary key that is used instead of the server's public key. The key that is created depends on the cipher suite. With export versions of RSA that do not allow a public key greater than 512 bits, the temporary shorter key is signed with the unusable public key for authenticity.

Why is this important? The client will need this key to encrypt the Premaster Secret, whichis discussed below, in the Client Key Exchange message.

Note

- This message is not used in non-export versions of Microsoft applications, since non-export RSA certificates will always include the server's public key in its certificate.

(Optional) Client Certificate Request Message

The server must always present its certificates to the client, but the client is not always required to authenticate itself. Therefore, the client is not always required to send its certificates to the server. If the server does not require client authentication, then this message is not sent.

This step might be used for Web sites such as a banking Web site, where the server must confirm the identity of the client before providing sensitive information. If the application requires mutual authentication, the server sends a Client Certificate Request. The Client Certificate Request message includes:

- The type of certificate required (typically RSA or DSS)
- A list of acceptable CAs

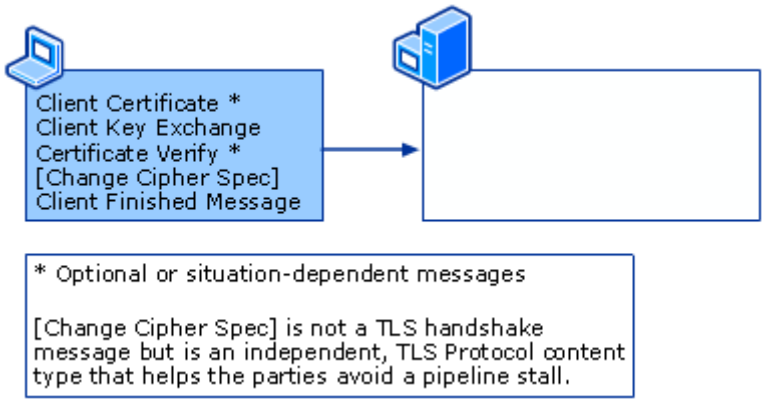
Server Hello Done Message

This message indicates that the server is finished and awaiting a response from the client. This message has no content. It signals that the Server Hello sequence is finished.

Client Responses to Server Hello

The client receives the server's message and checks the certification hierarchy of the server's certificate.

Client Responses to Server Hello



Client Certificate Message (Required if Requested)

If the server sent a Client Certificate Request, the client sends its certificate to the server for client authentication. The client's certificate contains the client's public key. The Client Certificate message includes the client's **certificate list**. The first certificate in the list is the client's X.509v3 certificate that contains the client's public key. After that are other validating certificates, up to but not including the root certificate from the CA, signed by the CA.

Client Key Exchange Message

The client sends a Client Key Exchange message after computing the premaster secret using the two random values that are generated during the Client Hello message and the Server Hello message. Before it is transmitted to the server, the premaster secret is encrypted by the public key from the server's certificate. Both parties compute the master secret locally and derive the session key from it.

If the server can decrypt this data and complete the protocol, the client is assured that the server has the correct private key. This step is crucial to prove the authenticity of the server. Only the server with the private key that matches the public key in the certificate can decrypt this data and continue the protocol negotiation.

The Client Key Exchange message includes:

- **Client's protocol version**. The server will verify that it matches the original value sent in the Client Hello message. This measure guards against rollback attacks. Rollback attacks work by manipulating the message in order to cause the server and the client to use a less secure, earlier version of the protocol.
- **Pre-master secret**. This is the client-generated number (48-byte for RSA), encrypted with the server's public key, that is used with the Client Random and the Server Random to create the Master Secret.

Certificate Verify Message (Required if Client Certificate Is Sent)

This message is sent only if the client previously sent a Client Certificate message. The client is authenticated by using its private key to sign a hash of all the messages up to this point. The server verifies the signature with the public key of the signer, which ensures that it was signed with the client's private key. This message contains a long signature to verify the client's certificate, if one was requested and sent.

For RSA, the signature consists of:

- An MD5 hash of all previous handshake messages.
- An SHA-1 hash of all previous handshake messages.
- Both hashes, which are concatenated and encrypted with the client's private key.

For DSS, the signature consists of:

- An SHA-1 hash of all previous handshake messages.
- Encrypted with the client's private key.

Change Cipher Spec Message

The Change Cipher Spec message notifies the server that all future messages including the Client Finished message

are encrypted using the keys and algorithms just negotiated. At this point, client and server are authenticated and the client has sent the pre-master secret. Both the client and the server have calculated the Master Secret. Up until now, however, any encryption has used the client's or server's private/public keys. The Change Cipher Spec message tells the server that the client is ready to use the Write Key for all further encryption.

Note

- The Change Cipher Spec message is not part of the Handshake Protocol. The TLS RFC defines a separate protocol for Change Cipher Spec. This allows developers to define a specific Change Cipher Spec message.

Computing the Master Secret Key and Subsequent Keys

The Handshake sequence securely exchanges data that is used to create the Master Secret. This secret, however, is never used by itself in encryption; rather, several keys are derived from the Master Secret. These keys are then used for message authentication and encryption.

Note

- The details in this section and the next (hashing in the Record Layer) provide a glimpse inside the computational "black box" and are meant to illustrate the complexity that is necessary to secure communications. The key material exchange specifics change over time as new algorithms are developed.

Computing the Master Secret

The Master Secret results from the pseudorandom material that is exchanged in the Client Hello message and Server Hello message that is discussed earlier in the sections on the Client and Server Final messages. Both the server and the client create the Master Secret key. It is never exchanged.

To create the Master Secret key, the system passes the following as variables to a pseudorandom function (PRF):

- The 48-byte Pre-Master Secret.
- The literal phrase "master secret"
- The concatenation of Client Random number and Server Random number.

The result is the Master Secret.

MAC Secret and Write Encryption Keys

To enable message encryption and hashing, four keys need to be created on both the server and the client:

- Client Write MAC Secret
- Server Write MAC Secret
- Client Write Key
- Server Write Key

These keys are never exchanged.

To produce these keys, the Master Secret is passed to the PRF until enough output has been generated to yield a value that is divided into six parts. (The last two parts are only generated for non-export block ciphers.)

Client Write MAC Secret This key is added to client message hashes. The client uses the key to create the initial hash. The server uses it to authenticate client messages.

Server Write MAC Secret This key is added to server message hashes. The server uses the key to create the initial hash. The client uses it to authenticate server messages.

Client Write Key The client uses this key to encrypt messages. The server uses the key to decrypt client messages.

Server Write Key The server uses this key to encrypt messages. The client uses the key to decrypt server messages.

Finished Message

The Finished message is a hash of the entire conversation which provides further authentication of the client. This message is a complicated verification that the client who is sending the Finished message is truly the client who started the Hello conversation. It is the first message that the Record Layer encrypts and hashes with the Write Key and the Write MAC Secret.

Construction of the message at the Handshake Protocol

To generate the message contents, the system passes the following as variables to a pseudorandom function

(PRF):

- The Master Secret.
- The literal phrase "client finished".
- The concatenation of an MD5 hash of all previous handshake messages and an SHA-1 hash of all previous handshake messages.

The Handshake protocol passes the result to the Record Layer.

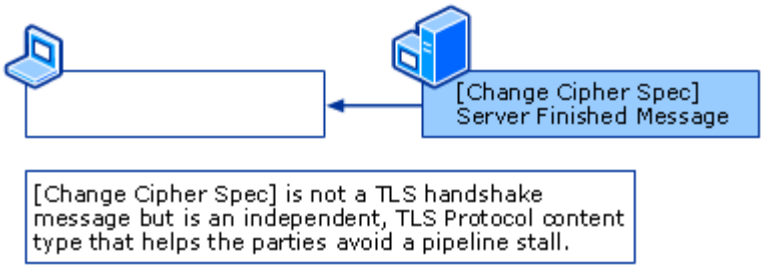
Completing the message at the Record Layer

The Record Layer hashes the data using HMAC with the Client Write MAC Secret, which is derived from the Master Secret. Then, the Record Layer encrypts the data with the Client Write Key, which is also derived from the Master Secret.

Server Finished Messages

If the server has a copy of the private half of its public key pair, it can decrypt the ClientKeyExchange message and generate a matching set of encryption and MAC keys. The server responds to the client with a request to communicate with secret key cryptography, so that both parties have acknowledged that they will use the cryptography and keys that they have agreed upon.

Server Finished Messages



Change Cipher Spec Message

This message notifies the client that the server will begin encrypting messages with the keys that were just negotiated. The server switches its record layer security state to symmetric encryption using the session keys.

Finished Message

This message is a hash of the entire exchange to this point using the session key and the MAC Secret. If the client can successfully decrypt this message and validate the contained hashes, the client is assured that the TLS/SSL handshake was successful, and the keys computed on the client machine match those computed on the server.

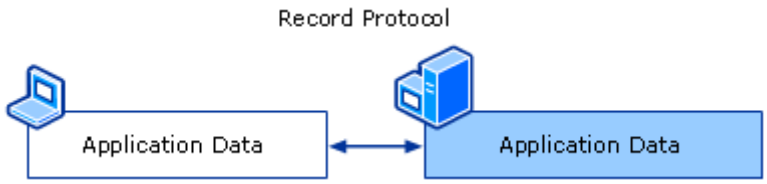
This message is the same as the Client Finished message with these differences:

- The literal phrase "server finished" is used instead of "client finished".
- The hashed handshake messages include the client's Client Finished message.
- The server uses the Server Write MAC Secret and Server Write Key for hashing and encrypting at the Record Layer.

Application Data Flow

So far in this process, client and server applications have been authenticated through the use of certificates and public/private keys, a Premaster Secret and other data have been exchanged to create the Master Secret, and both the server and the client have successfully proven that they have not changed identities throughout the Hello sequence. Now the applications can begin to communicate, using the established keys and parameters. The Record Layer fragments, compresses, hashes, and encrypts all further communications. When necessary, it decrypts, verifies, decompresses, and reassembles the communication

Record Protocol



The Record Protocol

When the record protocol receives the data from the application layer, it might perform the following tasks:

- Fragments the data into blocks or reassembles fragmented data into its original structure. Schannel does not support fragmentation at the Record Layer.
- Numbers the sequence of data blocks in the message to protect against attacks that attempt to reorder data.
- Compresses or decompresses the data using the compression algorithm negotiated in the handshake protocol. Schannel does not support compression at the Record Layer.
- Encrypts or decrypts the data using the encryption keys and cryptographic algorithm negotiated during the handshake protocol.
- Applies an HMAC (or, for SSL 3.0, a MAC) to outgoing data. It then computes the HMAC and verifies that it is identical to the value that was transmitted in order to check data integrity when a message is received.

Once the record protocol has completed its operations on the data, it sends the data to the application for transmission. If the data is incoming, it is sent to the appropriate process for reception.

HMAC Hashing in the Record Layer

Fragmented, compressed application data is hashed as it passes through the Record Layer before being encrypted.

Note

- The details in this section provide a glimpse inside the computational "black box" and are meant to illustrate the complexity necessary to secure communications. The key material exchange specifics change over time as new algorithms are developed.

The MAC consists of two values that are hashed:

- The secret key value: Client or Server Write MAC Secret (derived from the Master Secret).
- A concatenation of:
  - The record’s sequence number.
  - Packet type.
  - Protocol version.
  - Packet length.
  - The non-encrypted application data.

The result of this hashing and the original compressed data fragment are then encrypted and passed down to TCP.

When the message is decrypted at its destination, a new hash is computed, based on the compressed fragment and the MAC Secret. The new hash is compared to the hash that was sent in the message. If they match, the message’s data integrity is verified.

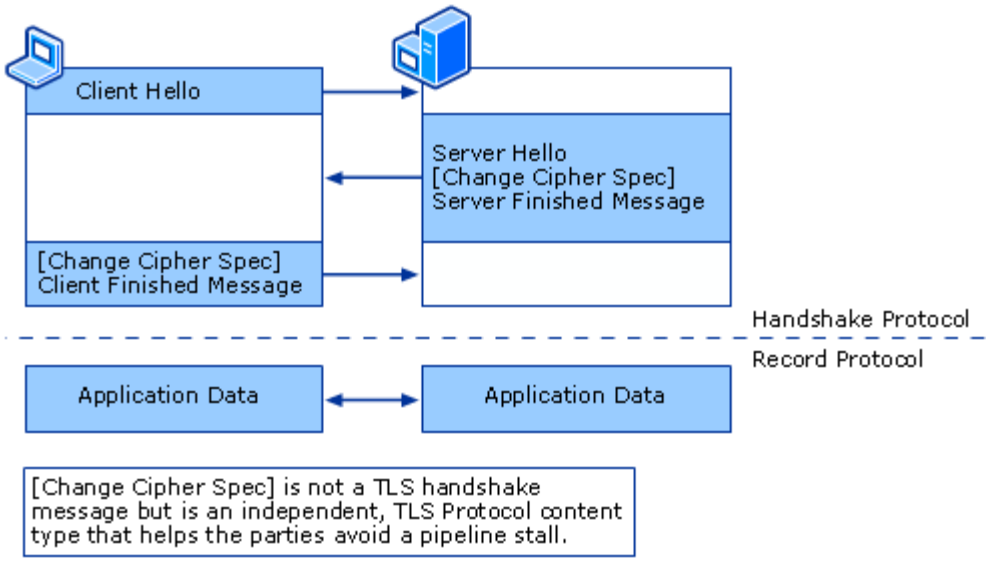
Note

- Data can be encrypted and decrypted, but you cannot reverse engineer a hash. Hashing is a one-way process. Running the process backward does not recreate the original data. This is why a new hash is computed and then compared to the sent hash.

Resuming a Secure Session

Once a secure session has been established between a client and a server, the client can attempt to resume the session in the future, using the same keys used in the previous session. If the server is able to resume the session, then the abridged version of the Handshake Protocol below will occur. If not, then a full handshake will occur.

Resume Session Messages



- The client sends a Client Hello message using the Session ID of the session to be resumed.
- The server checks its session cache for a matching Session ID. If a match is found and the server is able to resume the session, it sends a Server Hello message with the Session ID.

  Note
  - If a session ID match is not found, the server generates a new session ID and the TLS client and server perform a full handshake.
- The server must send a Change Cipher Spec message to notify the client that the server will begin encrypting messages with a new set of encryption and MAC keys, based on the cached Master Secret and the Client Random Server Random values from the current connection. This ensures that every connection has its own set of encryption keys. The server switches its record layer security state to symmetric encryption using the session keys.
- The server sends a Server Finished message.
- The client must send a Change Cipher Spec message to notify the server that the client will start using the session keys for hashing and encrypting messages.
- The client sends a Client Finished message.
- The client and server can now resume application data exchange over the secure channel.

Renegotiation

Renegotiation Methods

Renegotiation can originate from either the client or the server with the following exceptions:

- No client-side renegotiation in Windows 2000 and Windows XP.
- No renegotiation in SSL 2.0.

Server Renegotiation Scenarios

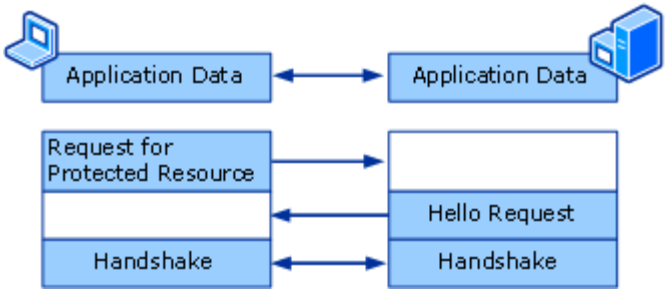
The server can request renegotiation in two situations:

- After the initial handshake, the client requests an access-protected resource. The server can initiate a renegotiation to ask for client authentication. This is the case where the server sends the Hello Request. When the client receives a Hello Request, it begins a new handshake by sending a Client Hello message.
- To refresh keys. This is performed in the same way as the second case, above.

Note
- Schannel does not allow the client to ignore the Hello request. The client must initiate a new handshake by sending a Client Hello message, or the server will close the connection with a fatal error.

Typical Server Renegotiation

Server Initiated Renegotiation



Server-initiated renegotiation uses the following procedure:

- The client and server successfully complete a full TLS/SSL handshake.
- The client requests for an access-protected resource.
- The server sends a Hello Request message to the client.
- The client receives the Hello Request message.

Note

- Schannel does not allow the client to ignore the Hello request, but other clients might. The client must initiate a new handshake by sending a Client Hello message or the Windows server closes the connection.

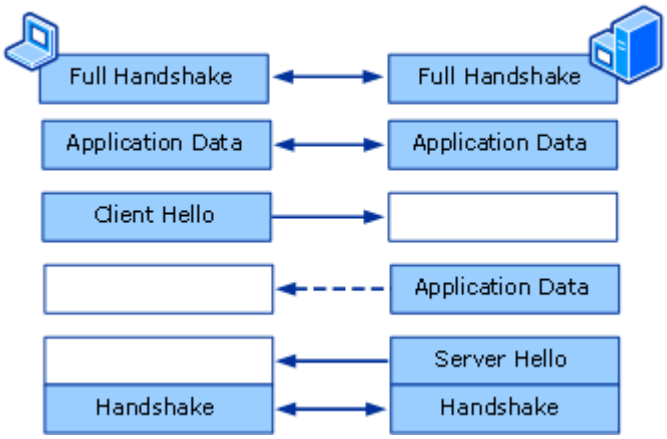
- The client sends a Client Hello message to the server to initiate a new full handshake.

Client Renegotiation Scenarios

The client can also initiate a renegotiation anytime by sending a Client Hello message. This is usually performed to refresh encryption keys.

Typical Client Renegotiation

Client Initiated Renegotiation



Client-initiated renegotiation uses the following procedure:

- The client and the server successfully complete a full SSL handshake.
- The client needs to refresh encryption keys. The client sends a Client Hello message to the server.
- The server receives the Client Hello message. The server has no way of knowing that this message is not application data, and so it passes the message on. It is returned as an "extra" buffer containing the unprocessed Client Hello message with a flag signaling that it is a renegotiation.

Note

- The server might not respond to the renegotiation request right away. In the most general case, the server might have already sent some application data at the same time that the client is requesting the renegotiation, or it might decide to send some data before responding to the renegotiation request. However, the client will never send more application data after it has initiated a renegotiation.

- The server sends the Server Hello group of messages to the client.

- The client receives the Server Hello messages. The client has no way of knowing that this message is not application data, and so it passes the message on. It is returned as an "extra" buffer containing the unprocessed server hello messages with a flag signaling that it is a renegotiation
- The client sends the next set of handshake messages, which are sent to the server, and the handshake proceeds as usual.

The Schannel Session Cache

The first time that a client connects to a server, a full handshake is performed. Once this is complete, the Master Secret, Cipher Suite, and certificates are stored in the session cache on the respective client and server machines. TLS/SSL supports a reconnect operation that can be used to enable a client and server to resume a previously-negotiated session. This allows subsequent connections between the same client and server to use a reconnect handshake rather than a full handshake. This is desirable in many cases because the TLS/SSL reconnect requires far less CPU and network bandwidth because there is no need for a key exchange or certificate exchange. But there are cases in which reconnects can be troublesome; such as during performance testing or when the client never reconnects to the same Web server.

Schannel supports reconnects by use of the Schannel session cache because it keeps a list of previous TLS/SSL sessions that are established with the current authenticated logon data that is used by a security principal to establish its identity. The sessions are referenced by the TLS/SSL session ID. Schannel currently maintains up to ten thousand cached sessions for a maximum of ten hours.

How Schannel Uses Certificate Mapping

When a server application requires client authentication, Schannel automatically attempts to map the certificate that is supplied by the client to a user account. You can authenticate users who log on with a client certificate by creating mappings, which relate the certificate information to a Windows user account. After you create and enable a certificate mapping, each time a client presents a client certificate; your server application automatically associates that user with the appropriate Windows user account.

Types of Mapping

In most cases, a certificate is mapped to a user account in one of two ways: a single certificate is mapped to a single user account (one-to-one mapping), or multiple certificates are mapped to one user account (many-to-one mapping). By default schannel will use the following four certificate mapping methods, in order of preference.

S4U Certificate mapping

This is new for Windows Server 2003. Schannel uses the UPN from the client certificate and attempts an S4U logon using the Kerberos security package. The CA that issued the client certificate must have NTAuth trust. The domain of the server and the user must be running in Windows Server 2003 function level. This is the only mapping method that works even when the server is in a different forest than the client user.

User Principal Name mapping

Enterprise CAs place an entry, called a user principal name (UPN), in each certificate. The UPN looks very much like an e-mail name. The UPN is unique within an Active Directory forest. The UPN finds the user’s account in Active Directory, and that account is logged on. UPN mappings are implicit and therefore do not require administrators to create and maintain.

Schannel uses the UPN from the client certificate, and queries Active Directory for a matching user account. The CA that issued the client certificate must have NTAuth trust.

One-to-One Mapping also known as subject/issuer mapping

A method that maps a single user certificate to a single Active Directory user account. Unlike UPN mapping, the administrator must explicitly map a certificate to a user’s account. This certificate can come from any source, as long as the root CA for that certificate is trusted for client authentication. The client certificate can be renewed, etc without affecting this process as long as the subject and issuer name fields do not change.

For example, suppose that users obtain certificates from a CA that is approved by your company. You then take these user certificates and map them to the Active Directory user accounts. This allows users to connect to the server, using TLS from anywhere by providing their client certificate.

Schannel builds an "altSecurityId" string from the client’s certificate’s subject and issuer name fields, and queries Active Directory for a user account with a matching **altSecurityId** property.

Many-to-One Mapping

Many-to-one mapping involves mapping many certificates to a single user account. Many-to-one mapping works the same way as one-to-one mapping, but it maps all client certificates issued by a given CA to the same user

account. This certificate can come from any source, as long as the root CA for that certificate is trusted for client authentication.

Schannel builds an "altSecurityId" string from the client's certificate's issuer name field, and queries Active Directory for a user account with a matching **altSecurityId** property.

Back to Top

Network Ports Used by TLS/SSL

Port Assignments for Common Applications over TLS/SSL

Service Name	TCP
smtp	25
https	443
nntps	563
ldaps	636
ftps-data	989
ftps	990
telnets	992
imaps	993
pop3s	995
ms-sql-s	1433
mfst-gc-ssl	3269
tftps	3713

Back to Top

Related Information

- RFC 2246 in the IETF RFC Database
- RFC 3546 in the IETF RFC Database
- RFC 2817 in the IETF RFC Database