

pre_master_secret

This random value is generated by the client and is used to generate the master secret, as specified in [Section 8.1](#).

7.4.7.2. Client Diffie-Hellman public value

Meaning of this message:

This structure conveys the client's Diffie-Hellman public value (Yc) if it was not already included in the client's certificate. The encoding used for Yc is determined by the enumerated PublicValueEncoding. This structure is a variant of the client key exchange message, not a message in itself.

Structure of this message:

```
enum { implicit, explicit } PublicValueEncoding;
```

implicit

If the client certificate already contains a suitable Diffie-Hellman key, then Yc is implicit and does not need to be sent again. In this case, the Client Key Exchange message will be sent, but will be empty.

explicit

Yc needs to be sent.

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

dh_Yc

The client's Diffie-Hellman public value (Yc).

7.4.8. Certificate verify

When this message will be sent:

This message is used to provide explicit verification of a client certificate. This message is only sent following a client certificate that has signing capability (i.e. all certificates except those containing fixed Diffie-Hellman parameters). When sent, it will immediately follow the client key exchange message.

Structure of this message:

```
struct {
    Signature signature;
} CertificateVerify;
```

The Signature type is defined in 7.4.3.

```
CertificateVerify.signature.md5_hash
    MD5(handshake_messages);
```

```
Certificate.signature.sha_hash
    SHA(handshake_messages);
```

Here handshake_messages refers to all handshake messages sent or received starting at client hello up to but not including this message, including the type and length fields of the handshake messages. This is the concatenation of all the Handshake structures as defined in 7.4 exchanged thus far.

7.4.9. Finished

When this message will be sent:

A finished message is always sent immediately after a change cipher spec message to verify that the key exchange and authentication processes were successful. It is essential that a change cipher spec message be received between the other handshake messages and the Finished message.

Meaning of this message:

The finished message is the first protected with the just-negotiated algorithms, keys, and secrets. Recipients of finished messages must verify that the contents are correct. Once a side has sent its Finished message and received and validated the Finished message from its peer, it may begin to send and receive application data over the connection.

```
struct {
    opaque verify_data[12];
} Finished;
```

```
verify_data
    PRF(master_secret, finished_label, MD5(handshake_messages) +
    SHA-1(handshake_messages)) [0..11];
```

```
finished_label
    For Finished messages sent by the client, the string "client
    finished". For Finished messages sent by the server, the
    string "server finished".
```

```
handshake_messages
    All of the data from all handshake messages up to but not
    including this message. This is only data visible at the
    handshake layer and does not include record layer headers.
```

This is the concatenation of all the Handshake structures as defined in 7.4 exchanged thus far.

It is a fatal error if a finished message is not preceded by a change cipher spec message at the appropriate point in the handshake.

The hash contained in finished messages sent by the server incorporate Sender.server; those sent by the client incorporate Sender.client. The value handshake_messages includes all handshake messages starting at client hello up to, but not including, this finished message. This may be different from handshake_messages in [Section 7.4.8](#) because it would include the certificate verify message (if sent). Also, the handshake_messages for the finished message sent by the client will be different from that for the finished message sent by the server, because the one which is sent second will include the prior one.

Note: Change cipher spec messages, alerts and any other record types are not handshake messages and are not included in the hash computations. Also, Hello Request messages are omitted from handshake hashes.

8. Cryptographic computations

In order to begin connection protection, the TLS Record Protocol requires specification of a suite of algorithms, a master secret, and the client and server random values. The authentication, encryption, and MAC algorithms are determined by the cipher_suite selected by the server and revealed in the server hello message. The compression algorithm is negotiated in the hello messages, and the random values are exchanged in the hello messages. All that remains is to calculate the master secret.

8.1. Computing the master secret

For all key exchange methods, the same algorithm is used to convert the pre_master_secret into the master_secret. The pre_master_secret should be deleted from memory once the master_secret has been computed.

```
master_secret = PRF(pre_master_secret, "master secret",
                    ClientHello.random + ServerHello.random)
[0..47];
```

The master secret is always exactly 48 bytes in length. The length of the premaster secret will vary depending on key exchange method.

8.1.1. RSA

When RSA is used for server authentication and key exchange, a 48-byte `pre_master_secret` is generated by the client, encrypted under the server's public key, and sent to the server. The server uses its private key to decrypt the `pre_master_secret`. Both parties then convert the `pre_master_secret` into the `master_secret`, as specified above.

RSA digital signatures are performed using PKCS #1 [[PKCS1](#)] block type 1. RSA public key encryption is performed using PKCS #1 block type 2.

8.1.2. Diffie-Hellman

A conventional Diffie-Hellman computation is performed. The negotiated key (`Z`) is used as the `pre_master_secret`, and is converted into the `master_secret`, as specified above.

Note: Diffie-Hellman parameters are specified by the server, and may be either ephemeral or contained within the server's certificate.

9. Mandatory Cipher Suites

In the absence of an application profile standard specifying otherwise, a TLS compliant application MUST implement the cipher suite `TLS_DHE_DSS_WITH_3DES_EDE_CBC_SHA`.

10. Application data protocol

Application data messages are carried by the Record Layer and are fragmented, compressed and encrypted based on the current connection state. The messages are treated as transparent data to the record layer.

A. Protocol constant values

This section describes protocol types and constants.

A.1. Record layer

```

struct {
    uint8 major, minor;
} ProtocolVersion;

ProtocolVersion version = { 3, 1 };    /* TLS v1.0 */

enum {
    change_cipher_spec(20), alert(21), handshake(22),
    application_data(23), (255)
} ContentType;

struct {
    ContentType type;
    ProtocolVersion version;
    uint16 length;
    opaque fragment[TLSPlaintext.length];
} TLSPlaintext;

struct {
    ContentType type;
    ProtocolVersion version;
    uint16 length;
    opaque fragment[TLSCompressed.length];
} TLSCompressed;

struct {
    ContentType type;
    ProtocolVersion version;
    uint16 length;
    select (CipherSpec.cipher_type) {
        case stream: GenericStreamCipher;
        case block:  GenericBlockCipher;
    } fragment;
} TLSCiphertext;

stream-ciphered struct {
    opaque content[TLSCompressed.length];
    opaque MAC[CipherSpec.hash_size];
} GenericStreamCipher;

block-ciphered struct {
    opaque content[TLSCompressed.length];

```

```
    opaque MAC[CipherSpec.hash_size];
    uint8 padding[GenericBlockCipher.padding_length];
    uint8 padding_length;
} GenericBlockCipher;
```

A.2. Change cipher specs message

```
struct {
    enum { change_cipher_spec(1), (255) } type;
} ChangeCipherSpec;
```

A.3. Alert messages

```
enum { warning(1), fatal(2), (255) } AlertLevel;
```

```
enum {
    close_notify(0),
    unexpected_message(10),
    bad_record_mac(20),
    decryption_failed(21),
    record_overflow(22),
    decompression_failure(30),
    handshake_failure(40),
    bad_certificate(42),
    unsupported_certificate(43),
    certificate_revoked(44),
    certificate_expired(45),
    certificate_unknown(46),
    illegal_parameter(47),
    unknown_ca(48),
    access_denied(49),
    decode_error(50),
    decrypt_error(51),
    export_restriction(60),
    protocol_version(70),
    insufficient_security(71),
    internal_error(80),
    user_canceled(90),
    no_renegotiation(100),
    (255)
} AlertDescription;
```

```
struct {
    AlertLevel level;
    AlertDescription description;
} Alert;
```

A.4. Handshake protocol

```

enum {
    hello_request(0), client_hello(1), server_hello(2),
    certificate(11), server_key_exchange (12),
    certificate_request(13), server_hello_done(14),
    certificate_verify(15), client_key_exchange(16),
    finished(20), (255)
} HandshakeType;

struct {
    HandshakeType msg_type;
    uint24 length;
    select (HandshakeType) {
        case hello_request:      HelloRequest;
        case client_hello:       ClientHello;
        case server_hello:       ServerHello;
        case certificate:        Certificate;
        case server_key_exchange: ServerKeyExchange;
        case certificate_request: CertificateRequest;
        case server_hello_done:  ServerHelloDone;
        case certificate_verify:  CertificateVerify;
        case client_key_exchange: ClientKeyExchange;
        case finished:           Finished;
    } body;
} Handshake;

```

A.4.1. Hello messages

```

struct { } HelloRequest;

struct {
    uint32  gmt_unix_time;
    opaque  random_bytes[28];
} Random;

opaque SessionID<0..32>;

uint8 CipherSuite[2];

enum { null(0), (255) } CompressionMethod;

struct {
    ProtocolVersion client_version;
    Random random;
    SessionID session_id;
    CipherSuite cipher_suites<2..2^16-1>;
    CompressionMethod compression_methods<1..2^8-1>;
}

```

```

} ClientHello;

struct {
    ProtocolVersion server_version;
    Random random;
    SessionID session_id;
    CipherSuite cipher_suite;
    CompressionMethod compression_method;
} ServerHello;

```

A.4.2. Server authentication and key exchange messages

```

opaque ASN.1Cert<2^24-1>;

struct {
    ASN.1Cert certificate_list<1..2^24-1>;
} Certificate;

enum { rsa, diffie_hellman } KeyExchangeAlgorithm;

struct {
    opaque RSA_modulus<1..2^16-1>;
    opaque RSA_exponent<1..2^16-1>;
} ServerRSAParams;

struct {
    opaque DH_p<1..2^16-1>;
    opaque DH_g<1..2^16-1>;
    opaque DH_Ys<1..2^16-1>;
} ServerDHPParams;

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

enum { anonymous, rsa, dsa } SignatureAlgorithm;

select (SignatureAlgorithm)
{
    case anonymous: struct { };
    case rsa:
        digitally-signed struct {

```

```

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

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

opaque DistinguishedName<1..2^16-1>;

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

struct { } ServerHelloDone;

```

A.4.3. Client authentication and key exchange messages

```

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

struct {
    ProtocolVersion client_version;
    opaque random[46];
} PreMasterSecret;

struct {
    public-key-encrypted PreMasterSecret pre_master_secret;
} EncryptedPreMasterSecret;

enum { implicit, explicit } PublicValueEncoding;

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

```

```

    } dh_public;
} ClientDiffieHellmanPublic;

struct {
    Signature signature;
} CertificateVerify;

```

A.4.4. Handshake finalization message

```

struct {
    opaque verify_data[12];
} Finished;

```

A.5. The CipherSuite

The following values define the CipherSuite codes used in the client hello and server hello messages.

A CipherSuite defines a cipher specification supported in TLS Version 1.0.

TLS_NULL_WITH_NULL_NULL is specified and is the initial state of a TLS connection during the first handshake on that channel, but must not be negotiated, as it provides no more protection than an unsecured connection.

```
CipherSuite TLS_NULL_WITH_NULL_NULL = { 0x00,0x00 };
```

The following CipherSuite definitions require that the server provide an RSA certificate that can be used for key exchange. The server may request either an RSA or a DSS signature-capable certificate in the certificate request message.

```

CipherSuite TLS_RSA_WITH_NULL_MD5           = { 0x00,0x01 };
CipherSuite TLS_RSA_WITH_NULL_SHA           = { 0x00,0x02 };
CipherSuite TLS_RSA_EXPORT_WITH_RC4_40_MD5 = { 0x00,0x03 };
CipherSuite TLS_RSA_WITH_RC4_128_MD5       = { 0x00,0x04 };
CipherSuite TLS_RSA_WITH_RC4_128_SHA       = { 0x00,0x05 };
CipherSuite TLS_RSA_EXPORT_WITH_RC2_CBC_40_MD5 = { 0x00,0x06 };
CipherSuite TLS_RSA_WITH_IDEA_CBC_SHA      = { 0x00,0x07 };
CipherSuite TLS_RSA_EXPORT_WITH_DES40_CBC_SHA = { 0x00,0x08 };
CipherSuite TLS_RSA_WITH_DES_CBC_SHA       = { 0x00,0x09 };
CipherSuite TLS_RSA_WITH_3DES_EDE_CBC_SHA  = { 0x00,0x0A };

```

The following CipherSuite definitions are used for server-authenticated (and optionally client-authenticated) Diffie-Hellman. DH denotes cipher suites in which the server's certificate contains the Diffie-Hellman parameters signed by the certificate authority

(CA). DHE denotes ephemeral Diffie-Hellman, where the Diffie-Hellman parameters are signed by a DSS or RSA certificate, which has been signed by the CA. The signing algorithm used is specified after the DH or DHE parameter. The server can request an RSA or DSS signature-capable certificate from the client for client authentication or it may request a Diffie-Hellman certificate. Any Diffie-Hellman certificate provided by the client must use the parameters (group and generator) described by the server.

```
CipherSuite TLS_DH_DSS_EXPORT_WITH_DES40_CBC_SHA = { 0x00,0x0B };
CipherSuite TLS_DH_DSS_WITH_DES_CBC_SHA          = { 0x00,0x0C };
CipherSuite TLS_DH_DSS_WITH_3DES_EDE_CBC_SHA      = { 0x00,0x0D };
CipherSuite TLS_DH_RSA_EXPORT_WITH_DES40_CBC_SHA  = { 0x00,0x0E };
CipherSuite TLS_DH_RSA_WITH_DES_CBC_SHA           = { 0x00,0x0F };
CipherSuite TLS_DH_RSA_WITH_3DES_EDE_CBC_SHA      = { 0x00,0x10 };
CipherSuite TLS_DHE_DSS_EXPORT_WITH_DES40_CBC_SHA = { 0x00,0x11 };
CipherSuite TLS_DHE_DSS_WITH_DES_CBC_SHA          = { 0x00,0x12 };
CipherSuite TLS_DHE_DSS_WITH_3DES_EDE_CBC_SHA     = { 0x00,0x13 };
CipherSuite TLS_DHE_RSA_EXPORT_WITH_DES40_CBC_SHA = { 0x00,0x14 };
CipherSuite TLS_DHE_RSA_WITH_DES_CBC_SHA          = { 0x00,0x15 };
CipherSuite TLS_DHE_RSA_WITH_3DES_EDE_CBC_SHA     = { 0x00,0x16 };
```

The following cipher suites are used for completely anonymous Diffie-Hellman communications in which neither party is authenticated. Note that this mode is vulnerable to man-in-the-middle attacks and is therefore deprecated.

```
CipherSuite TLS_DH_anon_EXPORT_WITH_RC4_40_MD5    = { 0x00,0x17 };
CipherSuite TLS_DH_anon_WITH_RC4_128_MD5         = { 0x00,0x18 };
CipherSuite TLS_DH_anon_EXPORT_WITH_DES40_CBC_SHA = { 0x00,0x19 };
CipherSuite TLS_DH_anon_WITH_DES_CBC_SHA          = { 0x00,0x1A };
CipherSuite TLS_DH_anon_WITH_3DES_EDE_CBC_SHA     = { 0x00,0x1B };
```

Note: All cipher suites whose first byte is 0xFF are considered private and can be used for defining local/experimental algorithms. Interoperability of such types is a local matter.

Note: Additional cipher suites can be registered by publishing an RFC which specifies the cipher suites, including the necessary TLS protocol information, including message encoding, premaster secret derivation, symmetric encryption and MAC calculation and appropriate reference information for the algorithms involved. The RFC editor's office may, at its discretion, choose to publish specifications for cipher suites which are not completely described (e.g., for classified algorithms) if it finds the specification to be of technical interest and completely specified.

Note: The cipher suite values { 0x00, 0x1C } and { 0x00, 0x1D } are reserved to avoid collision with Fortezza-based cipher suites in SSL 3.

A.6. The Security Parameters

These security parameters are determined by the TLS Handshake Protocol and provided as parameters to the TLS Record Layer in order to initialize a connection state. SecurityParameters includes:

```
enum { null(0), (255) } CompressionMethod;

enum { server, client } ConnectionEnd;

enum { null, rc4, rc2, des, 3des, des40, idea }
BulkCipherAlgorithm;

enum { stream, block } CipherType;

enum { true, false } IsExportable;

enum { null, md5, sha } MACAlgorithm;

/* The algorithms specified in CompressionMethod,
BulkCipherAlgorithm, and MACAlgorithm may be added to. */

struct {
    ConnectionEnd entity;
    BulkCipherAlgorithm bulk_cipher_algorithm;
    CipherType cipher_type;
    uint8 key_size;
    uint8 key_material_length;
    IsExportable is_exportable;
    MACAlgorithm mac_algorithm;
    uint8 hash_size;
    CompressionMethod compression_algorithm;
    opaque master_secret[48];
    opaque client_random[32];
    opaque server_random[32];
} SecurityParameters;
```

B. Glossary**application protocol**

An application protocol is a protocol that normally layers directly on top of the transport layer (e.g., TCP/IP). Examples include HTTP, TELNET, FTP, and SMTP.

asymmetric cipher

See public key cryptography.

authentication

Authentication is the ability of one entity to determine the identity of another entity.

block cipher

A block cipher is an algorithm that operates on plaintext in groups of bits, called blocks. 64 bits is a common block size.

bulk cipher

A symmetric encryption algorithm used to encrypt large quantities of data.

cipher block chaining (CBC)

CBC is a mode in which every plaintext block encrypted with a block cipher is first exclusive-ORed with the previous ciphertext block (or, in the case of the first block, with the initialization vector). For decryption, every block is first decrypted, then exclusive-ORed with the previous ciphertext block (or IV).

certificate

As part of the X.509 protocol (a.k.a. ISO Authentication framework), certificates are assigned by a trusted Certificate Authority and provide a strong binding between a party's identity or some other attributes and its public key.

client

The application entity that initiates a TLS connection to a server. This may or may not imply that the client initiated the underlying transport connection. The primary operational difference between the server and client is that the server is generally authenticated, while the client is only optionally authenticated.

client write key

The key used to encrypt data written by the client.

client write MAC secret

The secret data used to authenticate data written by the client.

connection

A connection is a transport (in the OSI layering model definition) that provides a suitable type of service. For TLS, such connections are peer to peer relationships. The connections are transient. Every connection is associated with one session.

Data Encryption Standard

DES is a very widely used symmetric encryption algorithm. DES is a block cipher with a 56 bit key and an 8 byte block size. Note that in TLS, for key generation purposes, DES is treated as having an 8 byte key length (64 bits), but it still only provides 56 bits of protection. (The low bit of each key byte is presumed to be set to produce odd parity in that key byte.) DES can also be operated in a mode where three independent keys and three encryptions are used for each block of data; this uses 168 bits of key (24 bytes in the TLS key generation method) and provides the equivalent of 112 bits of security. [DES], [3DES]

Digital Signature Standard (DSS)

A standard for digital signing, including the Digital Signing Algorithm, approved by the National Institute of Standards and Technology, defined in NIST FIPS PUB 186, "Digital Signature Standard," published May, 1994 by the U.S. Dept. of Commerce. [DSS]

digital signatures

Digital signatures utilize public key cryptography and one-way hash functions to produce a signature of the data that can be authenticated, and is difficult to forge or repudiate.

handshake

An initial negotiation between client and server that establishes the parameters of their transactions.

Initialization Vector (IV)

When a block cipher is used in CBC mode, the initialization vector is exclusive-ORed with the first plaintext block prior to encryption.

IDEA

A 64-bit block cipher designed by Xuejia Lai and James Massey. [IDEA]

Message Authentication Code (MAC)

A Message Authentication Code is a one-way hash computed from a message and some secret data. It is difficult to forge without knowing the secret data. Its purpose is to detect if the message has been altered.

master secret

Secure secret data used for generating encryption keys, MAC secrets, and IVs.

MD5

MD5 is a secure hashing function that converts an arbitrarily long data stream into a digest of fixed size (16 bytes). [[MD5](#)]

public key cryptography

A class of cryptographic techniques employing two-key ciphers. Messages encrypted with the public key can only be decrypted with the associated private key. Conversely, messages signed with the private key can be verified with the public key.

one-way hash function

A one-way transformation that converts an arbitrary amount of data into a fixed-length hash. It is computationally hard to reverse the transformation or to find collisions. MD5 and SHA are examples of one-way hash functions.

RC2

A block cipher developed by Ron Rivest at RSA Data Security, Inc. [[RSADSI](#)] described in [[RC2](#)].

RC4

A stream cipher licensed by RSA Data Security [[RSADSI](#)]. A compatible cipher is described in [[RC4](#)].

RSA

A very widely used public-key algorithm that can be used for either encryption or digital signing. [[RSA](#)]

salt

Non-secret random data used to make export encryption keys resist precomputation attacks.

server

The server is the application entity that responds to requests for connections from clients. See also under client.

session

A TLS session is an association between a client and a server. Sessions are created by the handshake protocol. Sessions define a set of cryptographic security parameters, which can be shared among multiple connections. Sessions are used to avoid the expensive negotiation of new security parameters for each connection.

session identifier

A session identifier is a value generated by a server that identifies a particular session.

server write key

The key used to encrypt data written by the server.

server write MAC secret

The secret data used to authenticate data written by the server.

SHA

The Secure Hash Algorithm is defined in FIPS PUB 180-1. It produces a 20-byte output. Note that all references to SHA actually use the modified SHA-1 algorithm. [[SHA](#)]

SSL

Netscape's Secure Socket Layer protocol [[SSL3](#)]. TLS is based on SSL Version 3.0

stream cipher

An encryption algorithm that converts a key into a cryptographically-strong keystream, which is then exclusive-ORed with the plaintext.

symmetric cipher

See bulk cipher.

Transport Layer Security (TLS)

This protocol; also, the Transport Layer Security working group of the Internet Engineering Task Force (IETF). See "Comments" at the end of this document.

C. CipherSuite definitions

CipherSuite	Is Exportable	Key Exchange	Cipher	Hash
TLS_NULL_WITH_NULL_NULL		* NULL	NULL	NULL
TLS_RSA_WITH_NULL_MD5		* RSA	NULL	MD5
TLS_RSA_WITH_NULL_SHA		* RSA	NULL	SHA
TLS_RSA_EXPORT_WITH_RC4_40_MD5		* RSA_EXPORT	RC4_40	MD5
TLS_RSA_WITH_RC4_128_MD5		RSA	RC4_128	MD5
TLS_RSA_WITH_RC4_128_SHA		RSA	RC4_128	SHA
TLS_RSA_EXPORT_WITH_RC2_CBC_40_MD5		* RSA_EXPORT	RC2_CBC_40	MD5
TLS_RSA_WITH_IDEA_CBC_SHA		RSA	IDEA_CBC	SHA
TLS_RSA_EXPORT_WITH_DES40_CBC_SHA		* RSA_EXPORT	DES40_CBC	SHA
TLS_RSA_WITH_DES_CBC_SHA		RSA	DES_CBC	SHA
TLS_RSA_WITH_3DES_EDE_CBC_SHA		RSA	3DES_EDE_CBC	SHA
TLS_DH_DSS_EXPORT_WITH_DES40_CBC_SHA		* DH_DSS_EXPORT	DES40_CBC	SHA
TLS_DH_DSS_WITH_DES_CBC_SHA		DH_DSS	DES_CBC	SHA
TLS_DH_DSS_WITH_3DES_EDE_CBC_SHA		DH_DSS	3DES_EDE_CBC	SHA
TLS_DH_RSA_EXPORT_WITH_DES40_CBC_SHA		* DH_RSA_EXPORT	DES40_CBC	SHA
TLS_DH_RSA_WITH_DES_CBC_SHA		DH_RSA	DES_CBC	SHA
TLS_DH_RSA_WITH_3DES_EDE_CBC_SHA		DH_RSA	3DES_EDE_CBC	SHA
TLS_DHE_DSS_EXPORT_WITH_DES40_CBC_SHA		* DHE_DSS_EXPORT	DES40_CBC	SHA
TLS_DHE_DSS_WITH_DES_CBC_SHA		DHE_DSS	DES_CBC	SHA
TLS_DHE_DSS_WITH_3DES_EDE_CBC_SHA		DHE_DSS	3DES_EDE_CBC	SHA
TLS_DHE_RSA_EXPORT_WITH_DES40_CBC_SHA		* DHE_RSA_EXPORT	DES40_CBC	SHA
TLS_DHE_RSA_WITH_DES_CBC_SHA		DHE_RSA	DES_CBC	SHA
TLS_DHE_RSA_WITH_3DES_EDE_CBC_SHA		DHE_RSA	3DES_EDE_CBC	SHA
TLS_DH_anon_EXPORT_WITH_RC4_40_MD5		* DH_anon_EXPORT	RC4_40	MD5
TLS_DH_anon_WITH_RC4_128_MD5		DH_anon	RC4_128	MD5
TLS_DH_anon_EXPORT_WITH_DES40_CBC_SHA		DH_anon	DES40_CBC	SHA
TLS_DH_anon_WITH_DES_CBC_SHA		DH_anon	DES_CBC	SHA
TLS_DH_anon_WITH_3DES_EDE_CBC_SHA		DH_anon	3DES_EDE_CBC	SHA

* Indicates IsExportable is True

Key Exchange Algorithm	Description	Key size limit
DHE_DSS	Ephemeral DH with DSS signatures	None
DHE_DSS_EXPORT	Ephemeral DH with DSS signatures	DH = 512 bits
DHE_RSA	Ephemeral DH with RSA signatures	None
DHE_RSA_EXPORT	Ephemeral DH with RSA signatures	DH = 512 bits, RSA = none
DH_anon	Anonymous DH, no signatures	None
DH_anon_EXPORT	Anonymous DH, no signatures	DH = 512 bits

RFC 2246

The TLS Protocol Version 1.0

January 1999

DH_DSS	DH with DSS-based certificates	None
DH_DSS_EXPORT	DH with DSS-based certificates	DH = 512 bits
DH_RSA	DH with RSA-based certificates	None
DH_RSA_EXPORT	DH with RSA-based certificates	DH = 512 bits, RSA = none
NULL	No key exchange	N/A
RSA	RSA key exchange	None
RSA_EXPORT	RSA key exchange	RSA = 512 bits

Key size limit

The key size limit gives the size of the largest public key that can be legally used for encryption in cipher suites that are exportable.

Cipher	Type	Key Material	Expanded Key Material	Effective Key Bits	IV Size	Block Size
NULL	* Stream	0	0	0	0	N/A
IDEA_CBC	Block	16	16	128	8	8
RC2_CBC_40	* Block	5	16	40	8	8
RC4_40	* Stream	5	16	40	0	N/A
RC4_128	Stream	16	16	128	0	N/A
DES40_CBC	* Block	5	8	40	8	8
DES_CBC	Block	8	8	56	8	8
3DES_EDE_CBC	Block	24	24	168	8	8

* Indicates IsExportable is true.

Type

Indicates whether this is a stream cipher or a block cipher running in CBC mode.

Key Material

The number of bytes from the key_block that are used for generating the write keys.

Expanded Key Material

The number of bytes actually fed into the encryption algorithm

Effective Key Bits

How much entropy material is in the key material being fed into the encryption routines.

IV Size

How much data needs to be generated for the initialization vector. Zero for stream ciphers; equal to the block size for block ciphers.

Block Size

The amount of data a block cipher enciphers in one chunk; a block cipher running in CBC mode can only encrypt an even multiple of its block size.

Hash function	Hash Size	Padding Size
NULL	0	0
MD5	16	48
SHA	20	40

D. Implementation Notes

The TLS protocol cannot prevent many common security mistakes. This section provides several recommendations to assist implementors.

D.1. Temporary RSA keys

US Export restrictions limit RSA keys used for encryption to 512 bits, but do not place any limit on lengths of RSA keys used for signing operations. Certificates often need to be larger than 512 bits, since 512-bit RSA keys are not secure enough for high-value transactions or for applications requiring long-term security. Some certificates are also designated signing-only, in which case they cannot be used for key exchange.

When the public key in the certificate cannot be used for encryption, the server signs a temporary RSA key, which is then exchanged. In exportable applications, the temporary RSA key should be the maximum allowable length (i.e., 512 bits). Because 512-bit RSA keys are relatively insecure, they should be changed often. For typical electronic commerce applications, it is suggested that keys be changed daily or every 500 transactions, and more often if possible. Note that while it is acceptable to use the same temporary key for multiple transactions, it must be signed each time it is used.

RSA key generation is a time-consuming process. In many cases, a low-priority process can be assigned the task of key generation.

Whenever a new key is completed, the existing temporary key can be replaced with the new one.

D.2. Random Number Generation and Seeding

TLS requires a cryptographically-secure pseudorandom number generator (PRNG). Care must be taken in designing and seeding PRNGs. PRNGs based on secure hash operations, most notably MD5 and/or SHA, are acceptable, but cannot provide more security than the size of the random number generator state. (For example, MD5-based PRNGs usually provide 128 bits of state.)

To estimate the amount of seed material being produced, add the number of bits of unpredictable information in each seed byte. For example, keystroke timing values taken from a PC compatible's 18.2 Hz timer provide 1 or 2 secure bits each, even though the total size of the counter value is 16 bits or more. To seed a 128-bit PRNG, one would thus require approximately 100 such timer values.

Warning: The seeding functions in RSAREF and versions of BSAFE prior to 3.0 are order-independent. For example, if 1000 seed bits are supplied, one at a time, in 1000 separate calls to the seed function, the PRNG will end up in a state which depends only on the number of 0 or 1 seed bits in the seed data (i.e., there are 1001 possible final states). Applications using BSAFE or RSAREF must take extra care to ensure proper seeding. This may be accomplished by accumulating seed bits into a buffer and processing them all at once or by processing an incrementing counter with every seed bit; either method will reintroduce order dependence into the seeding process.

D.3. Certificates and authentication

Implementations are responsible for verifying the integrity of certificates and should generally support certificate revocation messages. Certificates should always be verified to ensure proper signing by a trusted Certificate Authority (CA). The selection and addition of trusted CAs should be done very carefully. Users should be able to view information about the certificate and root CA.

D.4. CipherSuites

TLS supports a range of key sizes and security levels, including some which provide no or minimal security. A proper implementation will probably not support many cipher suites. For example, 40-bit encryption is easily broken, so implementations requiring strong security should not allow 40-bit keys. Similarly, anonymous Diffie-Hellman is strongly discouraged because it cannot prevent man-in-the-middle attacks. Applications should also enforce minimum and maximum key sizes. For example, certificate chains containing 512-bit RSA keys or signatures are not appropriate for high-security applications.

E. Backward Compatibility With SSL

For historical reasons and in order to avoid a profligate consumption of reserved port numbers, application protocols which are secured by TLS 1.0, SSL 3.0, and SSL 2.0 all frequently share the same connection port: for example, the https protocol (HTTP secured by SSL or TLS) uses port 443 regardless of which security protocol it is using. Thus, some mechanism must be determined to distinguish and negotiate among the various protocols.

TLS version 1.0 and SSL 3.0 are very similar; thus, supporting both is easy. TLS clients who wish to negotiate with SSL 3.0 servers should send client hello messages using the SSL 3.0 record format and client hello structure, sending {3, 1} for the version field to note that they support TLS 1.0. If the server supports only SSL 3.0, it will respond with an SSL 3.0 server hello; if it supports TLS, with a TLS server hello. The negotiation then proceeds as appropriate for the negotiated protocol.

Similarly, a TLS server which wishes to interoperate with SSL 3.0 clients should accept SSL 3.0 client hello messages and respond with an SSL 3.0 server hello if an SSL 3.0 client hello is received which has a version field of {3, 0}, denoting that this client does not support TLS.

Whenever a client already knows the highest protocol known to a server (for example, when resuming a session), it should initiate the connection in that native protocol.

TLS 1.0 clients that support SSL Version 2.0 servers must send SSL Version 2.0 client hello messages [SSL2]. TLS servers should accept either client hello format if they wish to support SSL 2.0 clients on the same connection port. The only deviations from the Version 2.0 specification are the ability to specify a version with a value of three and the support for more ciphering types in the CipherSpec.

Warning: The ability to send Version 2.0 client hello messages will be phased out with all due haste. Implementors should make every effort to move forward as quickly as possible. Version 3.0 provides better mechanisms for moving to newer versions.

The following cipher specifications are carryovers from SSL Version 2.0. These are assumed to use RSA for key exchange and authentication.

```
V2CipherSpec TLS_RC4_128_WITH_MD5           = { 0x01,0x00,0x80 } ;
V2CipherSpec TLS_RC4_128_EXPORT40_WITH_MD5 = { 0x02,0x00,0x80 } ;
V2CipherSpec TLS_RC2_CBC_128_CBC_WITH_MD5   = { 0x03,0x00,0x80 } ;
```

```

V2CipherSpec TLS_RC2_CBC_128_CBC_EXPORT40_WITH_MD5
                                     = { 0x04,0x00,0x80 };
V2CipherSpec TLS_IDEA_128_CBC_WITH_MD5    = { 0x05,0x00,0x80 };
V2CipherSpec TLS_DES_64_CBC_WITH_MD5      = { 0x06,0x00,0x40 };
V2CipherSpec TLS_DES_192_EDE3_CBC_WITH_MD5 = { 0x07,0x00,0xC0 };

```

Cipher specifications native to TLS can be included in Version 2.0 client hello messages using the syntax below. Any V2CipherSpec element with its first byte equal to zero will be ignored by Version 2.0 servers. Clients sending any of the above V2CipherSpecs should also include the TLS equivalent (see [Appendix A.5](#)):

```
V2CipherSpec (see TLS name) = { 0x00, CipherSuite };
```

E.1. Version 2 client hello

The Version 2.0 client hello message is presented below using this document's presentation model. The true definition is still assumed to be the SSL Version 2.0 specification.

```

uint8 V2CipherSpec[3];

struct {
    uint8 msg_type;
    Version version;
    uint16 cipher_spec_length;
    uint16 session_id_length;
    uint16 challenge_length;
    V2CipherSpec cipher_specs[V2ClientHello.cipher_spec_length];
    opaque session_id[V2ClientHello.session_id_length];
    Random challenge;
} V2ClientHello;

```

msg_type

This field, in conjunction with the version field, identifies a version 2 client hello message. The value should be one (1).

version

The highest version of the protocol supported by the client (equals ProtocolVersion.version, see [Appendix A.1](#)).

cipher_spec_length

This field is the total length of the field cipher_specs. It cannot be zero and must be a multiple of the V2CipherSpec length (3).

session_id_length

This field must have a value of either zero or 16. If zero, the client is creating a new session. If 16, the session_id field will contain the 16 bytes of session identification.

challenge_length

The length in bytes of the client's challenge to the server to authenticate itself. This value must be 32.

cipher_specs

This is a list of all CipherSpecs the client is willing and able to use. There must be at least one CipherSpec acceptable to the server.

session_id

If this field's length is not zero, it will contain the identification for a session that the client wishes to resume.

challenge

The client challenge to the server for the server to identify itself is a (nearly) arbitrary length random. The TLS server will right justify the challenge data to become the ClientHello.random data (padded with leading zeroes, if necessary), as specified in this protocol specification. If the length of the challenge is greater than 32 bytes, only the last 32 bytes are used. It is legitimate (but not necessary) for a V3 server to reject a V2 ClientHello that has fewer than 16 bytes of challenge data.

Note: Requests to resume a TLS session should use a TLS client hello.

E.2. Avoiding man-in-the-middle version rollback

When TLS clients fall back to Version 2.0 compatibility mode, they should use special PKCS #1 block formatting. This is done so that TLS servers will reject Version 2.0 sessions with TLS-capable clients.

When TLS clients are in Version 2.0 compatibility mode, they set the right-hand (least-significant) 8 random bytes of the PKCS padding (not including the terminal null of the padding) for the RSA encryption of the ENCRYPTED-KEY-DATA field of the CLIENT-MASTER-KEY to 0x03 (the other padding bytes are random). After decrypting the ENCRYPTED-KEY-DATA field, servers that support TLS should issue an error if these eight padding bytes are 0x03. Version 2.0 servers receiving blocks padded in this manner will proceed normally.

F. Security analysis

The TLS protocol is designed to establish a secure connection between a client and a server communicating over an insecure channel. This document makes several traditional assumptions, including that attackers have substantial computational resources and cannot obtain secret information from sources outside the protocol. Attackers are assumed to have the ability to capture, modify, delete, replay, and otherwise tamper with messages sent over the communication channel. This appendix outlines how TLS has been designed to resist a variety of attacks.

F.1. Handshake protocol

The handshake protocol is responsible for selecting a CipherSpec and generating a Master Secret, which together comprise the primary cryptographic parameters associated with a secure session. The handshake protocol can also optionally authenticate parties who have certificates signed by a trusted certificate authority.

F.1.1. Authentication and key exchange

TLS supports three authentication modes: authentication of both parties, server authentication with an unauthenticated client, and total anonymity. Whenever the server is authenticated, the channel is secure against man-in-the-middle attacks, but completely anonymous sessions are inherently vulnerable to such attacks. Anonymous servers cannot authenticate clients. If the server is authenticated, its certificate message must provide a valid certificate chain leading to an acceptable certificate authority. Similarly, authenticated clients must supply an acceptable certificate to the server. Each party is responsible for verifying that the other's certificate is valid and has not expired or been revoked.

The general goal of the key exchange process is to create a `pre_master_secret` known to the communicating parties and not to attackers. The `pre_master_secret` will be used to generate the `master_secret` (see [Section 8.1](#)). The `master_secret` is required to generate the certificate verify and finished messages, encryption keys, and MAC secrets (see [Sections 7.4.8](#), [7.4.9](#) and [6.3](#)). By sending a correct finished message, parties thus prove that they know the correct `pre_master_secret`.

F.1.1.1. Anonymous key exchange

Completely anonymous sessions can be established using RSA or Diffie-Hellman for key exchange. With anonymous RSA, the client encrypts a `pre_master_secret` with the server's uncertified public key

extracted from the server key exchange message. The result is sent in a client key exchange message. Since eavesdroppers do not know the server's private key, it will be infeasible for them to decode the `pre_master_secret`. (Note that no anonymous RSA Cipher Suites are defined in this document).

With Diffie-Hellman, the server's public parameters are contained in the server key exchange message and the client's are sent in the client key exchange message. Eavesdroppers who do not know the private values should not be able to find the Diffie-Hellman result (i.e. the `pre_master_secret`).

Warning: Completely anonymous connections only provide protection against passive eavesdropping. Unless an independent tamper-proof channel is used to verify that the finished messages were not replaced by an attacker, server authentication is required in environments where active man-in-the-middle attacks are a concern.

F.1.1.2. RSA key exchange and authentication

With RSA, key exchange and server authentication are combined. The public key may be either contained in the server's certificate or may be a temporary RSA key sent in a server key exchange message. When temporary RSA keys are used, they are signed by the server's RSA or DSS certificate. The signature includes the current `ClientHello.random`, so old signatures and temporary keys cannot be replayed. Servers may use a single temporary RSA key for multiple negotiation sessions.

Note: The temporary RSA key option is useful if servers need large certificates but must comply with government-imposed size limits on keys used for key exchange.

After verifying the server's certificate, the client encrypts a `pre_master_secret` with the server's public key. By successfully decoding the `pre_master_secret` and producing a correct finished message, the server demonstrates that it knows the private key corresponding to the server certificate.

When RSA is used for key exchange, clients are authenticated using the certificate verify message (see [Section 7.4.8](#)). The client signs a value derived from the `master_secret` and all preceding handshake messages. These handshake messages include the server certificate, which binds the signature to the server, and `ServerHello.random`, which binds the signature to the current handshake process.

F.1.1.3. Diffie-Hellman key exchange with authentication

When Diffie-Hellman key exchange is used, the server can either supply a certificate containing fixed Diffie-Hellman parameters or can use the server key exchange message to send a set of temporary Diffie-Hellman parameters signed with a DSS or RSA certificate. Temporary parameters are hashed with the hello.random values before signing to ensure that attackers do not replay old parameters. In either case, the client can verify the certificate or signature to ensure that the parameters belong to the server.

If the client has a certificate containing fixed Diffie-Hellman parameters, its certificate contains the information required to complete the key exchange. Note that in this case the client and server will generate the same Diffie-Hellman result (i.e., pre_master_secret) every time they communicate. To prevent the pre_master_secret from staying in memory any longer than necessary, it should be converted into the master_secret as soon as possible. Client Diffie-Hellman parameters must be compatible with those supplied by the server for the key exchange to work.

If the client has a standard DSS or RSA certificate or is unauthenticated, it sends a set of temporary parameters to the server in the client key exchange message, then optionally uses a certificate verify message to authenticate itself.

F.1.2. Version rollback attacks

Because TLS includes substantial improvements over SSL Version 2.0, attackers may try to make TLS-capable clients and servers fall back to Version 2.0. This attack can occur if (and only if) two TLS-capable parties use an SSL 2.0 handshake.

Although the solution using non-random PKCS #1 block type 2 message padding is inelegant, it provides a reasonably secure way for Version 3.0 servers to detect the attack. This solution is not secure against attackers who can brute force the key and substitute a new ENCRYPTED-KEY-DATA message containing the same key (but with normal padding) before the application specified wait threshold has expired. Parties concerned about attacks of this scale should not be using 40-bit encryption keys anyway. Altering the padding of the least-significant 8 bytes of the PKCS padding does not impact security for the size of the signed hashes and RSA key lengths used in the protocol, since this is essentially equivalent to increasing the input block size by 8 bytes.

F.1.3. Detecting attacks against the handshake protocol

An attacker might try to influence the handshake exchange to make the parties select different encryption algorithms than they would normally choose. Because many implementations will support 40-bit exportable encryption and some may even support null encryption or MAC algorithms, this attack is of particular concern.

For this attack, an attacker must actively change one or more handshake messages. If this occurs, the client and server will compute different values for the handshake message hashes. As a result, the parties will not accept each others' finished messages. Without the master_secret, the attacker cannot repair the finished messages, so the attack will be discovered.

F.1.4. Resuming sessions

When a connection is established by resuming a session, new ClientHello.random and ServerHello.random values are hashed with the session's master_secret. Provided that the master_secret has not been compromised and that the secure hash operations used to produce the encryption keys and MAC secrets are secure, the connection should be secure and effectively independent from previous connections. Attackers cannot use known encryption keys or MAC secrets to compromise the master_secret without breaking the secure hash operations (which use both SHA and MD5).

Sessions cannot be resumed unless both the client and server agree. If either party suspects that the session may have been compromised, or that certificates may have expired or been revoked, it should force a full handshake. An upper limit of 24 hours is suggested for session ID lifetimes, since an attacker who obtains a master_secret may be able to impersonate the compromised party until the corresponding session ID is retired. Applications that may be run in relatively insecure environments should not write session IDs to stable storage.

F.1.5. MD5 and SHA

TLS uses hash functions very conservatively. Where possible, both MD5 and SHA are used in tandem to ensure that non-catastrophic flaws in one algorithm will not break the overall protocol.

F.2. Protecting application data

The master_secret is hashed with the ClientHello.random and ServerHello.random to produce unique data encryption keys and MAC secrets for each connection.

Outgoing data is protected with a MAC before transmission. To prevent message replay or modification attacks, the MAC is computed from the MAC secret, the sequence number, the message length, the message contents, and two fixed character strings. The message type field is necessary to ensure that messages intended for one TLS Record Layer client are not redirected to another. The sequence number ensures that attempts to delete or reorder messages will be detected. Since sequence numbers are 64-bits long, they should never overflow. Messages from one party cannot be inserted into the other's output, since they use independent MAC secrets. Similarly, the server-write and client-write keys are independent so stream cipher keys are used only once.

If an attacker does break an encryption key, all messages encrypted with it can be read. Similarly, compromise of a MAC key can make message modification attacks possible. Because MACs are also encrypted, message-alteration attacks generally require breaking the encryption algorithm as well as the MAC.

Note: MAC secrets may be larger than encryption keys, so messages can remain tamper resistant even if encryption keys are broken.

F.3. Final notes

For TLS to be able to provide a secure connection, both the client and server systems, keys, and applications must be secure. In addition, the implementation must be free of security errors.

The system is only as strong as the weakest key exchange and authentication algorithm supported, and only trustworthy cryptographic functions should be used. Short public keys, 40-bit bulk encryption keys, and anonymous servers should be used with great caution. Implementations and users must be careful when deciding which certificates and certificate authorities are acceptable; a dishonest certificate authority can do tremendous damage.

G. Patent Statement

Some of the cryptographic algorithms proposed for use in this protocol have patent claims on them. In addition Netscape Communications Corporation has a patent claim on the Secure Sockets Layer (SSL) work that this standard is based on. The Internet Standards Process as defined in RFC 2026 requests that a statement be obtained from a Patent holder indicating that a license will be made available to applicants under reasonable terms and conditions.

The Massachusetts Institute of Technology has granted RSA Data Security, Inc., exclusive sub-licensing rights to the following patent issued in the United States:

Cryptographic Communications System and Method ("RSA"), No. 4,405,829

Netscape Communications Corporation has been issued the following patent in the United States:

Secure Socket Layer Application Program Apparatus And Method ("SSL"), No. 5,657,390

Netscape Communications has issued the following statement:

Intellectual Property Rights

Secure Sockets Layer

The United States Patent and Trademark Office ("the PT0") recently issued U.S. Patent No. 5,657,390 ("the SSL Patent") to Netscape for inventions described as Secure Sockets Layers ("SSL"). The IETF is currently considering adopting SSL as a transport protocol with security features. Netscape encourages the royalty-free adoption and use of the SSL protocol upon the following terms and conditions:

- * If you already have a valid SSL Ref license today which includes source code from Netscape, an additional patent license under the SSL patent is not required.
- * If you don't have an SSL Ref license, you may have a royalty free license to build implementations covered by the SSL Patent Claims or the IETF TLS specification provided that you do not to assert any patent rights against Netscape or other companies for the implementation of SSL or the IETF TLS recommendation.

RFC 2246

The TLS Protocol Version 1.0

January 1999

What are "Patent Claims":

Patent claims are claims in an issued foreign or domestic patent that:

- 1) must be infringed in order to implement methods or build products according to the IETF TLS specification; or
- 2) patent claims which require the elements of the SSL patent claims and/or their equivalents to be infringed.

The Internet Society, Internet Architecture Board, Internet Engineering Steering Group and the Corporation for National Research Initiatives take no position on the validity or scope of the patents and patent applications, nor on the appropriateness of the terms of the assurance. The Internet Society and other groups mentioned above have not made any determination as to any other intellectual property rights which may apply to the practice of this standard. Any further consideration of these matters is the user's own responsibility.

Security Considerations

Security issues are discussed throughout this memo.

References

- [3DES] W. Tuchman, "Hellman Presents No Shortcut Solutions To DES," IEEE Spectrum, v. 16, n. 7, July 1979, pp40-41.
- [BLEI] Bleichenbacher D., "Chosen Ciphertext Attacks against Protocols Based on RSA Encryption Standard PKCS #1" in Advances in Cryptology -- CRYPTO'98, LNCS vol. 1462, pages: 1--12, 1998.
- [DES] ANSI X3.106, "American National Standard for Information Systems-Data Link Encryption," American National Standards Institute, 1983.
- [DH1] W. Diffie and M. E. Hellman, "New Directions in Cryptography," IEEE Transactions on Information Theory, V. IT-22, n. 6, Jun 1977, pp. 74-84.
- [DSS] NIST FIPS PUB 186, "Digital Signature Standard," National Institute of Standards and Technology, U.S. Department of Commerce, May 18, 1994.
- [FTP] Postel J., and J. Reynolds, "File Transfer Protocol", STD 9, RFC 959, October 1985.

Dierks & Allen

Standards Track

[Page 75]

RFC 2246

The TLS Protocol Version 1.0

January 1999

- [HTTP] Berners-Lee, T., Fielding, R., and H. Frystyk, "Hypertext Transfer Protocol -- HTTP/1.0", RFC 1945, May 1996.
- [HMAC] Krawczyk, H., Bellare, M., and R. Canetti, "HMAC: Keyed-Hashing for Message Authentication," RFC 2104, February 1997.
- [IDEA] X. Lai, "On the Design and Security of Block Ciphers," ETH Series in Information Processing, v. 1, Konstanz: Hartung-Gorre Verlag, 1992.
- [MD2] Kaliski, B., "The MD2 Message Digest Algorithm", RFC 1319, April 1992.
- [MD5] Rivest, R., "The MD5 Message Digest Algorithm", RFC 1321, April 1992.
- [PKCS1] RSA Laboratories, "PKCS #1: RSA Encryption Standard," version 1.5, November 1993.
- [PKCS6] RSA Laboratories, "PKCS #6: RSA Extended Certificate Syntax Standard," version 1.5, November 1993.
- [PKCS7] RSA Laboratories, "PKCS #7: RSA Cryptographic Message Syntax Standard," version 1.5, November 1993.
- [PKIX] Housley, R., Ford, W., Polk, W. and D. Solo, "Internet Public Key Infrastructure: Part I: X.509 Certificate and CRL Profile", RFC 2459, January 1999.
- [RC2] Rivest, R., "A Description of the RC2(r) Encryption Algorithm", RFC 2268, January 1998.
- [RC4] Thayer, R. and K. Kaukonen, A Stream Cipher Encryption Algorithm, Work in Progress.
- [RSA] R. Rivest, A. Shamir, and L. M. Adleman, "A Method for Obtaining Digital Signatures and Public-Key Cryptosystems," Communications of the ACM, v. 21, n. 2, Feb 1978, pp. 120-126.
- [RSADSI] Contact RSA Data Security, Inc., Tel: 415-595-8782
- [SCH] B. Schneier. Applied Cryptography: Protocols, Algorithms, and Source Code in C, Published by John Wiley & Sons, Inc. 1994.

RFC 2246

The TLS Protocol Version 1.0

January 1999

- [SHA] NIST FIPS PUB 180-1, "Secure Hash Standard," National Institute of Standards and Technology, U.S. Department of Commerce, Work in Progress, May 31, 1994.
- [SSL2] Hickman, Kipp, "The SSL Protocol", Netscape Communications Corp., Feb 9, 1995.
- [SSL3] A. Frier, P. Karlton, and P. Kocher, "The SSL 3.0 Protocol", Netscape Communications Corp., Nov 18, 1996.
- [TCP] Postel, J., "Transmission Control Protocol," STD 7, RFC 793, September 1981.
- [TEL] Postel J., and J. Reynolds, "Telnet Protocol Specifications", STD 8, RFC 854, May 1993.
- [TEL] Postel J., and J. Reynolds, "Telnet Option Specifications", STD 8, RFC 855, May 1993.
- [X509] CCITT. Recommendation X.509: "The Directory - Authentication Framework". 1988.
- [XDR] R. Srinivansan, Sun Microsystems, RFC-1832: XDR: External Data Representation Standard, August 1995.

Credits

Win Treese
Open Market

E-Mail: treese@openmarket.com

Editors

Christopher Allen
Certicom

E-Mail: callen@certicom.com

Tim Dierks
Certicom

E-Mail: tdierks@certicom.com

Authors' Addresses

Tim Dierks
Certicom

E-Mail: tdierks@certicom.com

Philip L. Karlton
Netscape Communications

Dierks & Allen

Standards Track

[Page 77]

RFC 2246

The TLS Protocol Version 1.0

January 1999

Alan O. Freier
Netscape Communications

E-Mail: freier@netscape.com

Paul C. Kocher
Independent Consultant

E-Mail: pck@netcom.com

Other contributors

Martin Abadi
Digital Equipment Corporation

E-Mail: ma@pa.dec.com

Robert Relyea
Netscape Communications

E-Mail: relyea@netscape.com

Ran Canetti
IBM Watson Research Center

E-Mail: canetti@watson.ibm.com

Jim Roskind
Netscape Communications

E-Mail: jar@netscape.com

Taher Elgamal
Securify

E-Mail: elgamal@securify.com

Micheal J. Sabin, Ph. D.
Consulting Engineer

E-Mail: msabin@netcom.com

Anil R. Gangolli
Structured Arts Computing Corp.

E-Mail: gangolli@structuredarts.com

Dan Simon
Microsoft

E-Mail: dansimon@microsoft.com

Kipp E.B. Hickman
Netscape Communications

E-Mail: kipp@netscape.com

Tom Weinstein
Netscape Communications

E-Mail: tomw@netscape.com

Hugo Krawczyk
IBM Watson Research Center

E-Mail: hugo@watson.ibm.com

Comments

The discussion list for the IETF TLS working group is located at the e-mail address <ietf-tls@lists.consensus.com>. Information on the group and information on how to subscribe to the list is at <<http://lists.consensus.com/>>.

RFC 2246

The TLS Protocol Version 1.0

January 1999

Archives of the list can be found at:

<<http://www.imc.org/ietf-tls/mail-archive/>>

RFC 2246

The TLS Protocol Version 1.0

January 1999

Full Copyright Statement

Copyright (C) The Internet Society (1999). All Rights Reserved.

This document and translations of it may be copied and furnished to others, and derivative works that comment on or otherwise explain it or assist in its implementation may be prepared, copied, published and distributed, in whole or in part, without restriction of any kind, provided that the above copyright notice and this paragraph are included on all such copies and derivative works. However, this document itself may not be modified in any way, such as by removing the copyright notice or references to the Internet Society or other Internet organizations, except as needed for the purpose of developing Internet standards in which case the procedures for copyrights defined in the Internet Standards process must be followed, or as required to translate it into languages other than English.

The limited permissions granted above are perpetual and will not be revoked by the Internet Society or its successors or assigns.

This document and the information contained herein is provided on an "AS IS" basis and THE INTERNET SOCIETY AND THE INTERNET ENGINEERING TASK FORCE DISCLAIMS ALL WARRANTIES, EXPRESS OR IMPLIED, INCLUDING BUT NOT LIMITED TO ANY WARRANTY THAT THE USE OF THE INFORMATION HEREIN WILL NOT INFRINGE ANY RIGHTS OR ANY IMPLIED WARRANTIES OF MERCHANTABILITY OR FITNESS FOR A PARTICULAR PURPOSE.