Network Working Group T. Dierks

Request for Comments: 5246 Independent

Obsoletes: 3268, 4346, 4366 E. Rescorla

Updates: 4492 RTFM, Inc.

Category: Standards Track August 2008

The Transport Layer Security (TLS) Protocol

Version 1.2

Status of This Memo

This document specifies an Internet standards track protocol for the

Internet community, and requests discussion and suggestions for

improvements. Please refer to the current edition of the "Internet

Official Protocol Standards" (STD 1) for the standardization state

and status of this protocol. Distribution of this memo is unlimited.

Abstract

This document specifies Version 1.2 of the Transport Layer Security

(TLS) protocol. The TLS protocol provides communications security

over the Internet. The protocol allows client/server applications to

communicate in a way that is designed to prevent eavesdropping,

tampering, or message forgery.

Table of Contents

1. Introduction ....................................................4

1.1. Requirements Terminology ...................................5

1.2. Major Differences from TLS 1.1 .............................5

2. Goals ...........................................................6

3. Goals of This Document ..........................................7

4. Presentation Language ...........................................7

4.1. Basic Block Size ...........................................7

4.2. Miscellaneous ..............................................8

4.3. Vectors ....................................................8

4.4. Numbers ....................................................9

4.5. Enumerateds ................................................9

4.6. Constructed Types .........................................10

4.6.1. Variants ...........................................10

4.7. Cryptographic Attributes ..................................12

4.8. Constants .................................................14

5. HMAC and the Pseudorandom Function .............................14

6. The TLS Record Protocol ........................................15

6.1. Connection States .........................................16

6.2. Record Layer ..............................................19

6.2.1. Fragmentation ......................................19

Dierks & Rescorla Standards Track [Page 1]

RFC 5246 TLS August 2008

6.2.2. Record Compression and Decompression ...............20

6.2.3. Record Payload Protection ..........................21

6.2.3.1. Null or Standard Stream Cipher ............22

6.2.3.2. CBC Block Cipher ..........................22

6.2.3.3. AEAD Ciphers ..............................24

6.3. Key Calculation ...........................................25

7. The TLS Handshaking Protocols ..................................26

7.1. Change Cipher Spec Protocol ...............................27

7.2. Alert Protocol ............................................28

7.2.1. Closure Alerts .....................................29

7.2.2. Error Alerts .......................................30

7.3. Handshake Protocol Overview ...............................33

7.4. Handshake Protocol ........................................37

7.4.1. Hello Messages .....................................38

7.4.1.1. Hello Request .............................38

7.4.1.2. Client Hello ..............................39

7.4.1.3. Server Hello ..............................42

7.4.1.4. Hello Extensions ..........................44

7.4.1.4.1. Signature Algorithms ...........45

7.4.2. Server Certificate .................................47

7.4.3. Server Key Exchange Message ........................50

7.4.4. Certificate Request ................................53

7.4.5. Server Hello Done ..................................55

7.4.6. Client Certificate .................................55

7.4.7. Client Key Exchange Message ........................57

7.4.7.1. RSA-Encrypted Premaster Secret Message ....58

7.4.7.2. Client Diffie-Hellman Public Value ........61

7.4.8. Certificate Verify .................................62

7.4.9. Finished ...........................................63

8. Cryptographic Computations .....................................64

8.1. Computing the Master Secret ...............................64

8.1.1. RSA ................................................65

8.1.2. Diffie-Hellman .....................................65

9. Mandatory Cipher Suites ........................................65

10. Application Data Protocol .....................................65

11. Security Considerations .......................................65

12. IANA Considerations ...........................................65

Appendix A. Protocol Data Structures and Constant Values ..........68

A.1. Record Layer ..............................................68

A.2. Change Cipher Specs Message ...............................69

A.3. Alert Messages ............................................69

A.4. Handshake Protocol ........................................70

A.4.1. Hello Messages .....................................71

A.4.2. Server Authentication and Key Exchange Messages ....72

A.4.3. Client Authentication and Key Exchange Messages ....74

A.4.4. Handshake Finalization Message .....................74

A.5. The Cipher Suite ..........................................75

A.6. The Security Parameters ...................................77

Dierks & Rescorla Standards Track [Page 2]

RFC 5246 TLS August 2008

A.7. Changes to RFC 4492 .......................................78

Appendix B. Glossary ..............................................78

Appendix C. Cipher Suite Definitions ..............................83

Appendix D. Implementation Notes ..................................85

D.1. Random Number Generation and Seeding ......................85

D.2. Certificates and Authentication ...........................85

D.3. Cipher Suites .............................................85

D.4. Implementation Pitfalls ...................................85

Appendix E. Backward Compatibility ................................87

E.1. Compatibility with TLS 1.0/1.1 and SSL 3.0 ................87

E.2. Compatibility with SSL 2.0 ................................88

E.3. Avoiding Man-in-the-Middle Version Rollback ...............90

Appendix F. Security Analysis .....................................91

F.1. Handshake Protocol ........................................91

F.1.1. Authentication and Key Exchange ....................91

F.1.1.1. Anonymous Key Exchange ....................91

F.1.1.2. RSA Key Exchange and Authentication .......92

F.1.1.3. Diffie-Hellman Key Exchange with

Authentication ............................92

F.1.2. Version Rollback Attacks ...........................93

F.1.3. Detecting Attacks Against the Handshake Protocol ...94

F.1.4. Resuming Sessions ..................................94

F.2. Protecting Application Data ...............................94

F.3. Explicit IVs ..............................................95

F.4. Security of Composite Cipher Modes ........................95

F.5. Denial of Service .........................................96

F.6. Final Notes ...............................................96

Normative References ..............................................97

Informative References ............................................98

Working Group Information ........................................101

Contributors .....................................................101

Dierks & Rescorla Standards Track [Page 3]

RFC 5246 TLS August 2008

1. Introduction

The primary goal of the TLS protocol is to provide privacy and data

integrity between two communicating applications. The protocol is

composed of two layers: the TLS Record Protocol and the TLS Handshake

Protocol. At the lowest level, layered on top of some reliable

transport protocol (e.g., TCP [TCP]), is the TLS Record Protocol.

The TLS Record Protocol provides connection security that has two

basic properties:

- The connection is private. Symmetric cryptography is used for

data encryption (e.g., AES [AES], RC4 [SCH], etc.). The keys for

this symmetric encryption are generated uniquely for each

connection and are based on a secret negotiated by another

protocol (such as the TLS Handshake Protocol). The Record

Protocol can also be used without encryption.

- The connection is reliable. Message transport includes a message

integrity check using a keyed MAC. Secure hash functions (e.g.,

SHA-1, etc.) are used for MAC computations. The Record Protocol

can operate without a MAC, but is generally only used in this mode

while another protocol is using the Record Protocol as a transport

for negotiating security parameters.

The TLS Record Protocol is used for encapsulation of various higher-

level protocols. One such encapsulated protocol, the TLS Handshake

Protocol, allows the server and client to authenticate each other and

to negotiate an encryption algorithm and cryptographic keys before

the application protocol transmits or receives its first byte of

data. The TLS Handshake Protocol provides connection security that

has three basic properties:

- The peer's identity can be authenticated using asymmetric, or

public key, cryptography (e.g., RSA [RSA], DSA [DSS], etc.). This

authentication can be made optional, but is generally required for

at least one of the peers.

- The negotiation of a shared secret is secure: the negotiated

secret is unavailable to eavesdroppers, and for any authenticated

connection the secret cannot be obtained, even by an attacker who

can place himself in the middle of the connection.

- The negotiation is reliable: no attacker can modify the

negotiation communication without being detected by the parties to

the communication.

Dierks & Rescorla Standards Track [Page 4]

RFC 5246 TLS August 2008

One advantage of TLS is that it is application protocol independent.

Higher-level protocols can layer on top of the TLS protocol

transparently. The TLS standard, however, does not specify how

protocols add security with TLS; the decisions on how to initiate TLS

handshaking and how to interpret the authentication certificates

exchanged are left to the judgment of the designers and implementors

of protocols that run on top of TLS.

1.1. Requirements Terminology

The key words "MUST", "MUST NOT", "REQUIRED", "SHALL", "SHALL NOT",

"SHOULD", "SHOULD NOT", "RECOMMENDED", "MAY", and "OPTIONAL" in this

document are to be interpreted as described in RFC 2119 [REQ].

1.2. Major Differences from TLS 1.1

This document is a revision of the TLS 1.1 [TLS1.1] protocol which

contains improved flexibility, particularly for negotiation of

cryptographic algorithms. The major changes are:

- The MD5/SHA-1 combination in the pseudorandom function (PRF) has

been replaced with cipher-suite-specified PRFs. All cipher suites

in this document use P\_SHA256.

- The MD5/SHA-1 combination in the digitally-signed element has been

replaced with a single hash. Signed elements now include a field

that explicitly specifies the hash algorithm used.

- Substantial cleanup to the client's and server's ability to

specify which hash and signature algorithms they will accept.

Note that this also relaxes some of the constraints on signature

and hash algorithms from previous versions of TLS.

- Addition of support for authenticated encryption with additional

data modes.

- TLS Extensions definition and AES Cipher Suites were merged in

from external [TLSEXT] and [TLSAES].

- Tighter checking of EncryptedPreMasterSecret version numbers.

- Tightened up a number of requirements.

- Verify\_data length now depends on the cipher suite (default is

still 12).

- Cleaned up description of Bleichenbacher/Klima attack defenses.

Dierks & Rescorla Standards Track [Page 5]

RFC 5246 TLS August 2008

- Alerts MUST now be sent in many cases.

- After a certificate\_request, if no certificates are available,

clients now MUST send an empty certificate list.

- TLS\_RSA\_WITH\_AES\_128\_CBC\_SHA is now the mandatory to implement

cipher suite.

- Added HMAC-SHA256 cipher suites.

- Removed IDEA and DES cipher suites. They are now deprecated and

will be documented in a separate document.

- Support for the SSLv2 backward-compatible hello is now a MAY, not

a SHOULD, with sending it a SHOULD NOT. Support will probably

become a SHOULD NOT in the future.

- Added limited "fall-through" to the presentation language to allow

multiple case arms to have the same encoding.

- Added an Implementation Pitfalls sections

- The usual clarifications and editorial work.

2. Goals

The goals of the TLS protocol, in order of priority, are as follows:

1. Cryptographic security: TLS should be used to establish a secure

connection between two parties.

2. Interoperability: Independent programmers should be able to

develop applications utilizing TLS that can successfully exchange

cryptographic parameters without knowledge of one another's code.

3. Extensibility: TLS seeks to provide a framework into which new

public key and bulk encryption methods can be incorporated as

necessary. This will also accomplish two sub-goals: preventing

the need to create a new protocol (and risking the introduction of

possible new weaknesses) and avoiding the need to implement an

entire new security library.

4. Relative efficiency: Cryptographic operations tend to be highly

CPU intensive, particularly public key operations. For this

reason, the TLS protocol has incorporated an optional session

caching scheme to reduce the number of connections that need to be

established from scratch. Additionally, care has been taken to

reduce network activity.

Dierks & Rescorla Standards Track [Page 6]

RFC 5246 TLS August 2008

3. Goals of This Document

This document and the TLS protocol itself are based on the SSL 3.0

Protocol Specification as published by Netscape. The differences

between this protocol and SSL 3.0 are not dramatic, but they are

significant enough that the various versions of TLS and SSL 3.0 do

not interoperate (although each protocol incorporates a mechanism by

which an implementation can back down to prior versions). This

document is intended primarily for readers who will be implementing

the protocol and for those doing cryptographic analysis of it. The

specification has been written with this in mind, and it is intended

to reflect the needs of those two groups. For that reason, many of

the algorithm-dependent data structures and rules are included in the

body of the text (as opposed to in an appendix), providing easier

access to them.

This document is not intended to supply any details of service

definition or of interface definition, although it does cover select

areas of policy as they are required for the maintenance of solid

security.

4. Presentation Language

This document deals with the formatting of data in an external

representation. The following very basic and somewhat casually

defined presentation syntax will be used. The syntax draws from

several sources in its structure. Although it resembles the

programming language "C" in its syntax and XDR [XDR] in both its

syntax and intent, it would be risky to draw too many parallels. The

purpose of this presentation language is to document TLS only; it has

no general application beyond that particular goal.

4.1. Basic Block Size

The representation of all data items is explicitly specified. The

basic data block size is one byte (i.e., 8 bits). Multiple byte data

items are concatenations of bytes, from left to right, from top to

bottom. From the byte stream, a multi-byte item (a numeric in the

example) is formed (using C notation) by:

value = (byte[0] << 8\*(n-1)) | (byte[1] << 8\*(n-2)) |

... | byte[n-1];

This byte ordering for multi-byte values is the commonplace network

byte order or big-endian format.

Dierks & Rescorla Standards Track [Page 7]

RFC 5246 TLS August 2008

4.2. Miscellaneous

Comments begin with "/\*" and end with "\*/".

Optional components are denoted by enclosing them in "[[ ]]" double

brackets.

Single-byte entities containing uninterpreted data are of type

opaque.

4.3. Vectors

A vector (single-dimensioned array) is a stream of homogeneous data

elements. The size of the vector may be specified at documentation

time or left unspecified until runtime. In either case, the length

declares the number of bytes, not the number of elements, in the

vector. The syntax for specifying a new type, T', that is a fixed-

length vector of type T is

T T'[n];

Here, T' occupies n bytes in the data stream, where n is a multiple

of the size of T. The length of the vector is not included in the

encoded stream.

In the following example, Datum is defined to be three consecutive

bytes that the protocol does not interpret, while Data is three

consecutive Datum, consuming a total of nine bytes.

opaque Datum[3]; /\* three uninterpreted bytes \*/

Datum Data[9]; /\* 3 consecutive 3 byte vectors \*/

Variable-length vectors are defined by specifying a subrange of legal

lengths, inclusively, using the notation <floor..ceiling>. When

these are encoded, the actual length precedes the vector's contents

in the byte stream. The length will be in the form of a number

consuming as many bytes as required to hold the vector's specified

maximum (ceiling) length. A variable-length vector with an actual

length field of zero is referred to as an empty vector.

T T'<floor..ceiling>;

In the following example, mandatory is a vector that must contain

between 300 and 400 bytes of type opaque. It can never be empty.

The actual length field consumes two bytes, a uint16, which is

sufficient to represent the value 400 (see Section 4.4). On the

other hand, longer can represent up to 800 bytes of data, or 400

uint16 elements, and it may be empty. Its encoding will include a

Dierks & Rescorla Standards Track [Page 8]

RFC 5246 TLS August 2008

two-byte actual length field prepended to the vector. The length of

an encoded vector must be an even multiple of the length of a single

element (for example, a 17-byte vector of uint16 would be illegal).

opaque mandatory<300..400>;

/\* length field is 2 bytes, cannot be empty \*/

uint16 longer<0..800>;

/\* zero to 400 16-bit unsigned integers \*/

4.4. Numbers

The basic numeric data type is an unsigned byte (uint8). All larger

numeric data types are formed from fixed-length series of bytes

concatenated as described in Section 4.1 and are also unsigned. The

following numeric types are predefined.

uint8 uint16[2];

uint8 uint24[3];

uint8 uint32[4];

uint8 uint64[8];

All values, here and elsewhere in the specification, are stored in

network byte (big-endian) order; the uint32 represented by the hex

bytes 01 02 03 04 is equivalent to the decimal value 16909060.

Note that in some cases (e.g., DH parameters) it is necessary to

represent integers as opaque vectors. In such cases, they are

represented as unsigned integers (i.e., leading zero octets are not

required even if the most significant bit is set).

4.5. Enumerateds

An additional sparse data type is available called enum. A field of

type enum can only assume the values declared in the definition.

Each definition is a different type. Only enumerateds of the same

type may be assigned or compared. Every element of an enumerated

must be assigned a value, as demonstrated in the following example.

Since the elements of the enumerated are not ordered, they can be

assigned any unique value, in any order.

enum { e1(v1), e2(v2), ... , en(vn) [[, (n)]] } Te;

An enumerated occupies as much space in the byte stream as would its

maximal defined ordinal value. The following definition would cause

one byte to be used to carry fields of type Color.

enum { red(3), blue(5), white(7) } Color;

Dierks & Rescorla Standards Track [Page 9]

RFC 5246 TLS August 2008

One may optionally specify a value without its associated tag to

force the width definition without defining a superfluous element.

In the following example, Taste will consume two bytes in the data

stream but can only assume the values 1, 2, or 4.

enum { sweet(1), sour(2), bitter(4), (32000) } Taste;

The names of the elements of an enumeration are scoped within the

defined type. In the first example, a fully qualified reference to

the second element of the enumeration would be Color.blue. Such

qualification is not required if the target of the assignment is well

specified.

Color color = Color.blue; /\* overspecified, legal \*/

Color color = blue; /\* correct, type implicit \*/

For enumerateds that are never converted to external representation,

the numerical information may be omitted.

enum { low, medium, high } Amount;

4.6. Constructed Types

Structure types may be constructed from primitive types for

convenience. Each specification declares a new, unique type. The

syntax for definition is much like that of C.

struct {

T1 f1;

T2 f2;

...

Tn fn;

} [[T]];

The fields within a structure may be qualified using the type's name,

with a syntax much like that available for enumerateds. For example,

T.f2 refers to the second field of the previous declaration.

Structure definitions may be embedded.

4.6.1. Variants

Defined structures may have variants based on some knowledge that is

available within the environment. The selector must be an enumerated

type that defines the possible variants the structure defines. There

must be a case arm for every element of the enumeration declared in

the select. Case arms have limited fall-through: if two case arms

follow in immediate succession with no fields in between, then they

Dierks & Rescorla Standards Track [Page 10]

RFC 5246 TLS August 2008

both contain the same fields. Thus, in the example below, "orange"

and "banana" both contain V2. Note that this is a new piece of

syntax in TLS 1.2.

The body of the variant structure may be given a label for reference.

The mechanism by which the variant is selected at runtime is not

prescribed by the presentation language.

struct {

T1 f1;

T2 f2;

....

Tn fn;

select (E) {

case e1: Te1;

case e2: Te2;

case e3: case e4: Te3;

....

case en: Ten;

} [[fv]];

} [[Tv]];

For example:

enum { apple, orange, banana } VariantTag;

struct {

uint16 number;

opaque string<0..10>; /\* variable length \*/

} V1;

struct {

uint32 number;

opaque string[10]; /\* fixed length \*/

} V2;

struct {

select (VariantTag) { /\* value of selector is implicit \*/

case apple:

V1; /\* VariantBody, tag = apple \*/

case orange:

case banana:

V2; /\* VariantBody, tag = orange or banana \*/

} variant\_body; /\* optional label on variant \*/

} VariantRecord;

Dierks & Rescorla Standards Track [Page 11]

RFC 5246 TLS August 2008

4.7. Cryptographic Attributes

The five cryptographic operations -- digital signing, stream cipher

encryption, block cipher encryption, authenticated encryption with

additional data (AEAD) encryption, and public key encryption -- are

designated digitally-signed, stream-ciphered, block-ciphered, aead-

ciphered, and public-key-encrypted, respectively. A field's

cryptographic processing is specified by prepending an appropriate

key word designation before the field's type specification.

Cryptographic keys are implied by the current session state (see

Section 6.1).

A digitally-signed element is encoded as a struct DigitallySigned:

struct {

SignatureAndHashAlgorithm algorithm;

opaque signature<0..2^16-1>;

} DigitallySigned;

The algorithm field specifies the algorithm used (see Section

7.4.1.4.1 for the definition of this field). Note that the

introduction of the algorithm field is a change from previous

versions. The signature is a digital signature using those

algorithms over the contents of the element. The contents themselves

do not appear on the wire but are simply calculated. The length of

the signature is specified by the signing algorithm and key.

In RSA signing, the opaque vector contains the signature generated

using the RSASSA-PKCS1-v1\_5 signature scheme defined in [PKCS1]. As

discussed in [PKCS1], the DigestInfo MUST be DER-encoded [X680]

[X690]. For hash algorithms without parameters (which includes

SHA-1), the DigestInfo.AlgorithmIdentifier.parameters field MUST be

NULL, but implementations MUST accept both without parameters and

with NULL parameters. Note that earlier versions of TLS used a

different RSA signature scheme that did not include a DigestInfo

encoding.

In DSA, the 20 bytes of the SHA-1 hash are run directly through the

Digital Signing Algorithm with no additional hashing. This produces

two values, r and s. The DSA signature is an opaque vector, as

above, the contents of which are the DER encoding of:

Dss-Sig-Value ::= SEQUENCE {

r INTEGER,

s INTEGER

}

Dierks & Rescorla Standards Track [Page 12]

RFC 5246 TLS August 2008

Note: In current terminology, DSA refers to the Digital Signature

Algorithm and DSS refers to the NIST standard. In the original SSL

and TLS specs, "DSS" was used universally. This document uses "DSA"

to refer to the algorithm, "DSS" to refer to the standard, and it

uses "DSS" in the code point definitions for historical continuity.

In stream cipher encryption, the plaintext is exclusive-ORed with an

identical amount of output generated from a cryptographically secure

keyed pseudorandom number generator.

In block cipher encryption, every block of plaintext encrypts to a

block of ciphertext. All block cipher encryption is done in CBC

(Cipher Block Chaining) mode, and all items that are block-ciphered

will be an exact multiple of the cipher block length.

In AEAD encryption, the plaintext is simultaneously encrypted and

integrity protected. The input may be of any length, and aead-

ciphered output is generally larger than the input in order to

accommodate the integrity check value.

In public key encryption, a public key algorithm is used to encrypt

data in such a way that it can be decrypted only with the matching

private key. A public-key-encrypted element is encoded as an opaque

vector <0..2^16-1>, where the length is specified by the encryption

algorithm and key.

RSA encryption is done using the RSAES-PKCS1-v1\_5 encryption scheme

defined in [PKCS1].

In the following example

stream-ciphered struct {

uint8 field1;

uint8 field2;

digitally-signed opaque {

uint8 field3<0..255>;

uint8 field4;

};

} UserType;

The contents of the inner struct (field3 and field4) are used as

input for the signature/hash algorithm, and then the entire structure

is encrypted with a stream cipher. The length of this structure, in

bytes, would be equal to two bytes for field1 and field2, plus two

bytes for the signature and hash algorithm, plus two bytes for the

length of the signature, plus the length of the output of the signing

Dierks & Rescorla Standards Track [Page 13]

RFC 5246 TLS August 2008

algorithm. The length of the signature is known because the

algorithm and key used for the signing are known prior to encoding or

decoding this structure.

4.8. Constants

Typed constants can be defined for purposes of specification by

declaring a symbol of the desired type and assigning values to it.

Under-specified types (opaque, variable-length vectors, and

structures that contain opaque) cannot be assigned values. No fields

of a multi-element structure or vector may be elided.

For example:

struct {

uint8 f1;

uint8 f2;

} Example1;

Example1 ex1 = {1, 4}; /\* assigns f1 = 1, f2 = 4 \*/

5. HMAC and the Pseudorandom Function

The TLS record layer uses a keyed Message Authentication Code (MAC)

to protect message integrity. The cipher suites defined in this

document use a construction known as HMAC, described in [HMAC], which

is based on a hash function. Other cipher suites MAY define their

own MAC constructions, if needed.

In addition, a construction is required to do expansion of secrets

into blocks of data for the purposes of key generation or validation.

This pseudorandom function (PRF) takes as input a secret, a seed, and

an identifying label and produces an output of arbitrary length.

In this section, we define one PRF, based on HMAC. This PRF with the

SHA-256 hash function is used for all cipher suites defined in this

document and in TLS documents published prior to this document when

TLS 1.2 is negotiated. New cipher suites MUST explicitly specify a

PRF and, in general, SHOULD use the TLS PRF with SHA-256 or a

stronger standard hash function.

First, we define a data expansion function, P\_hash(secret, data),

that uses a single hash function to expand a secret and seed into an

arbitrary quantity of output:

Dierks & Rescorla Standards Track [Page 14]

RFC 5246 TLS August 2008

P\_hash(secret, seed) = HMAC\_hash(secret, A(1) + seed) +

HMAC\_hash(secret, A(2) + seed) +

HMAC\_hash(secret, A(3) + seed) + ...

where + indicates concatenation.

A() is defined as:

A(0) = seed

A(i) = HMAC\_hash(secret, A(i-1))

P\_hash can be iterated as many times as necessary to produce the

required quantity of data. For example, if P\_SHA256 is being used to

create 80 bytes of data, it will have to be iterated three times

(through A(3)), creating 96 bytes of output data; the last 16 bytes

of the final iteration will then be discarded, leaving 80 bytes of

output data.

TLS's PRF is created by applying P\_hash to the secret as:

PRF(secret, label, seed) = P\_<hash>(secret, label + seed)

The label is an ASCII string. It should be included in the exact

form it is given without a length byte or trailing null character.

For example, the label "slithy toves" would be processed by hashing

the following bytes:

73 6C 69 74 68 79 20 74 6F 76 65 73

6. The TLS Record Protocol

The TLS Record Protocol is a layered protocol. At each layer,

messages may include fields for length, description, and content.

The Record Protocol takes messages to be transmitted, fragments the

data into manageable blocks, optionally compresses the data, applies

a MAC, encrypts, and transmits the result. Received data is

decrypted, verified, decompressed, reassembled, and then delivered to

higher-level clients.

Four protocols that use the record protocol are described in this

document: the handshake protocol, the alert protocol, the change

cipher spec protocol, and the application data protocol. In order to

allow extension of the TLS protocol, additional record content types

can be supported by the record protocol. New record content type

values are assigned by IANA in the TLS Content Type Registry as

described in Section 12.

Dierks & Rescorla Standards Track [Page 15]

RFC 5246 TLS August 2008

Implementations MUST NOT send record types not defined in this

document unless negotiated by some extension. If a TLS

implementation receives an unexpected record type, it MUST send an

unexpected\_message alert.

Any protocol designed for use over TLS must be carefully designed to

deal with all possible attacks against it. As a practical matter,

this means that the protocol designer must be aware of what security

properties TLS does and does not provide and cannot safely rely on

the latter.

Note in particular that type and length of a record are not protected

by encryption. If this information is itself sensitive, application

designers may wish to take steps (padding, cover traffic) to minimize

information leakage.

6.1. Connection States

A TLS connection state is the operating environment of the TLS Record

Protocol. It specifies a compression algorithm, an encryption

algorithm, and a MAC algorithm. In addition, the parameters for

these algorithms are known: the MAC key and the bulk encryption keys

for the connection in both the read and the write directions.

Logically, there are always four connection states outstanding: the

current read and write states, and the pending read and write states.

All records are processed under the current read and write states.

The security parameters for the pending states can be set by the TLS

Handshake Protocol, and the ChangeCipherSpec can selectively make

either of the pending states current, in which case the appropriate

current state is disposed of and replaced with the pending state; the

pending state is then reinitialized to an empty state. It is illegal

to make a state that has not been initialized with security

parameters a current state. The initial current state always

specifies that no encryption, compression, or MAC will be used.

The security parameters for a TLS Connection read and write state are

set by providing the following values:

connection end

Whether this entity is considered the "client" or the "server" in

this connection.

PRF algorithm

An algorithm used to generate keys from the master secret (see

Sections 5 and 6.3).

Dierks & Rescorla Standards Track [Page 16]

RFC 5246 TLS August 2008

bulk encryption algorithm

An algorithm to be used for bulk encryption. This specification

includes the key size of this algorithm, whether it is a block,

stream, or AEAD cipher, the block size of the cipher (if

appropriate), and the lengths of explicit and implicit

initialization vectors (or nonces).

MAC algorithm

An algorithm to be used for message authentication. This

specification includes the size of the value returned by the MAC

algorithm.

compression algorithm

An algorithm to be used for data compression. This specification

must include all information the algorithm requires to do

compression.

master secret

A 48-byte secret shared between the two peers in the connection.

client random

A 32-byte value provided by the client.

server random

A 32-byte value provided by the server.

These parameters are defined in the presentation language as:

enum { server, client } ConnectionEnd;

enum { tls\_prf\_sha256 } PRFAlgorithm;

enum { null, rc4, 3des, aes }

BulkCipherAlgorithm;

enum { stream, block, aead } CipherType;

enum { null, hmac\_md5, hmac\_sha1, hmac\_sha256,

hmac\_sha384, hmac\_sha512} MACAlgorithm;

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

/\* The algorithms specified in CompressionMethod, PRFAlgorithm,

BulkCipherAlgorithm, and MACAlgorithm may be added to. \*/

Dierks & Rescorla Standards Track [Page 17]

RFC 5246 TLS August 2008

struct {

ConnectionEnd entity;

PRFAlgorithm prf\_algorithm;

BulkCipherAlgorithm bulk\_cipher\_algorithm;

CipherType cipher\_type;

uint8 enc\_key\_length;

uint8 block\_length;

uint8 fixed\_iv\_length;

uint8 record\_iv\_length;

MACAlgorithm mac\_algorithm;

uint8 mac\_length;

uint8 mac\_key\_length;

CompressionMethod compression\_algorithm;

opaque master\_secret[48];

opaque client\_random[32];

opaque server\_random[32];

} SecurityParameters;

The record layer will use the security parameters to generate the

following six items (some of which are not required by all ciphers,

and are thus empty):

client write MAC key

server write MAC key

client write encryption key

server write encryption key

client write IV

server write IV

The client write parameters are used by the server when receiving and

processing records and vice versa. The algorithm used for generating

these items from the security parameters is described in Section 6.3.

Once the security parameters have been set and the keys have been

generated, the connection states can be instantiated by making them

the current states. These current states MUST be updated for each

record processed. Each connection state includes the following

elements:

compression state

The current state of the compression algorithm.

cipher state

The current state of the encryption algorithm. This will consist

of the scheduled key for that connection. For stream ciphers,

this will also contain whatever state information is necessary to

allow the stream to continue to encrypt or decrypt data.

Dierks & Rescorla Standards Track [Page 18]

RFC 5246 TLS August 2008

MAC key

The MAC key for this connection, as generated above.

sequence number

Each connection state contains a sequence number, which is

maintained separately for read and write states. The sequence

number MUST be set to zero whenever a connection state is made the

active state. Sequence numbers are of type uint64 and may not

exceed 2^64-1. Sequence numbers do not wrap. If a TLS

implementation would need to wrap a sequence number, it must

renegotiate instead. A sequence number is incremented after each

record: specifically, the first record transmitted under a

particular connection state MUST use sequence number 0.

6.2. Record Layer

The TLS record layer receives uninterpreted data from higher layers

in non-empty blocks of arbitrary size.

6.2.1. Fragmentation

The record layer fragments information blocks into TLSPlaintext

records carrying data in chunks of 2^14 bytes or less. Client

message boundaries are not preserved in the record layer (i.e.,

multiple client messages of the same ContentType MAY be coalesced

into a single TLSPlaintext record, or a single message MAY be

fragmented across several records).

struct {

uint8 major;

uint8 minor;

} ProtocolVersion;

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;

type

The higher-level protocol used to process the enclosed fragment.

Dierks & Rescorla Standards Track [Page 19]

RFC 5246 TLS August 2008

version

The version of the protocol being employed. This document

describes TLS Version 1.2, which uses the version { 3, 3 }. The

version value 3.3 is historical, deriving from the use of {3, 1}

for TLS 1.0. (See Appendix A.1.) Note that a client that

supports multiple versions of TLS may not know what version will

be employed before it receives the ServerHello. See Appendix E

for discussion about what record layer version number should be

employed for ClientHello.

length

The length (in bytes) of the following TLSPlaintext.fragment. The

length MUST NOT exceed 2^14.

fragment

The application data. This data is transparent and treated as an

independent block to be dealt with by the higher-level protocol

specified by the type field.

Implementations MUST NOT send zero-length fragments of Handshake,

Alert, or ChangeCipherSpec content types. Zero-length fragments of

Application data MAY be sent as they are potentially useful as a

traffic analysis countermeasure.

Note: Data of different TLS record layer content types MAY be

interleaved. Application data is generally of lower precedence for

transmission than other content types. However, records MUST be

delivered to the network in the same order as they are protected by

the record layer. Recipients MUST receive and process interleaved

application layer traffic during handshakes subsequent to the first

one on a connection.

6.2.2. Record Compression and Decompression

All records are compressed using the compression algorithm defined in

the current session state. There is always an active compression

algorithm; however, initially it is defined as

CompressionMethod.null. The compression algorithm translates a

TLSPlaintext structure into a TLSCompressed structure. Compression

functions are initialized with default state information whenever a

connection state is made active. [RFC3749] describes compression

algorithms for TLS.

Compression must be lossless and may not increase the content length

by more than 1024 bytes. If the decompression function encounters a

TLSCompressed.fragment that would decompress to a length in excess of

2^14 bytes, it MUST report a fatal decompression failure error.

Dierks & Rescorla Standards Track [Page 20]

RFC 5246 TLS August 2008

struct {

ContentType type; /\* same as TLSPlaintext.type \*/

ProtocolVersion version;/\* same as TLSPlaintext.version \*/

uint16 length;

opaque fragment[TLSCompressed.length];

} TLSCompressed;

length

The length (in bytes) of the following TLSCompressed.fragment.

The length MUST NOT exceed 2^14 + 1024.

fragment

The compressed form of TLSPlaintext.fragment.

Note: A CompressionMethod.null operation is an identity operation;

no fields are altered.

Implementation note: Decompression functions are responsible for

ensuring that messages cannot cause internal buffer overflows.

6.2.3. Record Payload Protection

The encryption and MAC functions translate a TLSCompressed

structure into a TLSCiphertext. The decryption functions reverse

the process. The MAC of the record also includes a sequence

number so that missing, extra, or repeated messages are

detectable.

struct {

ContentType type;

ProtocolVersion version;

uint16 length;

select (SecurityParameters.cipher\_type) {

case stream: GenericStreamCipher;

case block: GenericBlockCipher;

case aead: GenericAEADCipher;

} fragment;

} TLSCiphertext;

type

The type field is identical to TLSCompressed.type.

version

The version field is identical to TLSCompressed.version.

length

The length (in bytes) of the following TLSCiphertext.fragment.

The length MUST NOT exceed 2^14 + 2048.

Dierks & Rescorla Standards Track [Page 21]

RFC 5246 TLS August 2008

fragment

The encrypted form of TLSCompressed.fragment, with the MAC.

6.2.3.1. Null or Standard Stream Cipher

Stream ciphers (including BulkCipherAlgorithm.null; see Appendix A.6)

convert TLSCompressed.fragment structures to and from stream

TLSCiphertext.fragment structures.

stream-ciphered struct {

opaque content[TLSCompressed.length];

opaque MAC[SecurityParameters.mac\_length];

} GenericStreamCipher;

The MAC is generated as:

MAC(MAC\_write\_key, seq\_num +

TLSCompressed.type +

TLSCompressed.version +

TLSCompressed.length +

TLSCompressed.fragment);

where "+" denotes concatenation.

seq\_num

The sequence number for this record.

MAC

The MAC algorithm specified by SecurityParameters.mac\_algorithm.

Note that the MAC is computed before encryption. The stream cipher

encrypts the entire block, including the MAC. For stream ciphers

that do not use a synchronization vector (such as RC4), the stream

cipher state from the end of one record is simply used on the

subsequent packet. If the cipher suite is TLS\_NULL\_WITH\_NULL\_NULL,

encryption consists of the identity operation (i.e., the data is not

encrypted, and the MAC size is zero, implying that no MAC is used).

For both null and stream ciphers, TLSCiphertext.length is

TLSCompressed.length plus SecurityParameters.mac\_length.

6.2.3.2. CBC Block Cipher

For block ciphers (such as 3DES or AES), the encryption and MAC

functions convert TLSCompressed.fragment structures to and from block

TLSCiphertext.fragment structures.

Dierks & Rescorla Standards Track [Page 22]

RFC 5246 TLS August 2008

struct {

opaque IV[SecurityParameters.record\_iv\_length];

block-ciphered struct {

opaque content[TLSCompressed.length];

opaque MAC[SecurityParameters.mac\_length];

uint8 padding[GenericBlockCipher.padding\_length];

uint8 padding\_length;

};

} GenericBlockCipher;

The MAC is generated as described in Section 6.2.3.1.

IV

The Initialization Vector (IV) SHOULD be chosen at random, and

MUST be unpredictable. Note that in versions of TLS prior to 1.1,

there was no IV field, and the last ciphertext block of the

previous record (the "CBC residue") was used as the IV. This was

changed to prevent the attacks described in [CBCATT]. For block

ciphers, the IV length is of length

SecurityParameters.record\_iv\_length, which is equal to the

SecurityParameters.block\_size.

padding

Padding that is added to force the length of the plaintext to be

an integral multiple of the block cipher's block length. The

padding MAY be any length up to 255 bytes, as long as it results

in the TLSCiphertext.length being an integral multiple of the

block length. Lengths longer than necessary might be desirable to

frustrate attacks on a protocol that are based on analysis of the

lengths of exchanged messages. Each uint8 in the padding data

vector MUST be filled with the padding length value. The receiver

MUST check this padding and MUST use the bad\_record\_mac alert to

indicate padding errors.

padding\_length

The padding length MUST be such that the total size of the

GenericBlockCipher structure is a multiple of the cipher's block

length. Legal values range from zero to 255, inclusive. This

length specifies the length of the padding field exclusive of the

padding\_length field itself.

The encrypted data length (TLSCiphertext.length) is one more than the

sum of SecurityParameters.block\_length, TLSCompressed.length,

SecurityParameters.mac\_length, and padding\_length.

Example: If the block length is 8 bytes, the content length

(TLSCompressed.length) is 61 bytes, and the MAC length is 20 bytes,

then the length before padding is 82 bytes (this does not include the

Dierks & Rescorla Standards Track [Page 23]

RFC 5246 TLS August 2008

IV. Thus, the padding length modulo 8 must be equal to 6 in order to

make the total length an even multiple of 8 bytes (the block length).

The padding length can be 6, 14, 22, and so on, through 254. If the

padding length were the minimum necessary, 6, the padding would be 6

bytes, each containing the value 6. Thus, the last 8 octets of the

GenericBlockCipher before block encryption would be xx 06 06 06 06 06

06 06, where xx is the last octet of the MAC.

Note: With block ciphers in CBC mode (Cipher Block Chaining), it is

critical that the entire plaintext of the record be known before any

ciphertext is transmitted. Otherwise, it is possible for the

attacker to mount the attack described in [CBCATT].

Implementation note: Canvel et al. [CBCTIME] have demonstrated a

timing attack on CBC padding based on the time required to compute

the MAC. In order to defend against this attack, implementations

MUST ensure that record processing time is essentially the same

whether or not the padding is correct. In general, the best way to

do this is to compute the MAC even if the padding is incorrect, and

only then reject the packet. For instance, if the pad appears to be

incorrect, the implementation might assume a zero-length pad and then

compute the MAC. This leaves a small timing channel, since MAC

performance depends to some extent on the size of the data fragment,

but it is not believed to be large enough to be exploitable, due to

the large block size of existing MACs and the small size of the

timing signal.

6.2.3.3. AEAD Ciphers

For AEAD [AEAD] ciphers (such as [CCM] or [GCM]), the AEAD function

converts TLSCompressed.fragment structures to and from AEAD

TLSCiphertext.fragment structures.

struct {

opaque nonce\_explicit[SecurityParameters.record\_iv\_length];

aead-ciphered struct {

opaque content[TLSCompressed.length];

};

} GenericAEADCipher;

AEAD ciphers take as input a single key, a nonce, a plaintext, and

"additional data" to be included in the authentication check, as

described in Section 2.1 of [AEAD]. The key is either the

client\_write\_key or the server\_write\_key. No MAC key is used.

Each AEAD cipher suite MUST specify how the nonce supplied to the

AEAD operation is constructed, and what is the length of the

GenericAEADCipher.nonce\_explicit part. In many cases, it is

Dierks & Rescorla Standards Track [Page 24]

RFC 5246 TLS August 2008

appropriate to use the partially implicit nonce technique described

in Section 3.2.1 of [AEAD]; with record\_iv\_length being the length of

the explicit part. In this case, the implicit part SHOULD be derived

from key\_block as client\_write\_iv and server\_write\_iv (as described

in Section 6.3), and the explicit part is included in

GenericAEAEDCipher.nonce\_explicit.

The plaintext is the TLSCompressed.fragment.

The additional authenticated data, which we denote as

additional\_data, is defined as follows:

additional\_data = seq\_num + TLSCompressed.type +

TLSCompressed.version + TLSCompressed.length;

where "+" denotes concatenation.

The aead\_output consists of the ciphertext output by the AEAD

encryption operation. The length will generally be larger than

TLSCompressed.length, but by an amount that varies with the AEAD

cipher. Since the ciphers might incorporate padding, the amount of

overhead could vary with different TLSCompressed.length values. Each

AEAD cipher MUST NOT produce an expansion of greater than 1024 bytes.

Symbolically,

AEADEncrypted = AEAD-Encrypt(write\_key, nonce, plaintext,

additional\_data)

In order to decrypt and verify, the cipher takes as input the key,

nonce, the "additional\_data", and the AEADEncrypted value. The

output is either the plaintext or an error indicating that the

decryption failed. There is no separate integrity check. That is:

TLSCompressed.fragment = AEAD-Decrypt(write\_key, nonce,

AEADEncrypted,

additional\_data)

If the decryption fails, a fatal bad\_record\_mac alert MUST be

generated.

6.3. Key Calculation

The Record Protocol requires an algorithm to generate keys required

by the current connection state (see Appendix A.6) from the security

parameters provided by the handshake protocol.

Dierks & Rescorla Standards Track [Page 25]

RFC 5246 TLS August 2008

The master secret is expanded into a sequence of secure bytes, which

is then split to a client write MAC key, a server write MAC key, a

client write encryption key, and a server write encryption key. Each

of these is generated from the byte sequence in that order. Unused

values are empty. Some AEAD ciphers may additionally require a

client write IV and a server write IV (see Section 6.2.3.3).

When keys and MAC keys are generated, the master secret is used as an

entropy source.

To generate the key material, compute

key\_block = PRF(SecurityParameters.master\_secret,

"key expansion",

SecurityParameters.server\_random +

SecurityParameters.client\_random);

until enough output has been generated. Then, the key\_block is

partitioned as follows:

client\_write\_MAC\_key[SecurityParameters.mac\_key\_length]

server\_write\_MAC\_key[SecurityParameters.mac\_key\_length]

client\_write\_key[SecurityParameters.enc\_key\_length]

server\_write\_key[SecurityParameters.enc\_key\_length]

client\_write\_IV[SecurityParameters.fixed\_iv\_length]

server\_write\_IV[SecurityParameters.fixed\_iv\_length]

Currently, the client\_write\_IV and server\_write\_IV are only generated

for implicit nonce techniques as described in Section 3.2.1 of

[AEAD].

Implementation note: The currently defined cipher suite which

requires the most material is AES\_256\_CBC\_SHA256. It requires 2 x 32

byte keys and 2 x 32 byte MAC keys, for a total 128 bytes of key

material.

7. The TLS Handshaking Protocols

TLS has three subprotocols that are used to allow peers to agree upon

security parameters for the record layer, to authenticate themselves,

to instantiate negotiated security parameters, and to report error

conditions to each other.

The Handshake Protocol is responsible for negotiating a session,

which consists of the following items:

Dierks & Rescorla Standards Track [Page 26]

RFC 5246 TLS August 2008

session identifier

An arbitrary byte sequence chosen by the server to identify an

active or resumable session state.

peer certificate

X509v3 [PKIX] certificate of the peer. This element of the state

may be null.

compression method

The algorithm used to compress data prior to encryption.

cipher spec

Specifies the pseudorandom function (PRF) used to generate keying

material, the bulk data encryption algorithm (such as null, AES,

etc.) and the MAC algorithm (such as HMAC-SHA1). It also defines

cryptographic attributes such as the mac\_length. (See Appendix

A.6 for formal definition.)

master secret

48-byte secret shared between the client and server.

is resumable

A flag indicating whether the session can be used to initiate new

connections.

These items are then used to create security parameters for use by

the record layer when protecting application data. Many connections

can be instantiated using the same session through the resumption

feature of the TLS Handshake Protocol.

7.1. Change Cipher Spec Protocol

The change cipher spec protocol exists to signal transitions in

ciphering strategies. The protocol consists of a single message,

which is encrypted and compressed under the current (not the pending)

connection state. The message consists of a single byte of value 1.

struct {

enum { change\_cipher\_spec(1), (255) } type;

} ChangeCipherSpec;

The ChangeCipherSpec message is sent by both the client and the

server to notify the receiving party that subsequent records will be

protected under the newly negotiated CipherSpec and keys. Reception

of this message causes the receiver to instruct the record layer to

immediately copy the read pending state into the read current state.

Immediately after sending this message, the sender MUST instruct the

record layer to make the write pending state the write active state.

Dierks & Rescorla Standards Track [Page 27]

RFC 5246 TLS August 2008

(See Section 6.1.) The ChangeCipherSpec message is sent during the

handshake after the security parameters have been agreed upon, but

before the verifying Finished message is sent.

Note: If a rehandshake occurs while data is flowing on a connection,

the communicating parties may continue to send data using the old

CipherSpec. However, once the ChangeCipherSpec has been sent, the

new CipherSpec MUST be used. The first side to send the

ChangeCipherSpec does not know that the other side has finished

computing the new keying material (e.g., if it has to perform a

time-consuming public key operation). Thus, a small window of time,

during which the recipient must buffer the data, MAY exist. In

practice, with modern machines this interval is likely to be fairly

short.

7.2. Alert Protocol

One of the content types supported by the TLS record layer is the

alert type. Alert messages convey the severity of the message

(warning or fatal) and a description of the alert. Alert messages

with a level of fatal result in the immediate termination of the

connection. In this case, other connections corresponding to the

session may continue, but the session identifier MUST be invalidated,

preventing the failed session from being used to establish new

connections. Like other messages, alert messages are encrypted and

compressed, as specified by the current connection state.

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

enum {

close\_notify(0),

unexpected\_message(10),

bad\_record\_mac(20),

decryption\_failed\_RESERVED(21),

record\_overflow(22),

decompression\_failure(30),

handshake\_failure(40),

no\_certificate\_RESERVED(41),

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),

Dierks & Rescorla Standards Track [Page 28]

RFC 5246 TLS August 2008

export\_restriction\_RESERVED(60),

protocol\_version(70),

insufficient\_security(71),

internal\_error(80),

user\_canceled(90),

no\_renegotiation(100),

unsupported\_extension(110),

(255)

} AlertDescription;

struct {

AlertLevel level;

AlertDescription description;

} Alert;

7.2.1. Closure Alerts

The client and the server must share knowledge that the connection is

ending in order to avoid a truncation attack. Either party may

initiate the exchange of closing messages.

close\_notify

This message notifies the recipient that the sender will not send

any more messages on this connection. Note that as of TLS 1.1,

failure to properly close a connection no longer requires that a

session not be resumed. This is a change from TLS 1.0 to conform

with widespread implementation practice.

Either party may initiate a close by sending a close\_notify alert.

Any data received after a closure alert is ignored.

Unless some other fatal alert has been transmitted, each party is

required to send a close\_notify alert before closing the write side

of the connection. The other party MUST respond with a close\_notify

alert of its own and close down the connection immediately,

discarding any pending writes. It is not required for the initiator

of the close to wait for the responding close\_notify alert before

closing the read side of the connection.

If the application protocol using TLS provides that any data may be

carried over the underlying transport after the TLS connection is

closed, the TLS implementation must receive the responding

close\_notify alert before indicating to the application layer that

the TLS connection has ended. If the application protocol will not

transfer any additional data, but will only close the underlying

transport connection, then the implementation MAY choose to close the

transport without waiting for the responding close\_notify. No part

Dierks & Rescorla Standards Track [Page 29]

RFC 5246 TLS August 2008

of this standard should be taken to dictate the manner in which a

usage profile for TLS manages its data transport, including when

connections are opened or closed.

Note: It is assumed that closing a connection reliably delivers

pending data before destroying the transport.

7.2.2. Error Alerts

Error handling in the TLS Handshake protocol is very simple. When an

error is detected, the detecting party sends a message to the other

party. Upon transmission or receipt of a fatal alert message, both

parties immediately close the connection. Servers and clients MUST

forget any session-identifiers, keys, and secrets associated with a

failed connection. Thus, any connection terminated with a fatal

alert MUST NOT be resumed.

Whenever an implementation encounters a condition which is defined as

a fatal alert, it MUST send the appropriate alert prior to closing

the connection. For all errors where an alert level is not

explicitly specified, the sending party MAY determine at its

discretion whether to treat this as a fatal error or not. If the

implementation chooses to send an alert but intends to close the

connection immediately afterwards, it MUST send that alert at the

fatal alert level.

If an alert with a level of warning is sent and received, generally

the connection can continue normally. If the receiving party decides

not to proceed with the connection (e.g., after having received a

no\_renegotiation alert that it is not willing to accept), it SHOULD

send a fatal alert to terminate the connection. Given this, the

sending party cannot, in general, know how the receiving party will

behave. Therefore, warning alerts are not very useful when the

sending party wants to continue the connection, and thus are

sometimes omitted. For example, if a peer decides to accept an

expired certificate (perhaps after confirming this with the user) and

wants to continue the connection, it would not generally send a

certificate\_expired alert.

The following error alerts are defined:

unexpected\_message

An inappropriate message was received. This alert is always fatal

and should never be observed in communication between proper

implementations.

Dierks & Rescorla Standards Track [Page 30]

RFC 5246 TLS August 2008

bad\_record\_mac

This alert is returned if a record is received with an incorrect

MAC. This alert also MUST be returned if an alert is sent because

a TLSCiphertext decrypted in an invalid way: either it wasn't an

even multiple of the block length, or its padding values, when

checked, weren't correct. This message is always fatal and should

never be observed in communication between proper implementations

(except when messages were corrupted in the network).

decryption\_failed\_RESERVED

This alert was used in some earlier versions of TLS, and may have

permitted certain attacks against the CBC mode [CBCATT]. It MUST

NOT be sent by compliant implementations.

record\_overflow

A TLSCiphertext record was received that 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 and

should never be observed in communication between proper

implementations (except when messages were corrupted in the

network).

decompression\_failure

The decompression function received improper input (e.g., data

that would expand to excessive length). This message is always

fatal and should never be observed in communication between proper

implementations.

handshake\_failure

Reception of a handshake\_failure alert message indicates that the

sender was unable to negotiate an acceptable set of security

parameters given the options available. This is a fatal error.

no\_certificate\_RESERVED

This alert was used in SSLv3 but not any version of TLS. It MUST

NOT be sent by compliant implementations.

bad\_certificate

A certificate was corrupt, contained signatures that did not

verify correctly, etc.

unsupported\_certificate

A certificate was of an unsupported type.

certificate\_revoked

A certificate was revoked by its signer.

Dierks & Rescorla Standards Track [Page 31]

RFC 5246 TLS August 2008

certificate\_expired

A certificate has expired or is not currently valid.

certificate\_unknown

Some other (unspecified) issue arose in processing the

certificate, rendering it unacceptable.

illegal\_parameter

A field in the handshake was out of range or inconsistent with

other fields. This message is always fatal.

unknown\_ca

A valid certificate chain or partial chain was received, but the

certificate was not accepted because the CA certificate could not

be located or couldn't be matched with a known, trusted CA. This

message is always fatal.

access\_denied

A valid certificate was received, but when access control was

applied, the sender decided not to 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 and should never be observed in

communication between proper implementations (except when messages

were corrupted in the network).

decrypt\_error

A handshake cryptographic operation failed, including being unable

to correctly verify a signature or validate a Finished message.

This message is always fatal.

export\_restriction\_RESERVED

This alert was used in some earlier versions of TLS. It MUST NOT

be sent by compliant implementations.

protocol\_version

The protocol version the client has attempted to negotiate is

recognized but not supported. (For example, old protocol versions

might be avoided for security reasons.) This message is always

fatal.

Dierks & Rescorla Standards Track [Page 32]

RFC 5246 TLS August 2008

insufficient\_security

Returned instead of handshake\_failure when a negotiation has

failed specifically because the server requires ciphers more

secure than those supported by the client. This message is always

fatal.

internal\_error

An internal error unrelated to the peer or the correctness of the

protocol (such as a memory allocation failure) makes it impossible

to continue. This message is always fatal.

user\_canceled

This handshake is being canceled for some reason 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 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 is

where a server has spawned a process to satisfy a request; the

process might receive security parameters (key length,

authentication, etc.) at startup, and it might be difficult to

communicate changes to these parameters after that point. This

message is always a warning.

unsupported\_extension

sent by clients that receive an extended server hello containing

an extension that they did not put in the corresponding client

hello. This message is always fatal.

New Alert values are assigned by IANA as described in Section 12.

7.3. Handshake Protocol Overview

The cryptographic parameters of the session state are produced by the

TLS Handshake Protocol, which operates on top of the TLS record

layer. When a TLS client and server first start communicating, they

agree on a protocol version, select cryptographic algorithms,

optionally authenticate each other, and use public-key encryption

techniques to generate shared secrets.

Dierks & Rescorla Standards Track [Page 33]

RFC 5246 TLS August 2008

The TLS Handshake Protocol involves the following steps:

- Exchange hello messages to agree on algorithms, exchange random

values, and check for session resumption.

- Exchange the necessary cryptographic parameters to allow the

client and server to agree on a premaster secret.

- Exchange certificates and cryptographic information to allow the

client and server to authenticate themselves.

- Generate a master secret from the premaster secret and exchanged

random values.

- Provide security parameters to the record layer.

- Allow the client and server to verify that their peer has

calculated the same security parameters and that the handshake

occurred without tampering by an attacker.

Note that higher layers should not be overly reliant on whether TLS

always negotiates the strongest possible connection between two

peers. There are a number of ways in which a man-in-the-middle

attacker can attempt to make two entities drop down to the least

secure method they support. The protocol has been designed to

minimize this risk, but there are still attacks available: for

example, an attacker could block access to the port a secure service

runs on, or attempt to get the peers to negotiate an unauthenticated

connection. The fundamental rule is that higher levels must be

cognizant of what their security requirements are and never transmit

information over a channel less secure than what they require. The

TLS protocol is secure in that any cipher suite offers its promised

level of security: if you negotiate 3DES with a 1024-bit RSA key

exchange with a host whose certificate you have verified, you can

expect to be that secure.

These goals are achieved by the handshake protocol, which can be

summarized as follows: The client sends a ClientHello message to

which the server must respond with a ServerHello message, or else a

fatal error will occur and the connection will fail. The ClientHello

and ServerHello are used to establish security enhancement

capabilities between client and server. The ClientHello and

ServerHello establish the following attributes: Protocol Version,

Session ID, Cipher Suite, and Compression Method. Additionally, two

random values are generated and exchanged: ClientHello.random and

ServerHello.random.

Dierks & Rescorla Standards Track [Page 34]

RFC 5246 TLS August 2008

The actual key exchange uses up to four messages: the server

Certificate, the ServerKeyExchange, the client Certificate, and the

ClientKeyExchange. New key exchange methods can be created by

specifying a format for these messages and by defining the use of the

messages to allow the client and server to agree upon a shared

secret. This secret MUST be quite long; currently defined key

exchange methods exchange secrets that range from 46 bytes upwards.

Following the hello messages, the server will send its certificate in

a Certificate message if it is to be authenticated. Additionally, a

ServerKeyExchange message may be sent, if it is required (e.g., if

the server has no certificate, or if its certificate is for signing

only). If the server is authenticated, it may request a certificate

from the client, if that is appropriate to the cipher suite selected.

Next, the server will send the ServerHelloDone message, indicating

that the hello-message phase of the handshake is complete. The

server will then wait for a client response. If the server has sent

a CertificateRequest message, the client MUST send the Certificate

message. The ClientKeyExchange message is now sent, and the content

of that message will depend on the public key algorithm selected

between the ClientHello and the ServerHello. If the client has sent

a certificate with signing ability, a digitally-signed

CertificateVerify message is sent to explicitly verify possession of

the private key in the certificate.

At this point, a ChangeCipherSpec message is sent by the client, and

the client copies the pending Cipher Spec into the current Cipher

Spec. The client then immediately sends the Finished message under

the new algorithms, keys, and secrets. In response, the server will

send its own ChangeCipherSpec message, transfer the pending to the

current Cipher Spec, and send its Finished message under the new

Cipher Spec. At this point, the handshake is complete, and the

client and server may begin to exchange application layer data. (See

flow chart below.) Application data MUST NOT be sent prior to the

completion of the first handshake (before a cipher suite other than

TLS\_NULL\_WITH\_NULL\_NULL is established).

Dierks & Rescorla Standards Track [Page 35]

RFC 5246 TLS August 2008

Client Server

ClientHello -------->

ServerHello

Certificate\*

ServerKeyExchange\*

CertificateRequest\*

<-------- ServerHelloDone

Certificate\*

ClientKeyExchange

CertificateVerify\*

[ChangeCipherSpec]

Finished -------->

[ChangeCipherSpec]

<-------- Finished

Application Data <-------> Application Data

Figure 1. Message flow for a full handshake

\* Indicates optional or situation-dependent messages that are not

always sent.

Note: To help avoid pipeline stalls, ChangeCipherSpec is an

independent TLS protocol content type, and is not actually a TLS

handshake message.

When the client and server decide to resume a previous session or

duplicate an existing session (instead of negotiating new security

parameters), the message flow is as follows:

The client sends a ClientHello using the Session ID of the session to

be resumed. The server then checks its session cache for a match.

If a match is found, and the server is willing to re-establish the

connection under the specified session state, it will send a

ServerHello with the same Session ID value. At this point, both

client and server MUST send ChangeCipherSpec messages and proceed

directly to Finished messages. Once the re-establishment is

complete, the client and server MAY begin to exchange application

layer data. (See flow chart below.) 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.

Dierks & Rescorla Standards Track [Page 36]

RFC 5246 TLS August 2008

Client Server

ClientHello -------->

ServerHello

[ChangeCipherSpec]

<-------- Finished

[ChangeCipherSpec]

Finished -------->

Application Data <-------> Application Data

Figure 2. Message flow for an abbreviated handshake

The contents and significance of each message will be presented in

detail in the following sections.

7.4. Handshake Protocol

The TLS Handshake Protocol is one of the defined higher-level clients

of the TLS Record Protocol. This protocol is used to negotiate the

secure attributes of a session. Handshake messages are supplied to

the TLS record layer, where they are encapsulated within one or more

TLSPlaintext structures, which are processed and transmitted as

specified by the current active session state.

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; /\* handshake type \*/

uint24 length; /\* bytes in message \*/

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;

Dierks & Rescorla Standards Track [Page 37]

RFC 5246 TLS August 2008

The handshake protocol messages are presented below in the order they

MUST be sent; sending handshake messages in an unexpected order

results in a fatal error. Unneeded handshake messages can be

omitted, however. Note one exception to the ordering: the

Certificate message is used twice in the handshake (from server to

client, then from client to server), but described only in its first

position. The one message that is not bound by these ordering rules

is the HelloRequest message, which can be sent at any time, but which

SHOULD be ignored by the client if it arrives in the middle of a

handshake.

New handshake message types are assigned by IANA as described in

Section 12.

7.4.1. Hello Messages

The hello phase messages are used to exchange security enhancement

capabilities between the client and server. When a new session

begins, the record layer's connection state encryption, hash, and

compression algorithms are initialized to null. The current

connection state is used for renegotiation messages.

7.4.1.1. Hello Request

When this message will be sent:

The HelloRequest message MAY be sent by the server at any time.

Meaning of this message:

HelloRequest is a simple notification that the client should begin

the negotiation process anew. In response, the client should send

a ClientHello message when convenient. This message is not

intended to establish which side is the client or server but

merely to initiate a new negotiation. Servers SHOULD NOT send a

HelloRequest immediately upon the client's initial connection. It

is the client's job to send a ClientHello at that time.

This message will be ignored by the client if the client is

currently negotiating a session. This message MAY be ignored by

the client if it does not wish to renegotiate a session, or the

client may, if it wishes, respond with a no\_renegotiation alert.

Since handshake messages are intended to have transmission

precedence over application data, it is expected that the

negotiation will begin before no more than a few records are

received from the client. If the server sends a HelloRequest but

does not receive a ClientHello in response, it may close the

connection with a fatal alert.

Dierks & Rescorla Standards Track [Page 38]

RFC 5246 TLS August 2008

After sending a HelloRequest, servers SHOULD NOT repeat the

request until the subsequent handshake negotiation is complete.

Structure of this message:

struct { } HelloRequest;

This message MUST NOT be included in the message hashes that are

maintained throughout the handshake and used in the Finished messages

and the certificate verify message.

7.4.1.2. Client Hello

When this message will be sent:

When a client first connects to a server, it is required to send

the ClientHello as its first message. The client can also send a

ClientHello in response to a HelloRequest or on its own initiative

in order to renegotiate the security parameters in an existing

connection.

Structure of this message:

The ClientHello message includes a random structure, which is used

later in the protocol.

struct {

uint32 gmt\_unix\_time;

opaque random\_bytes[28];

} Random;

gmt\_unix\_time

The current time and date in standard UNIX 32-bit format

(seconds since the midnight starting Jan 1, 1970, UTC, ignoring

leap seconds) according to the sender's internal clock. Clocks

are not required to be set correctly by the basic TLS protocol;

higher-level or application protocols may define additional

requirements. Note that, for historical reasons, the data

element is named using GMT, the predecessor of the current

worldwide time base, UTC.

random\_bytes

28 bytes generated by a secure random number generator.

The ClientHello message includes a variable-length session

identifier. If not empty, the value identifies a session between the

same client and server whose security parameters the client wishes to

reuse. The session identifier MAY be from an earlier connection,

Dierks & Rescorla Standards Track [Page 39]

RFC 5246 TLS August 2008

this connection, or from another currently active connection. The

second option is useful if the client only wishes to update the

random structures and derived values of a connection, and the third

option makes it possible to establish several independent secure

connections without repeating the full handshake protocol. These

independent connections may occur sequentially or simultaneously; a

SessionID becomes valid when the handshake negotiating it completes

with the exchange of Finished messages and persists until it is

removed due to aging or because a fatal error was encountered on a

connection associated with the session. The actual contents of the

SessionID are defined by the server.

opaque SessionID<0..32>;

Warning: Because the SessionID is transmitted without encryption or

immediate MAC protection, servers MUST NOT place confidential

information in session identifiers or let the contents of fake

session identifiers cause any breach of security. (Note that the

content of the handshake as a whole, including the SessionID, is

protected by the Finished messages exchanged at the end of the

handshake.)

The cipher suite list, passed from the client to the server in the

ClientHello message, contains the combinations of cryptographic

algorithms supported by the client in order of the client's

preference (favorite choice first). Each cipher suite defines a key

exchange algorithm, a bulk encryption algorithm (including secret key

length), a MAC algorithm, and a PRF. The server will select a cipher

suite or, if no acceptable choices are presented, return a handshake

failure alert and close the connection. If the list contains cipher

suites the server does not recognize, support, or wish to use, the

server MUST ignore those cipher suites, and process the remaining

ones as usual.

uint8 CipherSuite[2]; /\* Cryptographic suite selector \*/

The ClientHello includes a list of compression algorithms supported

by the client, ordered according to the client's preference.

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

Dierks & Rescorla Standards Track [Page 40]

RFC 5246 TLS August 2008

struct {

ProtocolVersion client\_version;

Random random;

SessionID session\_id;

CipherSuite cipher\_suites<2..2^16-2>;

CompressionMethod compression\_methods<1..2^8-1>;

select (extensions\_present) {

case false:

struct {};

case true:

Extension extensions<0..2^16-1>;

};

} ClientHello;

TLS allows extensions to follow the compression\_methods field in an

extensions block. The presence of extensions can be detected by

determining whether there are bytes following the compression\_methods

at the end of the ClientHello. Note that this method of detecting

optional data differs from the normal TLS method of having a

variable-length field, but it is used for compatibility with TLS

before extensions were defined.

client\_version

The version of the TLS protocol by which the client wishes to

communicate during this session. This SHOULD be the latest

(highest valued) version supported by the client. For this

version of the specification, the version will be 3.3 (see

Appendix E for details about backward compatibility).

random

A client-generated random structure.

session\_id

The ID of a session the client wishes to use for this connection.

This field is empty if no session\_id is available, or if the

client wishes to generate new security parameters.

cipher\_suites

This is a list of the cryptographic options supported by the

client, with the client's first preference first. If the

session\_id field is not empty (implying a session resumption

request), this vector MUST include at least the cipher\_suite from

that session. Values are defined in Appendix A.5.

compression\_methods

This is a list of the compression methods supported by the client,

sorted by client preference. If the session\_id field is not empty

(implying a session resumption request), it MUST include the

Dierks & Rescorla Standards Track [Page 41]

RFC 5246 TLS August 2008

compression\_method from that session. This vector MUST contain,

and all implementations MUST support, CompressionMethod.null.

Thus, a client and server will always be able to agree on a

compression method.

extensions

Clients MAY request extended functionality from servers by sending

data in the extensions field. The actual "Extension" format is

defined in Section 7.4.1.4.

In the event that a client requests additional functionality using

extensions, and this functionality is not supplied by the server, the

client MAY abort the handshake. A server MUST accept ClientHello

messages both with and without the extensions field, and (as for all

other messages) it MUST check that the amount of data in the message

precisely matches one of these formats; if not, then it MUST send a

fatal "decode\_error" alert.

After sending the ClientHello message, the client waits for a

ServerHello message. Any handshake message returned by the server,

except for a HelloRequest, is treated as a fatal error.

7.4.1.3. Server Hello

When this message will be sent:

The server will send this message in response to a ClientHello

message when it was able to find an acceptable set of algorithms.

If it cannot find such a match, it will respond with a handshake

failure alert.

Structure of this message:

struct {

ProtocolVersion server\_version;

Random random;

SessionID session\_id;

CipherSuite cipher\_suite;

CompressionMethod compression\_method;

select (extensions\_present) {

case false:

struct {};

case true:

Extension extensions<0..2^16-1>;

};

} ServerHello;

Dierks & Rescorla Standards Track [Page 42]

RFC 5246 TLS August 2008

The presence of extensions can be detected by determining whether

there are bytes following the compression\_method field at the end of

the ServerHello.

server\_version

This field will contain the lower of that suggested by the client

in the client hello and the highest supported by the server. For

this version of the specification, the version is 3.3. (See

Appendix E for details about backward compatibility.)

random

This structure is generated by the server and MUST be

independently generated from the ClientHello.random.

session\_id

This is the identity of the session corresponding to this

connection. If the ClientHello.session\_id was non-empty, the

server will look in its session cache for a match. If a match is

found and the server is willing to establish the new connection

using the specified session state, the server will respond with

the same value as was supplied by the client. This indicates a

resumed session and dictates that the parties must proceed

directly to the Finished messages. Otherwise, this field will

contain a different value identifying the new session. The server

may return an empty session\_id to indicate that the session will

not be cached and therefore cannot be resumed. If a session is

resumed, it must be resumed using the same cipher suite it was

originally negotiated with. Note that there is no requirement

that the server resume any session even if it had formerly

provided a session\_id. Clients MUST be prepared to do a full

negotiation -- including negotiating new cipher suites -- during

any handshake.

cipher\_suite

The single cipher suite selected by the server from the list in

ClientHello.cipher\_suites. For resumed sessions, this field is

the value from the state of the session being resumed.

compression\_method

The single compression algorithm selected by the server from the

list in ClientHello.compression\_methods. For resumed sessions,

this field is the value from the resumed session state.

extensions

A list of extensions. Note that only extensions offered by the

client can appear in the server's list.

Dierks & Rescorla Standards Track [Page 43]

RFC 5246 TLS August 2008

7.4.1.4. Hello Extensions

The extension format is:

struct {

ExtensionType extension\_type;

opaque extension\_data<0..2^16-1>;

} Extension;

enum {

signature\_algorithms(13), (65535)

} ExtensionType;

Here:

- "extension\_type" identifies the particular extension type.

- "extension\_data" contains information specific to the particular

extension type.

The initial set of extensions is defined in a companion document

[TLSEXT]. The list of extension types is maintained by IANA as

described in Section 12.

An extension type MUST NOT appear in the ServerHello unless the same

extension type appeared in the corresponding ClientHello. If a

client receives an extension type in ServerHello that it did not

request in the associated ClientHello, it MUST abort the handshake

with an unsupported\_extension fatal alert.

Nonetheless, "server-oriented" extensions may be provided in the

future within this framework. Such an extension (say, of type x)

would require the client to first send an extension of type x in a

ClientHello with empty extension\_data to indicate that it supports

the extension type. In this case, the client is offering the

capability to understand the extension type, and the server is taking

the client up on its offer.

When multiple extensions of different types are present in the

ClientHello or ServerHello messages, the extensions MAY appear in any

order. There MUST NOT be more than one extension of the same type.

Finally, note that extensions can be sent both when starting a new

session and when requesting session resumption. Indeed, a client

that requests session resumption does not in general know whether the

server will accept this request, and therefore it SHOULD send the

same extensions as it would send if it were not attempting

resumption.

Dierks & Rescorla Standards Track [Page 44]

RFC 5246 TLS August 2008

In general, the specification of each extension type needs to

describe the effect of the extension both during full handshake and

session resumption. Most current TLS extensions are relevant only

when a session is initiated: when an older session is resumed, the

server does not process these extensions in Client Hello, and does

not include them in Server Hello. However, some extensions may

specify different behavior during session resumption.

There are subtle (and not so subtle) interactions that may occur in

this protocol between new features and existing features which may

result in a significant reduction in overall security. The following

considerations should be taken into account when designing new

extensions:

- Some cases where a server does not agree to an extension are error

conditions, and some are simply refusals to support particular

features. In general, error alerts should be used for the former,

and a field in the server extension response for the latter.

- Extensions should, as far as possible, be designed to prevent any

attack that forces use (or non-use) of a particular feature by

manipulation of handshake messages. This principle should be

followed regardless of whether the feature is believed to cause a

security problem.

Often the fact that the extension fields are included in the

inputs to the Finished message hashes will be sufficient, but

extreme care is needed when the extension changes the meaning of

messages sent in the handshake phase. Designers and implementors

should be aware of the fact that until the handshake has been

authenticated, active attackers can modify messages and insert,

remove, or replace extensions.

- It would be technically possible to use extensions to change major

aspects of the design of TLS; for example the design of cipher

suite negotiation. This is not recommended; it would be more

appropriate to define a new version of TLS -- particularly since

the TLS handshake algorithms have specific protection against

version rollback attacks based on the version number, and the

possibility of version rollback should be a significant

consideration in any major design change.

7.4.1.4.1. Signature Algorithms

The client uses the "signature\_algorithms" extension to indicate to

the server which signature/hash algorithm pairs may be used in

digital signatures. The "extension\_data" field of this extension

contains a "supported\_signature\_algorithms" value.

Dierks & Rescorla Standards Track [Page 45]

RFC 5246 TLS August 2008

enum {

none(0), md5(1), sha1(2), sha224(3), sha256(4), sha384(5),

sha512(6), (255)

} HashAlgorithm;

enum { anonymous(0), rsa(1), dsa(2), ecdsa(3), (255) }

SignatureAlgorithm;

struct {

HashAlgorithm hash;

SignatureAlgorithm signature;

} SignatureAndHashAlgorithm;

SignatureAndHashAlgorithm

supported\_signature\_algorithms<2..2^16-2>;

Each SignatureAndHashAlgorithm value lists a single hash/signature

pair that the client is willing to verify. The values are indicated

in descending order of preference.

Note: Because not all signature algorithms and hash algorithms may be

accepted by an implementation (e.g., DSA with SHA-1, but not

SHA-256), algorithms here are listed in pairs.

hash

This field indicates the hash algorithm which may be used. The

values indicate support for unhashed data, MD5 [MD5], SHA-1,

SHA-224, SHA-256, SHA-384, and SHA-512 [SHS], respectively. The

"none" value is provided for future extensibility, in case of a

signature algorithm which does not require hashing before signing.

signature

This field indicates the signature algorithm that may be used.

The values indicate anonymous signatures, RSASSA-PKCS1-v1\_5

[PKCS1] and DSA [DSS], and ECDSA [ECDSA], respectively. The

"anonymous" value is meaningless in this context but used in

Section 7.4.3. It MUST NOT appear in this extension.

The semantics of this extension are somewhat complicated because the

cipher suite indicates permissible signature algorithms but not hash

algorithms. Sections 7.4.2 and 7.4.3 describe the appropriate rules.

If the client supports only the default hash and signature algorithms

(listed in this section), it MAY omit the signature\_algorithms

extension. If the client does not support the default algorithms, or

supports other hash and signature algorithms (and it is willing to

use them for verifying messages sent by the server, i.e., server

certificates and server key exchange), it MUST send the

Dierks & Rescorla Standards Track [Page 46]

RFC 5246 TLS August 2008

signature\_algorithms extension, listing the algorithms it is willing

to accept.

If the client does not send the signature\_algorithms extension, the

server MUST do the following:

- If the negotiated key exchange algorithm is one of (RSA, DHE\_RSA,

DH\_RSA, RSA\_PSK, ECDH\_RSA, ECDHE\_RSA), behave as if client had

sent the value {sha1,rsa}.

- If the negotiated key exchange algorithm is one of (DHE\_DSS,

DH\_DSS), behave as if the client had sent the value {sha1,dsa}.

- If the negotiated key exchange algorithm is one of (ECDH\_ECDSA,

ECDHE\_ECDSA), behave as if the client had sent value {sha1,ecdsa}.

Note: this is a change from TLS 1.1 where there are no explicit

rules, but as a practical matter one can assume that the peer

supports MD5 and SHA-1.

Note: this extension is not meaningful for TLS versions prior to 1.2.

Clients MUST NOT offer it if they are offering prior versions.

However, even if clients do offer it, the rules specified in [TLSEXT]

require servers to ignore extensions they do not understand.

Servers MUST NOT send this extension. TLS servers MUST support

receiving this extension.

When performing session resumption, this extension is not included in

Server Hello, and the server ignores the extension in Client Hello

(if present).

7.4.2. Server Certificate

When this message will be sent:

The server MUST send a Certificate message whenever the agreed-

upon key exchange method uses certificates for authentication

(this includes all key exchange methods defined in this document

except DH\_anon). This message will always immediately follow the

ServerHello message.

Meaning of this message:

This message conveys the server's certificate chain to the client.

The certificate MUST be appropriate for the negotiated cipher

suite's key exchange algorithm and any negotiated extensions.

Dierks & Rescorla Standards Track [Page 47]

RFC 5246 TLS August 2008

Structure of this message:

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

struct {

ASN.1Cert certificate\_list<0..2^24-1>;

} Certificate;

certificate\_list

This is a sequence (chain) of certificates. The sender's

certificate MUST come first in the list. Each following

certificate MUST directly certify the one preceding it. Because

certificate validation requires that root keys be distributed

independently, the self-signed certificate that specifies the root

certificate authority MAY be omitted from the chain, under the

assumption that the remote end must already possess it in order to

validate it in any case.

The same message type and structure will be used for the client's

response to a certificate request message. Note that a client MAY

send no certificates if it does not have an appropriate certificate

to send in response to the server's authentication request.

Note: PKCS #7 [PKCS7] is not used as the format for the certificate

vector because PKCS #6 [PKCS6] extended certificates are not used.

Also, PKCS #7 defines a SET rather than a SEQUENCE, making the task

of parsing the list more difficult.

The following rules apply to the certificates sent by the server:

- The certificate type MUST be X.509v3, unless explicitly negotiated

otherwise (e.g., [TLSPGP]).

- The end entity certificate's public key (and associated

restrictions) MUST be compatible with the selected key exchange

algorithm.

Key Exchange Alg. Certificate Key Type

RSA RSA public key; the certificate MUST allow the

RSA\_PSK key to be used for encryption (the

keyEncipherment bit MUST be set if the key

usage extension is present).

Note: RSA\_PSK is defined in [TLSPSK].

Dierks & Rescorla Standards Track [Page 48]

RFC 5246 TLS August 2008

DHE\_RSA RSA public key; the certificate MUST allow the

ECDHE\_RSA key to be used for signing (the

digitalSignature bit MUST be set if the key

usage extension is present) with the signature

scheme and hash algorithm that will be employed

in the server key exchange message.

Note: ECDHE\_RSA is defined in [TLSECC].

DHE\_DSS DSA public key; the certificate MUST allow the

key to be used for signing with the hash

algorithm that will be employed in the server

key exchange message.

DH\_DSS Diffie-Hellman public key; the keyAgreement bit

DH\_RSA MUST be set if the key usage extension is

present.

ECDH\_ECDSA ECDH-capable public key; the public key MUST

ECDH\_RSA use a curve and point format supported by the

client, as described in [TLSECC].

ECDHE\_ECDSA ECDSA-capable public key; the certificate MUST

allow the key to be used for signing with the

hash algorithm that will be employed in the

server key exchange message. The public key

MUST use a curve and point format supported by

the client, as described in [TLSECC].

- The "server\_name" and "trusted\_ca\_keys" extensions [TLSEXT] are

used to guide certificate selection.

If the client provided a "signature\_algorithms" extension, then all

certificates provided by the server MUST be signed by a

hash/signature algorithm pair that appears in that extension. Note

that this implies that a certificate containing a key for one

signature algorithm MAY be signed using a different signature

algorithm (for instance, an RSA key signed with a DSA key). This is

a departure from TLS 1.1, which required that the algorithms be the

same. Note that this also implies that the DH\_DSS, DH\_RSA,

ECDH\_ECDSA, and ECDH\_RSA key exchange algorithms do not restrict the

algorithm used to sign the certificate. Fixed DH certificates MAY be

signed with any hash/signature algorithm pair appearing in the

extension. The names DH\_DSS, DH\_RSA, ECDH\_ECDSA, and ECDH\_RSA are

historical.

Dierks & Rescorla Standards Track [Page 49]

RFC 5246 TLS August 2008

If the server has multiple certificates, it chooses one of them based

on the above-mentioned criteria (in addition to other criteria, such

as transport layer endpoint, local configuration and preferences,

etc.). If the server has a single certificate, it SHOULD attempt to

validate that it meets these criteria.

Note that there are certificates that use algorithms and/or algorithm

combinations that cannot be currently used with TLS. For example, a

certificate with RSASSA-PSS signature key (id-RSASSA-PSS OID in

SubjectPublicKeyInfo) cannot be used because TLS defines no

corresponding signature algorithm.

As cipher suites that specify new key exchange methods are specified

for the TLS protocol, they will imply the certificate format and the

required encoded keying information.

7.4.3. Server Key Exchange Message

When this message will be sent:

This message will be sent immediately after the server Certificate

message (or the ServerHello message, if this is an anonymous

negotiation).

The ServerKeyExchange message is sent by the server only when the

server Certificate message (if sent) does not contain enough data

to allow the client to exchange a premaster secret. This is true

for the following key exchange methods:

DHE\_DSS

DHE\_RSA

DH\_anon

It is not legal to send the ServerKeyExchange message for the

following key exchange methods:

RSA

DH\_DSS

DH\_RSA

Other key exchange algorithms, such as those defined in [TLSECC],

MUST specify whether the ServerKeyExchange message is sent or not;

and if the message is sent, its contents.

Dierks & Rescorla Standards Track [Page 50]

RFC 5246 TLS August 2008

Meaning of this message:

This message conveys cryptographic information to allow the client

to communicate the premaster secret: a Diffie-Hellman public key

with which the client can complete a key exchange (with the result

being the premaster secret) or a public key for some other

algorithm.

Structure of this message:

enum { dhe\_dss, dhe\_rsa, dh\_anon, rsa, dh\_dss, dh\_rsa

/\* may be extended, e.g., for ECDH -- see [TLSECC] \*/

} KeyExchangeAlgorithm;

struct {

opaque dh\_p<1..2^16-1>;

opaque dh\_g<1..2^16-1>;

opaque dh\_Ys<1..2^16-1>;

} ServerDHParams; /\* Ephemeral DH parameters \*/

dh\_p

The prime modulus used for the Diffie-Hellman operation.

dh\_g

The generator used for the Diffie-Hellman operation.

dh\_Ys

The server's Diffie-Hellman public value (g^X mod p).

Dierks & Rescorla Standards Track [Page 51]

RFC 5246 TLS August 2008

struct {

select (KeyExchangeAlgorithm) {

case dh\_anon:

ServerDHParams params;

case dhe\_dss:

case dhe\_rsa:

ServerDHParams params;

digitally-signed struct {

opaque client\_random[32];

opaque server\_random[32];

ServerDHParams params;

} signed\_params;

case rsa:

case dh\_dss:

case dh\_rsa:

struct {} ;

/\* message is omitted for rsa, dh\_dss, and dh\_rsa \*/

/\* may be extended, e.g., for ECDH -- see [TLSECC] \*/

};

} ServerKeyExchange;

params

The server's key exchange parameters.

signed\_params

For non-anonymous key exchanges, a signature over the server's

key exchange parameters.

If the client has offered the "signature\_algorithms" extension, the

signature algorithm and hash algorithm MUST be a pair listed in that

extension. Note that there is a possibility for inconsistencies

here. For instance, the client might offer DHE\_DSS key exchange but

omit any DSA pairs from its "signature\_algorithms" extension. In

order to negotiate correctly, the server MUST check any candidate

cipher suites against the "signature\_algorithms" extension before

selecting them. This is somewhat inelegant but is a compromise

designed to minimize changes to the original cipher suite design.

In addition, the hash and signature algorithms MUST be compatible

with the key in the server's end-entity certificate. RSA keys MAY be

used with any permitted hash algorithm, subject to restrictions in

the certificate, if any.

Because DSA signatures do not contain any secure indication of hash

algorithm, there is a risk of hash substitution if multiple hashes

may be used with any key. Currently, DSA [DSS] may only be used with

SHA-1. Future revisions of DSS [DSS-3] are expected to allow the use

of other digest algorithms with DSA, as well as guidance as to which

Dierks & Rescorla Standards Track [Page 52]

RFC 5246 TLS August 2008

digest algorithms should be used with each key size. In addition,

future revisions of [PKIX] may specify mechanisms for certificates to

indicate which digest algorithms are to be used with DSA.

As additional cipher suites are defined for TLS that include new key

exchange algorithms, the server key exchange message will be sent if

and only if the certificate type associated with the key exchange

algorithm does not provide enough information for the client to

exchange a premaster secret.

7.4.4. Certificate Request

When this message will be sent:

A non-anonymous server can optionally request a certificate from

the client, if appropriate for the selected cipher suite. This

message, if sent, will immediately follow the ServerKeyExchange

message (if it is sent; otherwise, this message follows the

server's Certificate message).

Structure of this message:

enum {

rsa\_sign(1), dss\_sign(2), rsa\_fixed\_dh(3), dss\_fixed\_dh(4),

rsa\_ephemeral\_dh\_RESERVED(5), dss\_ephemeral\_dh\_RESERVED(6),

fortezza\_dms\_RESERVED(20), (255)

} ClientCertificateType;

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

struct {

ClientCertificateType certificate\_types<1..2^8-1>;

SignatureAndHashAlgorithm

supported\_signature\_algorithms<2^16-1>;

DistinguishedName certificate\_authorities<0..2^16-1>;

} CertificateRequest;

certificate\_types

A list of the types of certificate types that the client may

offer.

rsa\_sign a certificate containing an RSA key

dss\_sign a certificate containing a DSA key

rsa\_fixed\_dh a certificate containing a static DH key.

dss\_fixed\_dh a certificate containing a static DH key

Dierks & Rescorla Standards Track [Page 53]

RFC 5246 TLS August 2008

supported\_signature\_algorithms

A list of the hash/signature algorithm pairs that the server is

able to verify, listed in descending order of preference.

certificate\_authorities

A list of the distinguished names [X501] of acceptable

certificate\_authorities, represented in DER-encoded format. These

distinguished names may specify a desired distinguished name for a

root CA or for a subordinate CA; thus, this message can be used to

describe known roots as well as a desired authorization space. If

the certificate\_authorities list is empty, then the client MAY

send any certificate of the appropriate ClientCertificateType,

unless there is some external arrangement to the contrary.

The interaction of the certificate\_types and

supported\_signature\_algorithms fields is somewhat complicated.

certificate\_types has been present in TLS since SSLv3, but was

somewhat underspecified. Much of its functionality is superseded by

supported\_signature\_algorithms. The following rules apply:

- Any certificates provided by the client MUST be signed using a

hash/signature algorithm pair found in

supported\_signature\_algorithms.

- The end-entity certificate provided by the client MUST contain a

key that is compatible with certificate\_types. If the key is a

signature key, it MUST be usable with some hash/signature

algorithm pair in supported\_signature\_algorithms.

- For historical reasons, the names of some client certificate types

include the algorithm used to sign the certificate. For example,

in earlier versions of TLS, rsa\_fixed\_dh meant a certificate

signed with RSA and containing a static DH key. In TLS 1.2, this

functionality has been obsoleted by the

supported\_signature\_algorithms, and the certificate type no longer

restricts the algorithm used to sign the certificate. For

example, if the server sends dss\_fixed\_dh certificate type and

{{sha1, dsa}, {sha1, rsa}} signature types, the client MAY reply

with a certificate containing a static DH key, signed with RSA-

SHA1.

New ClientCertificateType values are assigned by IANA as described in

Section 12.

Note: Values listed as RESERVED may not be used. They were used in

SSLv3.

Dierks & Rescorla Standards Track [Page 54]

RFC 5246 TLS August 2008

Note: It is a fatal handshake\_failure alert for an anonymous server

to request client authentication.

7.4.5. Server Hello Done

When this message will be sent:

The ServerHelloDone message is sent by the server to indicate the

end of the ServerHello and associated messages. After sending

this message, the server will wait for a client response.

Meaning of this message:

This message means that the server is done sending messages to

support the key exchange, and the client can proceed with its

phase of the key exchange.

Upon receipt of the ServerHelloDone message, the client SHOULD

verify that the server provided a valid certificate, if required,

and check that the server hello parameters are acceptable.

Structure of this message:

struct { } ServerHelloDone;

7.4.6. Client Certificate

When this message will be sent:

This is the first message the client can send after receiving a

ServerHelloDone message. This message is only sent if the server

requests a certificate. If no suitable certificate is available,

the client MUST send a certificate message containing no

certificates. That is, the certificate\_list structure has a

length of zero. If the client does not send any certificates, the

server MAY at its discretion either continue the handshake without

client authentication, or respond with a fatal handshake\_failure

alert. Also, if some aspect of the certificate chain was

unacceptable (e.g., it was not signed by a known, trusted CA), the

server MAY at its discretion either continue the handshake

(considering the client unauthenticated) or send a fatal alert.

Client certificates are sent using the Certificate structure

defined in Section 7.4.2.

Dierks & Rescorla Standards Track [Page 55]

RFC 5246 TLS August 2008

Meaning of this message:

This message conveys the client's certificate chain to the server;

the server will use it when verifying the CertificateVerify

message (when the client authentication is based on signing) or

calculating the premaster secret (for non-ephemeral Diffie-

Hellman). The certificate MUST be appropriate for the negotiated

cipher suite's key exchange algorithm, and any negotiated

extensions.

In particular:

- The certificate type MUST be X.509v3, unless explicitly negotiated

otherwise (e.g., [TLSPGP]).

- The end-entity certificate's public key (and associated

restrictions) has to be compatible with the certificate types

listed in CertificateRequest:

Client Cert. Type Certificate Key Type

rsa\_sign RSA public key; the certificate MUST allow the

key to be used for signing with the signature

scheme and hash algorithm that will be

employed in the certificate verify message.

dss\_sign DSA public key; the certificate MUST allow the

key to be used for signing with the hash

algorithm that will be employed in the

certificate verify message.

ecdsa\_sign ECDSA-capable public key; the certificate MUST

allow the key to be used for signing with the

hash algorithm that will be employed in the

certificate verify message; the public key

MUST use a curve and point format supported by

the server.

rsa\_fixed\_dh Diffie-Hellman public key; MUST use the same

dss\_fixed\_dh parameters as server's key.

rsa\_fixed\_ecdh ECDH-capable public key; MUST use the

ecdsa\_fixed\_ecdh same curve as the server's key, and MUST use a

point format supported by the server.

- If the certificate\_authorities list in the certificate request

message was non-empty, one of the certificates in the certificate

chain SHOULD be issued by one of the listed CAs.

Dierks & Rescorla Standards Track [Page 56]

RFC 5246 TLS August 2008

- The certificates MUST be signed using an acceptable hash/

signature algorithm pair, as described in Section 7.4.4. Note

that this relaxes the constraints on certificate-signing

algorithms found in prior versions of TLS.

Note that, as with the server certificate, there are certificates

that use algorithms/algorithm combinations that cannot be currently

used with TLS.

7.4.7. Client Key Exchange Message

When this message will be sent:

This message is always sent by the client. It MUST immediately

follow the client certificate message, if it is sent. Otherwise,

it MUST be the first message sent by the client after it receives

the ServerHelloDone message.

Meaning of this message:

With this message, the premaster secret is set, either by direct

transmission of the RSA-encrypted secret or by the transmission of

Diffie-Hellman parameters that will allow each side to agree upon

the same premaster secret.

When the client is using an ephemeral Diffie-Hellman exponent,

then this message contains the client's Diffie-Hellman public

value. If the client is sending a certificate containing a static

DH exponent (i.e., it is doing fixed\_dh client authentication),

then this message MUST be sent but MUST be empty.

Structure of this message:

The choice of messages depends on which key exchange method has

been selected. See Section 7.4.3 for the KeyExchangeAlgorithm

definition.

Dierks & Rescorla Standards Track [Page 57]

RFC 5246 TLS August 2008

struct {

select (KeyExchangeAlgorithm) {

case rsa:

EncryptedPreMasterSecret;

case dhe\_dss:

case dhe\_rsa:

case dh\_dss:

case dh\_rsa:

case dh\_anon:

ClientDiffieHellmanPublic;

} exchange\_keys;

} ClientKeyExchange;

7.4.7.1. RSA-Encrypted Premaster Secret Message

Meaning of this message:

If RSA is being used for key agreement and authentication, the

client generates a 48-byte premaster secret, encrypts it using the

public key from the server's certificate, and sends the result in

an encrypted premaster secret message. This structure is a

variant of the ClientKeyExchange message and is not a message in

itself.

Structure of this message:

struct {

ProtocolVersion client\_version;

opaque random[46];

} PreMasterSecret;

client\_version

The latest (newest) version supported by the client. This is

used to detect version rollback attacks.

random

46 securely-generated random bytes.

struct {

public-key-encrypted PreMasterSecret pre\_master\_secret;

} EncryptedPreMasterSecret;

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.

Dierks & Rescorla Standards Track [Page 58]

RFC 5246 TLS August 2008

Note: The version number in the PreMasterSecret is the version

offered by the client in the ClientHello.client\_version, not the

version negotiated for the connection. This feature is designed to

prevent rollback attacks. Unfortunately, some old implementations

use the negotiated version instead, and therefore checking the

version number may lead to failure to interoperate with such

incorrect client implementations.

Client implementations MUST always send the correct version number in

PreMasterSecret. If ClientHello.client\_version is TLS 1.1 or higher,

server implementations MUST check the version number as described in

the note below. If the version number is TLS 1.0 or earlier, server

implementations SHOULD check the version number, but MAY have a

configuration option to disable the check. Note that if the check

fails, the PreMasterSecret SHOULD be randomized as described below.

Note: Attacks discovered by Bleichenbacher [BLEI] and Klima et al.

[KPR03] can be used to attack a TLS server that reveals whether a

particular message, when decrypted, is properly PKCS#1 formatted,

contains a valid PreMasterSecret structure, or has the correct

version number.

As described by Klima [KPR03], these vulnerabilities can be avoided

by treating incorrectly formatted message blocks and/or mismatched

version numbers in a manner indistinguishable from correctly

formatted RSA blocks. In other words:

1. Generate a string R of 46 random bytes

2. Decrypt the message to recover the plaintext M

3. If the PKCS#1 padding is not correct, or the length of message

M is not exactly 48 bytes:

pre\_master\_secret = ClientHello.client\_version || R

else If ClientHello.client\_version <= TLS 1.0, and version

number check is explicitly disabled:

pre\_master\_secret = M

else:

pre\_master\_secret = ClientHello.client\_version || M[2..47]

Note that explicitly constructing the pre\_master\_secret with the

ClientHello.client\_version produces an invalid master\_secret if the

client has sent the wrong version in the original pre\_master\_secret.

An alternative approach is to treat a version number mismatch as a

PKCS-1 formatting error and randomize the premaster secret

completely:

Dierks & Rescorla Standards Track [Page 59]

RFC 5246 TLS August 2008

1. Generate a string R of 48 random bytes

2. Decrypt the message to recover the plaintext M

3. If the PKCS#1 padding is not correct, or the length of message

M is not exactly 48 bytes:

pre\_master\_secret = R

else If ClientHello.client\_version <= TLS 1.0, and version

number check is explicitly disabled:

premaster secret = M

else If M[0..1] != ClientHello.client\_version:

premaster secret = R

else:

premaster secret = M

Although no practical attacks against this construction are known,

Klima et al. [KPR03] describe some theoretical attacks, and therefore

the first construction described is RECOMMENDED.

In any case, a TLS server MUST NOT generate an alert if processing an

RSA-encrypted premaster secret message fails, or the version number

is not as expected. Instead, it MUST continue the handshake with a

randomly generated premaster secret. It may be useful to log the

real cause of failure for troubleshooting purposes; however, care

must be taken to avoid leaking the information to an attacker

(through, e.g., timing, log files, or other channels.)

The RSAES-OAEP encryption scheme defined in [PKCS1] is more secure

against the Bleichenbacher attack. However, for maximal

compatibility with earlier versions of TLS, this specification uses

the RSAES-PKCS1-v1\_5 scheme. No variants of the Bleichenbacher

attack are known to exist provided that the above recommendations are

followed.

Implementation note: Public-key-encrypted data is represented as an

opaque vector <0..2^16-1> (see Section 4.7). Thus, the RSA-encrypted

PreMasterSecret in a ClientKeyExchange is preceded by two length

bytes. These bytes are redundant in the case of RSA because the

EncryptedPreMasterSecret is the only data in the ClientKeyExchange

and its length can therefore be unambiguously determined. The SSLv3

specification was not clear about the encoding of public-key-

encrypted data, and therefore many SSLv3 implementations do not

include the length bytes -- they encode the RSA-encrypted data

directly in the ClientKeyExchange message.

This specification requires correct encoding of the

EncryptedPreMasterSecret complete with length bytes. The resulting

PDU is incompatible with many SSLv3 implementations. Implementors

Dierks & Rescorla Standards Track [Page 60]

RFC 5246 TLS August 2008

upgrading from SSLv3 MUST modify their implementations to generate

and accept the correct encoding. Implementors who wish to be

compatible with both SSLv3 and TLS should make their implementation's

behavior dependent on the protocol version.

Implementation note: It is now known that remote timing-based attacks

on TLS are possible, at least when the client and server are on the

same LAN. Accordingly, implementations that use static RSA keys MUST

use RSA blinding or some other anti-timing technique, as described in

[TIMING].

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, and not a message in itself.

Structure of this message:

enum { implicit, explicit } PublicValueEncoding;

implicit

If the client has sent a certificate which contains a suitable

Diffie-Hellman key (for fixed\_dh client authentication), then

Yc is implicit and does not need to be sent again. In this

case, the client key exchange message will be sent, but it MUST

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).

Dierks & Rescorla Standards Track [Page 61]

RFC 5246 TLS August 2008

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 MUST immediately follow the client key exchange message.

Structure of this message:

struct {

digitally-signed struct {

opaque handshake\_messages[handshake\_messages\_length];

}

} CertificateVerify;

Here handshake\_messages refers to all handshake messages sent or

received, starting at client hello and 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 Section 7.4) exchanged thus

far. Note that this requires both sides to either buffer the

messages or compute running hashes for all potential hash

algorithms up to the time of the CertificateVerify computation.

Servers can minimize this computation cost by offering a

restricted set of digest algorithms in the CertificateRequest

message.

The hash and signature algorithms used in the signature MUST be

one of those present in the supported\_signature\_algorithms field

of the CertificateRequest message. In addition, the hash and

signature algorithms MUST be compatible with the key in the

client's end-entity certificate. RSA keys MAY be used with any

permitted hash algorithm, subject to restrictions in the

certificate, if any.

Because DSA signatures do not contain any secure indication of

hash algorithm, there is a risk of hash substitution if multiple

hashes may be used with any key. Currently, DSA [DSS] may only be

used with SHA-1. Future revisions of DSS [DSS-3] are expected to

allow the use of other digest algorithms with DSA, as well as

guidance as to which digest algorithms should be used with each

key size. In addition, future revisions of [PKIX] may specify

mechanisms for certificates to indicate which digest algorithms

are to be used with DSA.

Dierks & Rescorla Standards Track [Page 62]

RFC 5246 TLS August 2008

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 one 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.

Structure of this message:

struct {

opaque verify\_data[verify\_data\_length];

} Finished;

verify\_data

PRF(master\_secret, finished\_label, Hash(handshake\_messages))

[0..verify\_data\_length-1];

finished\_label

For Finished messages sent by the client, the string

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

the string "server finished".

Hash denotes a Hash of the handshake messages. For the PRF

defined in Section 5, the Hash MUST be the Hash used as the basis

for the PRF. Any cipher suite which defines a different PRF MUST

also define the Hash to use in the Finished computation.

In previous versions of TLS, the verify\_data was always 12 octets

long. In the current version of TLS, it depends on the cipher

suite. Any cipher suite which does not explicitly specify

verify\_data\_length has a verify\_data\_length equal to 12. This

includes all existing cipher suites. Note that this

representation has the same encoding as with previous versions.

Future cipher suites MAY specify other lengths but such length

MUST be at least 12 bytes.

Dierks & Rescorla Standards Track [Page 63]

RFC 5246 TLS August 2008

handshake\_messages

All of the data from all messages in this handshake (not

including any HelloRequest 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

Section 7.4, exchanged thus far.

It is a fatal error if a Finished message is not preceded by a

ChangeCipherSpec message at the appropriate point in the handshake.

The value handshake\_messages includes all handshake messages starting

at ClientHello 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 CertificateVerify 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 that is sent second will include the prior one.

Note: ChangeCipherSpec messages, alerts, and any other record types

are not handshake messages and are not included in the hash

computations. Also, HelloRequest 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 ServerHello 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.

Dierks & Rescorla Standards Track [Page 64]

RFC 5246 TLS August 2008

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.

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. Leading bytes of Z that

contain all zero bits are stripped before it is used as the

pre\_master\_secret.

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\_RSA\_WITH\_AES\_128\_CBC\_SHA (see Appendix A.5 for the

definition).

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.

11. Security Considerations

Security issues are discussed throughout this memo, especially in

Appendices D, E, and F.

12. IANA Considerations

This document uses several registries that were originally created in

[TLS1.1]. IANA has updated these to reference this document. The

registries and their allocation policies (unchanged from [TLS1.1])

are listed below.

Dierks & Rescorla Standards Track [Page 65]

RFC 5246 TLS August 2008

- TLS ClientCertificateType Identifiers Registry: Future values in

the range 0-63 (decimal) inclusive are assigned via Standards

Action [RFC2434]. Values in the range 64-223 (decimal) inclusive

are assigned via Specification Required [RFC2434]. Values from

224-255 (decimal) inclusive are reserved for Private Use

[RFC2434].

- TLS Cipher Suite Registry: Future values with the first byte in

the range 0-191 (decimal) inclusive are assigned via Standards

Action [RFC2434]. Values with the first byte in the range 192-254

(decimal) are assigned via Specification Required [RFC2434].

Values with the first byte 255 (decimal) are reserved for Private

Use [RFC2434].

- This document defines several new HMAC-SHA256-based cipher suites,

whose values (in Appendix A.5) have been allocated from the TLS

Cipher Suite registry.

- TLS ContentType Registry: Future values are allocated via

Standards Action [RFC2434].

- TLS Alert Registry: Future values are allocated via Standards

Action [RFC2434].

- TLS HandshakeType Registry: Future values are allocated via

Standards Action [RFC2434].

This document also uses a registry originally created in [RFC4366].

IANA has updated it to reference this document. The registry and its

allocation policy (unchanged from [RFC4366]) is listed below:

- TLS ExtensionType Registry: Future values are allocated via IETF

Consensus [RFC2434]. IANA has updated this registry to include

the signature\_algorithms extension and its corresponding value

(see Section 7.4.1.4).

In addition, this document defines two new registries to be

maintained by IANA:

- TLS SignatureAlgorithm Registry: The registry has been initially

populated with the values described in Section 7.4.1.4.1. Future

values in the range 0-63 (decimal) inclusive are assigned via

Standards Action [RFC2434]. Values in the range 64-223 (decimal)

inclusive are assigned via Specification Required [RFC2434].

Values from 224-255 (decimal) inclusive are reserved for Private

Use [RFC2434].

Dierks & Rescorla Standards Track [Page 66]

RFC 5246 TLS August 2008

- TLS HashAlgorithm Registry: The registry has been initially

populated with the values described in Section 7.4.1.4.1. Future

values in the range 0-63 (decimal) inclusive are assigned via

Standards Action [RFC2434]. Values in the range 64-223 (decimal)

inclusive are assigned via Specification Required [RFC2434].

Values from 224-255 (decimal) inclusive are reserved for Private

Use [RFC2434].

This document also uses the TLS Compression Method Identifiers

Registry, defined in [RFC3749]. IANA has allocated value 0 for

the "null" compression method.

Dierks & Rescorla Standards Track [Page 67]

RFC 5246 TLS August 2008

Appendix A. Protocol Data Structures and Constant Values

This section describes protocol types and constants.

A.1. Record Layer

struct {

uint8 major;

uint8 minor;

} ProtocolVersion;

ProtocolVersion version = { 3, 3 }; /\* TLS v1.2\*/

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 (SecurityParameters.cipher\_type) {

case stream: GenericStreamCipher;

case block: GenericBlockCipher;

case aead: GenericAEADCipher;

} fragment;

} TLSCiphertext;

stream-ciphered struct {

opaque content[TLSCompressed.length];

opaque MAC[SecurityParameters.mac\_length];

} GenericStreamCipher;

Dierks & Rescorla Standards Track [Page 68]

RFC 5246 TLS August 2008

struct {

opaque IV[SecurityParameters.record\_iv\_length];

block-ciphered struct {

opaque content[TLSCompressed.length];

opaque MAC[SecurityParameters.mac\_length];

uint8 padding[GenericBlockCipher.padding\_length];

uint8 padding\_length;

};

} GenericBlockCipher;

struct {

opaque nonce\_explicit[SecurityParameters.record\_iv\_length];

aead-ciphered struct {

opaque content[TLSCompressed.length];

};

} GenericAEADCipher;

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\_RESERVED(21),

record\_overflow(22),

decompression\_failure(30),

handshake\_failure(40),

no\_certificate\_RESERVED(41),

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\_RESERVED(60),

protocol\_version(70),

Dierks & Rescorla Standards Track [Page 69]

RFC 5246 TLS August 2008

insufficient\_security(71),

internal\_error(80),

user\_canceled(90),

no\_renegotiation(100),

unsupported\_extension(110), /\* new \*/

(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;

Dierks & Rescorla Standards Track [Page 70]

RFC 5246 TLS August 2008

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..2^16-2>;

CompressionMethod compression\_methods<1..2^8-1>;

select (extensions\_present) {

case false:

struct {};

case true:

Extension extensions<0..2^16-1>;

};

} ClientHello;

struct {

ProtocolVersion server\_version;

Random random;

SessionID session\_id;

CipherSuite cipher\_suite;

CompressionMethod compression\_method;

select (extensions\_present) {

case false:

struct {};

case true:

Extension extensions<0..2^16-1>;

};

} ServerHello;

struct {

ExtensionType extension\_type;

opaque extension\_data<0..2^16-1>;

} Extension;

Dierks & Rescorla Standards Track [Page 71]

RFC 5246 TLS August 2008

enum {

signature\_algorithms(13), (65535)

} ExtensionType;

enum{

none(0), md5(1), sha1(2), sha224(3), sha256(4), sha384(5),

sha512(6), (255)

} HashAlgorithm;

enum {

anonymous(0), rsa(1), dsa(2), ecdsa(3), (255)

} SignatureAlgorithm;

struct {

HashAlgorithm hash;

SignatureAlgorithm signature;

} SignatureAndHashAlgorithm;

SignatureAndHashAlgorithm

supported\_signature\_algorithms<2..2^16-1>;

A.4.2. Server Authentication and Key Exchange Messages

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

struct {

ASN.1Cert certificate\_list<0..2^24-1>;

} Certificate;

enum { dhe\_dss, dhe\_rsa, dh\_anon, rsa,dh\_dss, dh\_rsa

/\* may be extended, e.g., for ECDH -- see [TLSECC] \*/

} KeyExchangeAlgorithm;

struct {

opaque dh\_p<1..2^16-1>;

opaque dh\_g<1..2^16-1>;

opaque dh\_Ys<1..2^16-1>;

} ServerDHParams; /\* Ephemeral DH parameters \*/

Dierks & Rescorla Standards Track [Page 72]

RFC 5246 TLS August 2008

struct {

select (KeyExchangeAlgorithm) {

case dh\_anon:

ServerDHParams params;

case dhe\_dss:

case dhe\_rsa:

ServerDHParams params;

digitally-signed struct {

opaque client\_random[32];

opaque server\_random[32];

ServerDHParams params;

} signed\_params;

case rsa:

case dh\_dss:

case dh\_rsa:

struct {} ;

/\* message is omitted for rsa, dh\_dss, and dh\_rsa \*/

/\* may be extended, e.g., for ECDH -- see [TLSECC] \*/

} ServerKeyExchange;

enum {

rsa\_sign(1), dss\_sign(2), rsa\_fixed\_dh(3), dss\_fixed\_dh(4),

rsa\_ephemeral\_dh\_RESERVED(5), dss\_ephemeral\_dh\_RESERVED(6),

fortezza\_dms\_RESERVED(20),

(255)

} ClientCertificateType;

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

struct {

ClientCertificateType certificate\_types<1..2^8-1>;

DistinguishedName certificate\_authorities<0..2^16-1>;

} CertificateRequest;

struct { } ServerHelloDone;

Dierks & Rescorla Standards Track [Page 73]

RFC 5246 TLS August 2008

A.4.3. Client Authentication and Key Exchange Messages

struct {

select (KeyExchangeAlgorithm) {

case rsa:

EncryptedPreMasterSecret;

case dhe\_dss:

case dhe\_rsa:

case dh\_dss:

case dh\_rsa:

case dh\_anon:

ClientDiffieHellmanPublic;

} 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 {

digitally-signed struct {

opaque handshake\_messages[handshake\_messages\_length];

}

} CertificateVerify;

A.4.4. Handshake Finalization Message

struct {

opaque verify\_data[verify\_data\_length];

} Finished;

Dierks & Rescorla Standards Track [Page 74]

RFC 5246 TLS August 2008

A.5. The Cipher Suite

The following values define the cipher suite codes used in the

ClientHello and ServerHello messages.

A cipher suite defines a cipher specification supported in TLS

Version 1.2.

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 any 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\_WITH\_NULL\_SHA256 = { 0x00,0x3B };

CipherSuite TLS\_RSA\_WITH\_RC4\_128\_MD5 = { 0x00,0x04 };

CipherSuite TLS\_RSA\_WITH\_RC4\_128\_SHA = { 0x00,0x05 };

CipherSuite TLS\_RSA\_WITH\_3DES\_EDE\_CBC\_SHA = { 0x00,0x0A };

CipherSuite TLS\_RSA\_WITH\_AES\_128\_CBC\_SHA = { 0x00,0x2F };

CipherSuite TLS\_RSA\_WITH\_AES\_256\_CBC\_SHA = { 0x00,0x35 };

CipherSuite TLS\_RSA\_WITH\_AES\_128\_CBC\_SHA256 = { 0x00,0x3C };

CipherSuite TLS\_RSA\_WITH\_AES\_256\_CBC\_SHA256 = { 0x00,0x3D };

The following cipher suite 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 signature-capable certificate, which has

been signed by the CA. The signing algorithm used by the server is

specified after the DHE component of the CipherSuite name. The

server can request any 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.

Dierks & Rescorla Standards Track [Page 75]

RFC 5246 TLS August 2008

CipherSuite TLS\_DH\_DSS\_WITH\_3DES\_EDE\_CBC\_SHA = { 0x00,0x0D };

CipherSuite TLS\_DH\_RSA\_WITH\_3DES\_EDE\_CBC\_SHA = { 0x00,0x10 };

CipherSuite TLS\_DHE\_DSS\_WITH\_3DES\_EDE\_CBC\_SHA = { 0x00,0x13 };

CipherSuite TLS\_DHE\_RSA\_WITH\_3DES\_EDE\_CBC\_SHA = { 0x00,0x16 };

CipherSuite TLS\_DH\_DSS\_WITH\_AES\_128\_CBC\_SHA = { 0x00,0x30 };

CipherSuite TLS\_DH\_RSA\_WITH\_AES\_128\_CBC\_SHA = { 0x00,0x31 };

CipherSuite TLS\_DHE\_DSS\_WITH\_AES\_128\_CBC\_SHA = { 0x00,0x32 };

CipherSuite TLS\_DHE\_RSA\_WITH\_AES\_128\_CBC\_SHA = { 0x00,0x33 };

CipherSuite TLS\_DH\_DSS\_WITH\_AES\_256\_CBC\_SHA = { 0x00,0x36 };

CipherSuite TLS\_DH\_RSA\_WITH\_AES\_256\_CBC\_SHA = { 0x00,0x37 };

CipherSuite TLS\_DHE\_DSS\_WITH\_AES\_256\_CBC\_SHA = { 0x00,0x38 };

CipherSuite TLS\_DHE\_RSA\_WITH\_AES\_256\_CBC\_SHA = { 0x00,0x39 };

CipherSuite TLS\_DH\_DSS\_WITH\_AES\_128\_CBC\_SHA256 = { 0x00,0x3E };

CipherSuite TLS\_DH\_RSA\_WITH\_AES\_128\_CBC\_SHA256 = { 0x00,0x3F };

CipherSuite TLS\_DHE\_DSS\_WITH\_AES\_128\_CBC\_SHA256 = { 0x00,0x40 };

CipherSuite TLS\_DHE\_RSA\_WITH\_AES\_128\_CBC\_SHA256 = { 0x00,0x67 };

CipherSuite TLS\_DH\_DSS\_WITH\_AES\_256\_CBC\_SHA256 = { 0x00,0x68 };

CipherSuite TLS\_DH\_RSA\_WITH\_AES\_256\_CBC\_SHA256 = { 0x00,0x69 };

CipherSuite TLS\_DHE\_DSS\_WITH\_AES\_256\_CBC\_SHA256 = { 0x00,0x6A };

CipherSuite TLS\_DHE\_RSA\_WITH\_AES\_256\_CBC\_SHA256 = { 0x00,0x6B };

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. Using this mode therefore is of limited use: These

cipher suites MUST NOT be used by TLS 1.2 implementations unless the

application layer has specifically requested to allow anonymous key

exchange. (Anonymous key exchange may sometimes be acceptable, for

example, to support opportunistic encryption when no set-up for

authentication is in place, or when TLS is used as part of more

complex security protocols that have other means to ensure

authentication.)

CipherSuite TLS\_DH\_anon\_WITH\_RC4\_128\_MD5 = { 0x00,0x18 };

CipherSuite TLS\_DH\_anon\_WITH\_3DES\_EDE\_CBC\_SHA = { 0x00,0x1B };

CipherSuite TLS\_DH\_anon\_WITH\_AES\_128\_CBC\_SHA = { 0x00,0x34 };

CipherSuite TLS\_DH\_anon\_WITH\_AES\_256\_CBC\_SHA = { 0x00,0x3A };

CipherSuite TLS\_DH\_anon\_WITH\_AES\_128\_CBC\_SHA256 = { 0x00,0x6C };

CipherSuite TLS\_DH\_anon\_WITH\_AES\_256\_CBC\_SHA256 = { 0x00,0x6D };

Note that using non-anonymous key exchange without actually verifying

the key exchange is essentially equivalent to anonymous key exchange,

and the same precautions apply. While non-anonymous key exchange

will generally involve a higher computational and communicational

cost than anonymous key exchange, it may be in the interest of

interoperability not to disable non-anonymous key exchange when the

application layer is allowing anonymous key exchange.

Dierks & Rescorla Standards Track [Page 76]

RFC 5246 TLS August 2008

New cipher suite values have been assigned by IANA as described in

Section 12.

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 { tls\_prf\_sha256 } PRFAlgorithm;

enum { null, rc4, 3des, aes } BulkCipherAlgorithm;

enum { stream, block, aead } CipherType;

enum { null, hmac\_md5, hmac\_sha1, hmac\_sha256, hmac\_sha384,

hmac\_sha512} MACAlgorithm;

/\* Other values may be added to the algorithms specified in

CompressionMethod, PRFAlgorithm, BulkCipherAlgorithm, and

MACAlgorithm. \*/

struct {

ConnectionEnd entity;

PRFAlgorithm prf\_algorithm;

BulkCipherAlgorithm bulk\_cipher\_algorithm;

CipherType cipher\_type;

uint8 enc\_key\_length;

uint8 block\_length;

uint8 fixed\_iv\_length;

uint8 record\_iv\_length;

MACAlgorithm mac\_algorithm;

uint8 mac\_length;

uint8 mac\_key\_length;

CompressionMethod compression\_algorithm;

opaque master\_secret[48];

opaque client\_random[32];

opaque server\_random[32];

} SecurityParameters;

Dierks & Rescorla Standards Track [Page 77]

RFC 5246 TLS August 2008

A.7. Changes to RFC 4492

RFC 4492 [TLSECC] adds Elliptic Curve cipher suites to TLS. This

document changes some of the structures used in that document. This

section details the required changes for implementors of both RFC

4492 and TLS 1.2. Implementors of TLS 1.2 who are not implementing

RFC 4492 do not need to read this section.

This document adds a "signature\_algorithm" field to the digitally-

signed element in order to identify the signature and digest

algorithms used to create a signature. This change applies to

digital signatures formed using ECDSA as well, thus allowing ECDSA

signatures to be used with digest algorithms other than SHA-1,

provided such use is compatible with the certificate and any

restrictions imposed by future revisions of [PKIX].

As described in Sections 7.4.2 and 7.4.6, the restrictions on the

signature algorithms used to sign certificates are no longer tied to

the cipher suite (when used by the server) or the

ClientCertificateType (when used by the client). Thus, the

restrictions on the algorithm used to sign certificates specified in

Sections 2 and 3 of RFC 4492 are also relaxed. As in this document,

the restrictions on the keys in the end-entity certificate remain.

Appendix B. Glossary

Advanced Encryption Standard (AES)

AES [AES] is a widely used symmetric encryption algorithm. AES is

a block cipher with a 128-, 192-, or 256-bit keys and a 16-byte

block size. TLS currently only supports the 128- and 256-bit key

sizes.

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.

authenticated encryption with additional data (AEAD)

A symmetric encryption algorithm that simultaneously provides

confidentiality and message integrity.

authentication

Authentication is the ability of one entity to determine the

identity of another entity.

Dierks & Rescorla Standards Track [Page 78]

RFC 5246 TLS August 2008

block cipher

A block cipher is an algorithm that operates on plaintext in

groups of bits, called blocks. 64 bits was, and 128 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 key

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 [DES] still is a very widely used symmetric encryption

algorithm although it is considered as rather weak now. 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

Dierks & Rescorla Standards Track [Page 79]

RFC 5246 TLS August 2008

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 [3DES] 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.

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 186-2, "Digital Signature

Standard", published January 2000 by the U.S. Department of

Commerce [DSS]. A significant update [DSS-3] has been drafted and

was published in March 2006.

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.

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 [MD5] is a hashing function that converts an arbitrarily long

data stream into a hash of fixed size (16 bytes). Due to

significant progress in cryptanalysis, at the time of publication

of this document, MD5 no longer can be considered a 'secure'

hashing function.

Dierks & Rescorla Standards Track [Page 80]

RFC 5246 TLS August 2008

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.

RC4

A stream cipher invented by Ron Rivest. A compatible cipher is

described in [SCH].

RSA

A very widely used public key algorithm that can be used for

either encryption or digital signing. [RSA]

server

The server is the application entity that responds to requests for

connections from clients. See also "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 that 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 key

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

SHA

The Secure Hash Algorithm [SHS] is defined in FIPS PUB 180-2. It

produces a 20-byte output. Note that all references to SHA

(without a numerical suffix) actually use the modified SHA-1

algorithm.

Dierks & Rescorla Standards Track [Page 81]

RFC 5246 TLS August 2008

SHA-256

The 256-bit Secure Hash Algorithm is defined in FIPS PUB 180-2.

It produces a 32-byte output.

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 "Working Group

Information" at the end of this document (see page 99).

Dierks & Rescorla Standards Track [Page 82]

RFC 5246 TLS August 2008

Appendix C. Cipher Suite Definitions

Cipher Suite Key Cipher Mac

Exchange

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\_WITH\_NULL\_SHA256 RSA NULL SHA256

TLS\_RSA\_WITH\_RC4\_128\_MD5 RSA RC4\_128 MD5

TLS\_RSA\_WITH\_RC4\_128\_SHA RSA RC4\_128 SHA

TLS\_RSA\_WITH\_3DES\_EDE\_CBC\_SHA RSA 3DES\_EDE\_CBC SHA

TLS\_RSA\_WITH\_AES\_128\_CBC\_SHA RSA AES\_128\_CBC SHA

TLS\_RSA\_WITH\_AES\_256\_CBC\_SHA RSA AES\_256\_CBC SHA

TLS\_RSA\_WITH\_AES\_128\_CBC\_SHA256 RSA AES\_128\_CBC SHA256

TLS\_RSA\_WITH\_AES\_256\_CBC\_SHA256 RSA AES\_256\_CBC SHA256

TLS\_DH\_DSS\_WITH\_3DES\_EDE\_CBC\_SHA DH\_DSS 3DES\_EDE\_CBC SHA

TLS\_DH\_RSA\_WITH\_3DES\_EDE\_CBC\_SHA DH\_RSA 3DES\_EDE\_CBC SHA

TLS\_DHE\_DSS\_WITH\_3DES\_EDE\_CBC\_SHA DHE\_DSS 3DES\_EDE\_CBC SHA

TLS\_DHE\_RSA\_WITH\_3DES\_EDE\_CBC\_SHA DHE\_RSA 3DES\_EDE\_CBC SHA

TLS\_DH\_anon\_WITH\_RC4\_128\_MD5 DH\_anon RC4\_128 MD5

TLS\_DH\_anon\_WITH\_3DES\_EDE\_CBC\_SHA DH\_anon 3DES\_EDE\_CBC SHA

TLS\_DH\_DSS\_WITH\_AES\_128\_CBC\_SHA DH\_DSS AES\_128\_CBC SHA

TLS\_DH\_RSA\_WITH\_AES\_128\_CBC\_SHA DH\_RSA AES\_128\_CBC SHA

TLS\_DHE\_DSS\_WITH\_AES\_128\_CBC\_SHA DHE\_DSS AES\_128\_CBC SHA

TLS\_DHE\_RSA\_WITH\_AES\_128\_CBC\_SHA DHE\_RSA AES\_128\_CBC SHA

TLS\_DH\_anon\_WITH\_AES\_128\_CBC\_SHA DH\_anon AES\_128\_CBC SHA

TLS\_DH\_DSS\_WITH\_AES\_256\_CBC\_SHA DH\_DSS AES\_256\_CBC SHA

TLS\_DH\_RSA\_WITH\_AES\_256\_CBC\_SHA DH\_RSA AES\_256\_CBC SHA

TLS\_DHE\_DSS\_WITH\_AES\_256\_CBC\_SHA DHE\_DSS AES\_256\_CBC SHA

TLS\_DHE\_RSA\_WITH\_AES\_256\_CBC\_SHA DHE\_RSA AES\_256\_CBC SHA

TLS\_DH\_anon\_WITH\_AES\_256\_CBC\_SHA DH\_anon AES\_256\_CBC SHA

TLS\_DH\_DSS\_WITH\_AES\_128\_CBC\_SHA256 DH\_DSS AES\_128\_CBC SHA256

TLS\_DH\_RSA\_WITH\_AES\_128\_CBC\_SHA256 DH\_RSA AES\_128\_CBC SHA256

TLS\_DHE\_DSS\_WITH\_AES\_128\_CBC\_SHA256 DHE\_DSS AES\_128\_CBC SHA256

TLS\_DHE\_RSA\_WITH\_AES\_128\_CBC\_SHA256 DHE\_RSA AES\_128\_CBC SHA256

TLS\_DH\_anon\_WITH\_AES\_128\_CBC\_SHA256 DH\_anon AES\_128\_CBC SHA256

TLS\_DH\_DSS\_WITH\_AES\_256\_CBC\_SHA256 DH\_DSS AES\_256\_CBC SHA256

TLS\_DH\_RSA\_WITH\_AES\_256\_CBC\_SHA256 DH\_RSA AES\_256\_CBC SHA256

TLS\_DHE\_DSS\_WITH\_AES\_256\_CBC\_SHA256 DHE\_DSS AES\_256\_CBC SHA256

TLS\_DHE\_RSA\_WITH\_AES\_256\_CBC\_SHA256 DHE\_RSA AES\_256\_CBC SHA256

TLS\_DH\_anon\_WITH\_AES\_256\_CBC\_SHA256 DH\_anon AES\_256\_CBC SHA256

Dierks & Rescorla Standards Track [Page 83]

RFC 5246 TLS August 2008

Key IV Block

Cipher Type Material Size Size

------------ ------ -------- ---- -----

NULL Stream 0 0 N/A

RC4\_128 Stream 16 0 N/A

3DES\_EDE\_CBC Block 24 8 8

AES\_128\_CBC Block 16 16 16

AES\_256\_CBC Block 32 16 16

MAC Algorithm mac\_length mac\_key\_length

-------- ----------- ---------- --------------

NULL N/A 0 0

MD5 HMAC-MD5 16 16

SHA HMAC-SHA1 20 20

SHA256 HMAC-SHA256 32 32

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.

IV Size

The amount of data needed to be generated for the initialization

vector. Zero for stream ciphers; equal to the block size for

block ciphers (this is equal to

SecurityParameters.record\_iv\_length).

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.

Dierks & Rescorla Standards Track [Page 84]

RFC 5246 TLS August 2008

Appendix D. Implementation Notes

The TLS protocol cannot prevent many common security mistakes. This

section provides several recommendations to assist implementors.

D.1. 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 SHA-1, are acceptable,

but cannot provide more security than the size of the random number

generator 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. Seeding a 128-bit PRNG would

thus require approximately 100 such timer values.

[RANDOM] provides guidance on the generation of random values.

D.2. 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.3. Cipher Suites

TLS supports a range of key sizes and security levels, including some

that provide no or minimal security. A proper implementation will

probably not support many cipher suites. For instance, 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.

D.4. Implementation Pitfalls

Implementation experience has shown that certain parts of earlier TLS

specifications are not easy to understand, and have been a source of

interoperability and security problems. Many of these areas have

Dierks & Rescorla Standards Track [Page 85]

RFC 5246 TLS August 2008

been clarified in this document, but this appendix contains a short

list of the most important things that require special attention from

implementors.

TLS protocol issues:

- Do you correctly handle handshake messages that are fragmented to

multiple TLS records (see Section 6.2.1)? Including corner cases

like a ClientHello that is split to several small fragments? Do

you fragment handshake messages that exceed the maximum fragment

size? In particular, the certificate and certificate request

handshake messages can be large enough to require fragmentation.

- Do you ignore the TLS record layer version number in all TLS

records before ServerHello (see Appendix E.1)?

- Do you handle TLS extensions in ClientHello correctly, including

omitting the extensions field completely?

- Do you support renegotiation, both client and server initiated?

While renegotiation is an optional feature, supporting it is

highly recommended.

- When the server has requested a client certificate, but no

suitable certificate is available, do you correctly send an empty

Certificate message, instead of omitting the whole message (see

Section 7.4.6)?

Cryptographic details:

- In the RSA-encrypted Premaster Secret, do you correctly send and

verify the version number? When an error is encountered, do you

continue the handshake to avoid the Bleichenbacher attack (see

Section 7.4.7.1)?

- What countermeasures do you use to prevent timing attacks against

RSA decryption and signing operations (see Section 7.4.7.1)?

- When verifying RSA signatures, do you accept both NULL and missing

parameters (see Section 4.7)? Do you verify that the RSA padding

doesn't have additional data after the hash value? [FI06]

- When using Diffie-Hellman key exchange, do you correctly strip

leading zero bytes from the negotiated key (see Section 8.1.2)?

- Does your TLS client check that the Diffie-Hellman parameters sent

by the server are acceptable (see Section F.1.1.3)?

Dierks & Rescorla Standards Track [Page 86]

RFC 5246 TLS August 2008

- How do you generate unpredictable IVs for CBC mode ciphers (see

Section 6.2.3.2)?

- Do you accept long CBC mode padding (up to 255 bytes; see Section

6.2.3.2)?

- How do you address CBC mode timing attacks (Section 6.2.3.2)?

- Do you use a strong and, most importantly, properly seeded random

number generator (see Appendix D.1) for generating the premaster

secret (for RSA key exchange), Diffie-Hellman private values, the

DSA "k" parameter, and other security-critical values?

Appendix E. Backward Compatibility

E.1. Compatibility with TLS 1.0/1.1 and SSL 3.0

Since there are various versions of TLS (1.0, 1.1, 1.2, and any

future versions) and SSL (2.0 and 3.0), means are needed to negotiate

the specific protocol version to use. The TLS protocol provides a

built-in mechanism for version negotiation so as not to bother other

protocol components with the complexities of version selection.

TLS versions 1.0, 1.1, and 1.2, and SSL 3.0 are very similar, and use

compatible ClientHello messages; thus, supporting all of them is

relatively easy. Similarly, servers can easily handle clients trying

to use future versions of TLS as long as the ClientHello format

remains compatible, and the client supports the highest protocol

version available in the server.

A TLS 1.2 client who wishes to negotiate with such older servers will

send a normal TLS 1.2 ClientHello, containing { 3, 3 } (TLS 1.2) in

ClientHello.client\_version. If the server does not support this

version, it will respond with a ServerHello containing an older

version number. If the client agrees to use this version, the

negotiation will proceed as appropriate for the negotiated protocol.

If the version chosen by the server is not supported by the client

(or not acceptable), the client MUST send a "protocol\_version" alert

message and close the connection.

If a TLS server receives a ClientHello containing a version number

greater than the highest version supported by the server, it MUST

reply according to the highest version supported by the server.

A TLS server can also receive a ClientHello containing a version

number smaller than the highest supported version. If the server

wishes to negotiate with old clients, it will proceed as appropriate

Dierks & Rescorla Standards Track [Page 87]

RFC 5246 TLS August 2008

for the highest version supported by the server that is not greater

than ClientHello.client\_version. For example, if the server supports

TLS 1.0, 1.1, and 1.2, and client\_version is TLS 1.0, the server will

proceed with a TLS 1.0 ServerHello. If server supports (or is

willing to use) only versions greater than client\_version, it MUST

send a "protocol\_version" alert message and close the connection.

Whenever a client already knows the highest protocol version known to

a server (for example, when resuming a session), it SHOULD initiate

the connection in that native protocol.

Note: some server implementations are known to implement version

negotiation incorrectly. For example, there are buggy TLS 1.0

servers that simply close the connection when the client offers a

version newer than TLS 1.0. Also, it is known that some servers will

refuse the connection if any TLS extensions are included in

ClientHello. Interoperability with such buggy servers is a complex

topic beyond the scope of this document, and may require multiple

connection attempts by the client.

Earlier versions of the TLS specification were not fully clear on

what the record layer version number (TLSPlaintext.version) should

contain when sending ClientHello (i.e., before it is known which

version of the protocol will be employed). Thus, TLS servers

compliant with this specification MUST accept any value {03,XX} as

the record layer version number for ClientHello.

TLS clients that wish to negotiate with older servers MAY send any

value {03,XX} as the record layer version number. Typical values

would be {03,00}, the lowest version number supported by the client,

and the value of ClientHello.client\_version. No single value will

guarantee interoperability with all old servers, but this is a

complex topic beyond the scope of this document.

E.2. Compatibility with SSL 2.0

TLS 1.2 clients that wish to support SSL 2.0 servers MUST send

version 2.0 CLIENT-HELLO messages defined in [SSL2]. The message

MUST contain the same version number as would be used for ordinary

ClientHello, and MUST encode the supported TLS cipher suites in the

CIPHER-SPECS-DATA field as described below.

Warning: The ability to send version 2.0 CLIENT-HELLO messages will

be phased out with all due haste, since the newer ClientHello format

provides better mechanisms for moving to newer versions and

negotiating extensions. TLS 1.2 clients SHOULD NOT support SSL 2.0.

Dierks & Rescorla Standards Track [Page 88]

RFC 5246 TLS August 2008

However, even TLS servers that do not support SSL 2.0 MAY accept

version 2.0 CLIENT-HELLO messages. The message is presented below in

sufficient detail for TLS server implementors; the true definition is

still assumed to be [SSL2].

For negotiation purposes, 2.0 CLIENT-HELLO is interpreted the same

way as a ClientHello with a "null" compression method and no

extensions. Note that this message MUST be sent directly on the

wire, not wrapped as a TLS record. For the purposes of calculating

Finished and CertificateVerify, the msg\_length field is not

considered to be a part of the handshake message.

uint8 V2CipherSpec[3];

struct {

uint16 msg\_length;

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];

opaque challenge[V2ClientHello.challenge\_length;

} V2ClientHello;

msg\_length

The highest bit MUST be 1; the remaining bits contain the length

of the following data in bytes.

msg\_type

This field, in conjunction with the version field, identifies a

version 2 ClientHello message. The value MUST be 1.

version

Equal to ClientHello.client\_version.

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 zero for a client that claims to

support TLS 1.2.

Dierks & Rescorla Standards Track [Page 89]

RFC 5246 TLS August 2008

challenge\_length

The length in bytes of the client's challenge to the server to

authenticate itself. Historically, permissible values are between

16 and 32 bytes inclusive. When using the SSLv2 backward-

compatible handshake the client SHOULD use a 32-byte challenge.

cipher\_specs

This is a list of all CipherSpecs the client is willing and able

to use. In addition to the 2.0 cipher specs defined in [SSL2],

this includes the TLS cipher suites normally sent in

ClientHello.cipher\_suites, with each cipher suite prefixed by a

zero byte. For example, the TLS cipher suite {0x00,0x0A} would be

sent as {0x00,0x00,0x0A}.

session\_id

This field MUST be empty.

challenge

Corresponds to ClientHello.random. If the challenge length is

less than 32, the TLS server will pad the data with leading (note:

not trailing) zero bytes to make it 32 bytes long.

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

E.3. Avoiding Man-in-the-Middle Version Rollback

When TLS clients fall back to Version 2.0 compatibility mode, they

MUST use special PKCS#1 block formatting. This is done so that TLS

servers will reject Version 2.0 sessions with TLS-capable clients.

When a client negotiates SSL 2.0 but also supports TLS, it MUST 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).

When a TLS-capable server negotiates SSL 2.0 it SHOULD, after

decrypting the ENCRYPTED-KEY-DATA field, check that these 8 padding

bytes are 0x03. If they are not, the server SHOULD generate a random

value for SECRET-KEY-DATA, and continue the handshake (which will

eventually fail since the keys will not match). Note that reporting

the error situation to the client could make the server vulnerable to

attacks described in [BLEI].

Dierks & Rescorla Standards Track [Page 90]

RFC 5246 TLS August 2008

Appendix 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 cipher spec 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 Finished messages, encryption keys, and MAC keys (see

Sections 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 Diffie-Hellman

for key exchange. The server's public parameters are contained in

the server key exchange message, and the client's are sent in the

Dierks & Rescorla Standards Track [Page 91]

RFC 5246 TLS August 2008

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 is contained in the server's certificate. Note that

compromise of the server's static RSA key results in a loss of

confidentiality for all sessions protected under that static key.

TLS users desiring Perfect Forward Secrecy should use DHE cipher

suites. The damage done by exposure of a private key can be limited

by changing one's private key (and certificate) frequently.

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 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

use the server key exchange message to send a set of temporary

Diffie-Hellman parameters signed with a DSA 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.,

Dierks & Rescorla Standards Track [Page 92]

RFC 5246 TLS August 2008

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 DSA 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.

If the same DH keypair is to be used for multiple handshakes, either

because the client or server has a certificate containing a fixed DH

keypair or because the server is reusing DH keys, care must be taken

to prevent small subgroup attacks. Implementations SHOULD follow the

guidelines found in [SUBGROUP].

Small subgroup attacks are most easily avoided by using one of the

DHE cipher suites and generating a fresh DH private key (X) for each

handshake. If a suitable base (such as 2) is chosen, g^X mod p can

be computed very quickly; therefore, the performance cost is

minimized. Additionally, using a fresh key for each handshake

provides Perfect Forward Secrecy. Implementations SHOULD generate a

new X for each handshake when using DHE cipher suites.

Because TLS allows the server to provide arbitrary DH groups, the

client should verify that the DH group is of suitable size as defined

by local policy. The client SHOULD also verify that the DH public

exponent appears to be of adequate size. [KEYSIZ] provides a useful

guide to the strength of various group sizes. The server MAY choose

to assist the client by providing a known group, such as those

defined in [IKEALG] or [MODP]. These can be verified by simple

comparison.

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.

Altering the padding of the least-significant 8 bytes of the PKCS

Dierks & Rescorla Standards Track [Page 93]

RFC 5246 TLS August 2008

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.

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 keys 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.

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.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 key, the sequence number, the message length, the

Dierks & Rescorla Standards Track [Page 94]

RFC 5246 TLS August 2008

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 keys. 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 keys may be larger than encryption keys, so messages can

remain tamper resistant even if encryption keys are broken.

F.3. Explicit IVs

[CBCATT] describes a chosen plaintext attack on TLS that depends on

knowing the IV for a record. Previous versions of TLS [TLS1.0] used

the CBC residue of the previous record as the IV and therefore

enabled this attack. This version uses an explicit IV in order to

protect against this attack.

F.4. Security of Composite Cipher Modes

TLS secures transmitted application data via the use of symmetric

encryption and authentication functions defined in the negotiated

cipher suite. The objective is to protect both the integrity and

confidentiality of the transmitted data from malicious actions by

active attackers in the network. It turns out that the order in

which encryption and authentication functions are applied to the data

plays an important role for achieving this goal [ENCAUTH].

The most robust method, called encrypt-then-authenticate, first

applies encryption to the data and then applies a MAC to the

ciphertext. This method ensures that the integrity and

confidentiality goals are obtained with ANY pair of encryption and

MAC functions, provided that the former is secure against chosen

plaintext attacks and that the MAC is secure against chosen-message

attacks. TLS uses another method, called authenticate-then-encrypt,

in which first a MAC is computed on the plaintext and then the

concatenation of plaintext and MAC is encrypted. This method has

been proven secure for CERTAIN combinations of encryption functions

and MAC functions, but it is not guaranteed to be secure in general.

Dierks & Rescorla Standards Track [Page 95]

RFC 5246 TLS August 2008

In particular, it has been shown that there exist perfectly secure

encryption functions (secure even in the information-theoretic sense)

that combined with any secure MAC function, fail to provide the

confidentiality goal against an active attack. Therefore, new cipher

suites and operation modes adopted into TLS need to be analyzed under

the authenticate-then-encrypt method to verify that they achieve the

stated integrity and confidentiality goals.

Currently, the security of the authenticate-then-encrypt method has

been proven for some important cases. One is the case of stream

ciphers in which a computationally unpredictable pad of the length of

the message, plus the length of the MAC tag, is produced using a

pseudorandom generator and this pad is exclusive-ORed with the

concatenation of plaintext and MAC tag. The other is the case of CBC

mode using a secure block cipher. In this case, security can be

shown if one applies one CBC encryption pass to the concatenation of

plaintext and MAC and uses a new, independent, and unpredictable IV

for each new pair of plaintext and MAC. In versions of TLS prior to

1.1, CBC mode was used properly EXCEPT that it used a predictable IV

in the form of the last block of the previous ciphertext. This made

TLS open to chosen plaintext attacks. This version of the protocol

is immune to those attacks. For exact details in the encryption

modes proven secure, see [ENCAUTH].

F.5. Denial of Service

TLS is susceptible to a number of denial-of-service (DoS) attacks.

In particular, an attacker who initiates a large number of TCP

connections can cause a server to consume large amounts of CPU for

doing RSA decryption. However, because TLS is generally used over

TCP, it is difficult for the attacker to hide his point of origin if

proper TCP SYN randomization is used [SEQNUM] by the TCP stack.

Because TLS runs over TCP, it is also susceptible to a number of DoS

attacks on individual connections. In particular, attackers can

forge RSTs, thereby terminating connections, or forge partial TLS

records, thereby causing the connection to stall. These attacks

cannot in general be defended against by a TCP-using protocol.

Implementors or users who are concerned with this class of attack

should use IPsec AH [AH] or ESP [ESP].

F.6. 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.

Dierks & Rescorla Standards Track [Page 96]

RFC 5246 TLS August 2008

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 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.

Normative References

[AES] National Institute of Standards and Technology,

"Specification for the Advanced Encryption Standard (AES)"

FIPS 197. November 26, 2001.

[3DES] National Institute of Standards and Technology,

"Recommendation for the Triple Data Encryption Algorithm

(TDEA) Block Cipher", NIST Special Publication 800-67, May

2004.

[DSS] NIST FIPS PUB 186-2, "Digital Signature Standard",

National Institute of Standards and Technology, U.S.

Department of Commerce, 2000.

[HMAC] Krawczyk, H., Bellare, M., and R. Canetti, "HMAC: Keyed-

Hashing for Message Authentication", RFC 2104, February

1997.

[MD5] Rivest, R., "The MD5 Message-Digest Algorithm", RFC 1321,

April 1992.

[PKCS1] Jonsson, J. and B. Kaliski, "Public-Key Cryptography

Standards (PKCS) #1: RSA Cryptography Specifications

Version 2.1", RFC 3447, February 2003.

[PKIX] Housley, R., Polk, W., Ford, W., and D. Solo, "Internet

X.509 Public Key Infrastructure Certificate and

Certificate Revocation List (CRL) Profile", RFC 3280,

April 2002.

[SCH] B. Schneier. "Applied Cryptography: Protocols, Algorithms,

and Source Code in C, 2nd ed.", Published by John Wiley &

Sons, Inc. 1996.

[SHS] NIST FIPS PUB 180-2, "Secure Hash Standard", National

Institute of Standards and Technology, U.S. Department of

Commerce, August 2002.

Dierks & Rescorla Standards Track [Page 97]

RFC 5246 TLS August 2008

[REQ] Bradner, S., "Key words for use in RFCs to Indicate

Requirement Levels", BCP 14, RFC 2119, March 1997.

[RFC2434] Narten, T. and H. Alvestrand, "Guidelines for Writing an

IANA Considerations Section in RFCs", BCP 26, RFC 2434,

October 1998.

[X680] ITU-T Recommendation X.680 (2002) | ISO/IEC 8824-1:2002,

Information technology - Abstract Syntax Notation One

(ASN.1): Specification of basic notation.

[X690] ITU-T Recommendation X.690 (2002) | ISO/IEC 8825-1:2002,

Information technology - ASN.1 encoding Rules:

Specification of Basic Encoding Rules (BER), Canonical

Encoding Rules (CER) and Distinguished Encoding Rules

(DER).

Informative References

[AEAD] McGrew, D., "An Interface and Algorithms for Authenticated

Encryption", RFC 5116, January 2008.

[AH] Kent, S., "IP Authentication Header", RFC 4302, December

2005.

[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.

[CBCATT] Moeller, B., "Security of CBC Ciphersuites in SSL/TLS:

Problems and Countermeasures",

http://www.openssl.org/~bodo/tls-cbc.txt.

[CBCTIME] Canvel, B., Hiltgen, A., Vaudenay, S., and M. Vuagnoux,

"Password Interception in a SSL/TLS Channel", Advances in

Cryptology -- CRYPTO 2003, LNCS vol. 2729, 2003.

[CCM] "NIST Special Publication 800-38C: The CCM Mode for

Authentication and Confidentiality",

http://csrc.nist.gov/publications/nistpubs/800-38C/

SP800-38C.pdf

[DES] National Institute of Standards and Technology, "Data

Encryption Standard (DES)", FIPS PUB 46-3, October 1999.

Dierks & Rescorla Standards Track [Page 98]

RFC 5246 TLS August 2008

[DSS-3] NIST FIPS PUB 186-3 Draft, "Digital Signature Standard",

National Institute of Standards and Technology, U.S.

Department of Commerce, 2006.

[ECDSA] American National Standards Institute, "Public Key

Cryptography for the Financial Services Industry: The

Elliptic Curve Digital Signature Algorithm (ECDSA)", ANS

X9.62-2005, November 2005.

[ENCAUTH] Krawczyk, H., "The Order of Encryption and Authentication

for Protecting Communications (Or: How Secure is SSL?)",

Crypto 2001.

[ESP] Kent, S., "IP Encapsulating Security Payload (ESP)", RFC

4303, December 2005.

[FI06] Hal Finney, "Bleichenbacher's RSA signature forgery based

on implementation error", ietf-openpgp@imc.org mailing

list, 27 August 2006, http://www.imc.org/ietf-openpgp/

mail-archive/msg14307.html.

[GCM] Dworkin, M., NIST Special Publication 800-38D,

"Recommendation for Block Cipher Modes of Operation:

Galois/Counter Mode (GCM) and GMAC", November 2007.

[IKEALG] Schiller, J., "Cryptographic Algorithms for Use in the

Internet Key Exchange Version 2 (IKEv2)", RFC 4307,

December 2005.

[KEYSIZ] Orman, H. and P. Hoffman, "Determining Strengths For

Public Keys Used For Exchanging Symmetric Keys", BCP 86,

RFC 3766, April 2004.

[KPR03] Klima, V., Pokorny, O., Rosa, T., "Attacking RSA-based

Sessions in SSL/TLS", http://eprint.iacr.org/2003/052/,

March 2003.

[MODP] Kivinen, T. and M. Kojo, "More Modular Exponential (MODP)

Diffie-Hellman groups for Internet Key Exchange (IKE)",

RFC 3526, May 2003.

[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.

Dierks & Rescorla Standards Track [Page 99]

RFC 5246 TLS August 2008

[RANDOM] Eastlake, D., 3rd, Schiller, J., and S. Crocker,

"Randomness Requirements for Security", BCP 106, RFC 4086,

June 2005.

[RFC3749] Hollenbeck, S., "Transport Layer Security Protocol

Compression Methods", RFC 3749, May 2004.

[RFC4366] Blake-Wilson, S., Nystrom, M., Hopwood, D., Mikkelsen, J.,

and T. Wright, "Transport Layer Security (TLS)

Extensions", RFC 4366, April 2006.

[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.

[SEQNUM] Bellovin, S., "Defending Against Sequence Number Attacks",

RFC 1948, May 1996.

[SSL2] Hickman, Kipp, "The SSL Protocol", Netscape Communications

Corp., Feb 9, 1995.

[SSL3] A. Freier, P. Karlton, and P. Kocher, "The SSL 3.0

Protocol", Netscape Communications Corp., Nov 18, 1996.

[SUBGROUP] Zuccherato, R., "Methods for Avoiding the "Small-Subgroup"

Attacks on the Diffie-Hellman Key Agreement Method for

S/MIME", RFC 2785, March 2000.

[TCP] Postel, J., "Transmission Control Protocol", STD 7, RFC

793, September 1981.

[TIMING] Boneh, D., Brumley, D., "Remote timing attacks are

practical", USENIX Security Symposium 2003.

[TLSAES] Chown, P., "Advanced Encryption Standard (AES)

Ciphersuites for Transport Layer Security (TLS)", RFC

3268, June 2002.

[TLSECC] Blake-Wilson, S., Bolyard, N., Gupta, V., Hawk, C., and B.

Moeller, "Elliptic Curve Cryptography (ECC) Cipher Suites

for Transport Layer Security (TLS)", RFC 4492, May 2006.

[TLSEXT] Eastlake, D., 3rd, "Transport Layer Security (TLS)

Extensions: Extension Definitions", Work in Progress,

February 2008.

Dierks & Rescorla Standards Track [Page 100]

RFC 5246 TLS August 2008

[TLSPGP] Mavrogiannopoulos, N., "Using OpenPGP Keys for Transport

Layer Security (TLS) Authentication", RFC 5081, November

2007.

[TLSPSK] Eronen, P., Ed., and H. Tschofenig, Ed., "Pre-Shared Key

Ciphersuites for Transport Layer Security (TLS)", RFC

4279, December 2005.

[TLS1.0] Dierks, T. and C. Allen, "The TLS Protocol Version 1.0",

RFC 2246, January 1999.

[TLS1.1] Dierks, T. and E. Rescorla, "The Transport Layer Security

(TLS) Protocol Version 1.1", RFC 4346, April 2006.

[X501] ITU-T Recommendation X.501: Information Technology - Open

Systems Interconnection - The Directory: Models, 1993.

[XDR] Eisler, M., Ed., "XDR: External Data Representation

Standard", STD 67, RFC 4506, May 2006.

Working Group Information

The discussion list for the IETF TLS working group is located at the

e-mail address <tls@ietf.org>. Information on the group and

information on how to subscribe to the list is at

<https://www1.ietf.org/mailman/listinfo/tls>

Archives of the list can be found at:

<http://www.ietf.org/mail-archive/web/tls/current/index.html>

Contributors

Christopher Allen (co-editor of TLS 1.0)

Alacrity Ventures

ChristopherA@AlacrityManagement.com

Martin Abadi

University of California, Santa Cruz

abadi@cs.ucsc.edu

Steven M. Bellovin

Columbia University

smb@cs.columbia.edu

Simon Blake-Wilson

BCI

sblakewilson@bcisse.com

Dierks & Rescorla Standards Track [Page 101]

RFC 5246 TLS August 2008

Ran Canetti

IBM

canetti@watson.ibm.com

Pete Chown

Skygate Technology Ltd

pc@skygate.co.uk

Taher Elgamal

taher@securify.com

Securify

Pasi Eronen

pasi.eronen@nokia.com

Nokia

Anil Gangolli

anil@busybuddha.org

Kipp Hickman

Alfred Hoenes

David Hopwood

Independent Consultant

david.hopwood@blueyonder.co.uk

Phil Karlton (co-author of SSLv3)

Paul Kocher (co-author of SSLv3)

Cryptography Research

paul@cryptography.com

Hugo Krawczyk

IBM

hugo@ee.technion.ac.il

Jan Mikkelsen

Transactionware

janm@transactionware.com

Magnus Nystrom

RSA Security

magnus@rsasecurity.com

Robert Relyea

Netscape Communications

relyea@netscape.com

Dierks & Rescorla Standards Track [Page 102]

RFC 5246 TLS August 2008

Jim Roskind

Netscape Communications

jar@netscape.com

Michael Sabin

Dan Simon

Microsoft, Inc.

dansimon@microsoft.com

Tom Weinstein

Tim Wright

Vodafone

timothy.wright@vodafone.com

Editors' Addresses

Tim Dierks

Independent

EMail: tim@dierks.org

Eric Rescorla

RTFM, Inc.

EMail: ekr@rtfm.com

Dierks & Rescorla Standards Track [Page 103]

RFC 5246 TLS August 2008

Full Copyright Statement

Copyright (C) The IETF Trust (2008).

This document is subject to the rights, licenses and restrictions

contained in BCP 78, and except as set forth therein, the authors

retain all their rights.

This document and the information contained herein are provided on an

"AS IS" basis and THE CONTRIBUTOR, THE ORGANIZATION HE/SHE REPRESENTS

OR IS SPONSORED BY (IF ANY), THE INTERNET SOCIETY, THE IETF TRUST AND

THE INTERNET ENGINEERING TASK FORCE DISCLAIM 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.

Intellectual Property

The IETF takes no position regarding the validity or scope of any

Intellectual Property Rights or other rights that might be claimed to

pertain to the implementation or use of the technology described in

this document or the extent to which any license under such rights

might or might not be available; nor does it represent that it has

made any independent effort to identify any such rights. Information

on the procedures with respect to rights in RFC documents can be

found in BCP 78 and BCP 79.

Copies of IPR disclosures made to the IETF Secretariat and any

assurances of licenses to be made available, or the result of an

attempt made to obtain a general license or permission for the use of

such proprietary rights by implementers or users of this

specification can be obtained from the IETF on-line IPR repository at

http://www.ietf.org/ipr.

The IETF invites any interested party to bring to its attention any

copyrights, patents or patent applications, or other proprietary

rights that may cover technology that may be required to implement

this standard. Please address the information to the IETF at

ietf-ipr@ietf.org.

Dierks & Rescorla Standards Track [Page 104]