WAP WTLS
Version 30-Apr-1998

Wireless Application Protocol
Wireless Transport Layer Security Specification

Disclaimer:

This document is subject to change without notice.

Contents

1 SCOPE **4**

2 DOCUMENT STATUS **5**

2.1 COPYRIGHT NOTICE 5
2.2 ERRATA 5
2.3 COMMENTS 5

3 REFERENCES **6**

3.1 NORMATIVE REFERENCES 6
3.2 INFORMATIVE REFERENCES 7
3.3 ACKNOWLEDGEMENTS 7

4 DEFINITIONS AND ABBREVIATIONS **8**

4.1 DEFINITIONS 8
4.2 ABBREVIATIONS 9
4.3 DOCUMENT CONVENTIONS 10

5 WTLS ARCHITECTURAL OVERVIEW **11**

5.1 REFERENCE MODEL 11

6 WTLS ELEMENTS FOR LAYER-TO-LAYER COMMUNICATION **12**

6.1 NOTATIONS USED 12
6.1.1 Definition of Service Primitives and Parameters 12
6.1.2 Time Sequence Charts 12
6.1.3 Primitive Types 13
6.1.4 Service Parameter Tables 13
6.2 WTLS TRANSPORT SERVICE 14
6.2.1 Service Primitives 14
6.3 WTLS CONNECTION MANAGEMENT 14
6.3.1 Overview 14
6.3.2 Service Primitives 15
6.3.3 Constraints on Using the Service Primitives 18

7 WTLS STATE TABLES **20**

7.1 CLIENT STATE TABLES 20
7.2 SERVER STATE TABLES 23

8 PRESENTATION LANGUAGE **27**

8.1 BASIC BLOCK SIZE 27
8.2 MISCELLANEOUS 27
8.3 VECTORS 27
8.4 NUMBERS 28
8.5 ENUMERATEDS 28
8.6 CONSTRUCTED TYPES 29
8.6.1 Variants 29
8.7 CRYPTOGRAPHIC ATTRIBUTES 30
8.8 CONSTANTS 31
8.9 STRING CONSTANTS 31

9 RECORD PROTOCOL SPECIFICATION **32**

9.1 CONNECTION STATE 32
9.2 RECORD LAYER 35
9.2.1 Fragmentation 35
9.2.2 Record Compression and Decompression 37
9.2.3 Record Payload Protection 38

10 HANDSHAKE PROTOCOL SPECIFICATION 41

10.1 CHANGE CIPHER SPEC PROTOCOL 41
10.2 ALERT PROTOCOL 42
10.2.1 Closure Alerts 43
10.2.2 Error Alerts 44
10.3 HANDSHAKE PROTOCOL OVERVIEW 46
10.4 HANDSHAKE RELIABILITY OVER DATAGRAMS 48
10.5 HANDSHAKE PROTOCOL 49
10.5.1 Hello Messages 50
10.5.2 Server Certificate 57
10.5.3 Server Key Exchange Message 60
10.5.4 Certificate Request 61
10.5.5 Server Hello Done 62
10.5.6 Client Certificate 62
10.5.7 Client Key Exchange Message 62
10.5.8 Certificate Verify 64
10.5.9 Finished 64

11 CRYPTOGRAPHIC COMPUTATIONS 66

11.1 COMPUTING THE MASTER SECRET 66
11.1.1 RSA Encryption Scheme 66
11.1.2 Diffie-Hellman 66
11.1.3 EC Diffie-Hellman 66
11.1.4 Session resume 67
11.2 KEY CALCULATION 67
11.3 HMAC AND THE PSEUDORANDOM FUNCTION 69
11.3.1 MAC Calculation 69
11.3.2 Pseudo-random Function 69

APPENDIX A ALGORITHM DEFINITIONS 71

APPENDIX B IMPLEMENTATION NOTES 81

B.1 NEGOTIATING NULL CIPHER SPEC 81
B.2 ANONYMOUS HANDSHAKES 81
B.3 KEY REFRESH 81
B.4 DENIAL-OF-SERVICE ATTACKS 82

APPENDIX C IMPLEMENTATION CLASSES 83

APPENDIX D REQUIREMENTS FOR THE WTLS PROTOCOL 84

1 Scope

The Wireless Application Protocol (WAP) is a result of continuous work to define an industry-wide specification for developing applications that operate over wireless communication networks. The scope for the WAP Forum is to define a set of specifications to be used by service applications. The wireless market is growing very quickly, and reaching new customers and services. To enable operators and manufacturers to meet the challenges in advanced services, differentiation and fast/flexible service creation WAP Forum defines a set of protocols in transport, security, transaction, session and application layers. For additional information on the WAP architecture, please refer to *"Wireless Application Protocol Architecture Specification"* [WAPARCH].

The Security layer protocol in the WAP architecture is called the Wireless Transport Layer Security, WTLS. The WTLS layer operates above the transport protocol layer. The WTLS layer is modular and it depends on the required security level of the given application whether it is used or not. WTLS provides the upper-level layer of WAP with a secure transport service interface that preserves the transport service interface below it. In addition, WTLS provides an interface for managing (eg, creating and terminating) secure connections.

The primary goal of the WTLS layer is to provide privacy, data integrity and authentication between two communicating applications. WTLS provides functionality similar to TLS 1.0 and incorporates new features such as datagram support, optimised handshake and dynamic key refreshing. The WTLS protocol is optimised for low-bandwidth bearer networks with relatively long latency.

2 Document Status

This document is available online in the following formats:

- PDF format at http://www.wapforum.org/.

2.1 Copyright Notice

2.2 Errata

Known problems associated with this document are published at http://www.wapforum.org/.

2.3 Comments

Comments regarding this document can be submitted to WAP Forum in the manner published at http://www.wapforum.org/.

3 References

3.1 Normative References

[WAPARCH] “WAP Architecture Specification, WAP Forum, 30-April-1998.
URL: http://www.wapforum.org/

[WAPWDP] “Wireless Datagram Protocol Specification”, WAP Forum, 30-April-1998.
URL: http://www.wapforum.org/

[WAPWTP] “Wireless Transaction Protocol Specification”, WAP Forum, 30-April-1998.
URL: http://www.wapforum.org/

[RFC2119] “Key words for use in RFCs to Indicate Requirement Levels”, Bradner, S., March 1997.
URL: ftp://ftp.isi.edu/in-notes/rfc2119

[TLS] “The TLS Protocol”, Dierks, T. and Allen, C., November 1997.
URL: ftp://ftp.ietf.org/internet-drafts/draft-ietf-tls-protocol-05.txt

[RFC2068] “Hypertext Transfer Protocol -- HTTP/1.1”, Fielding, R., et. al., January 1997.
URL: ftp://ftp.isi.edu/in-notes/rfc2068

[HMAC] “HMAC: Keyed-Hashing for Message Authentication”, Krawczyk, H., Bellare, M., and Canetti, R., RFC 2104, February 1997. URL: ftp://ftp.isi.edu/in-notes/rfc2104.txt

[SHA] “Secure Hash Standard”, NIST FIPS PUB 180-1, National Institute of Standards and Technology, U.S. Department of Commerce, DRAFT, May 1994.

[X509] “The Directory – Authentication Framework”, CCITT, Recommendation X.509, 1988.

[3DES] “Hellman Presents No Shortcut Solutions To DES”, Tuchman, W., IEEE Spectrum, v. 16, n. 7, July 1979, pp 40-41.

[DES] “American National Standard for Information Systems-Data Link Encryption”, ANSI X3.106, American National Standards Institute, 1983.

[DH1] “New Directions in Cryptography”, Diffie, W. and Hellman M. E., IEEE Transactions on Information Theory, V. IT-22, n. 6, Jun 1977, pp. 74-84.

[DSS] “Digital Signature Standard”, NIST FIPS PUB 186, National Institute of Standards and Technology, U.S. Department of Commerce, May 1994.

[IDEA] “On the Design and Security of Block Ciphers”, Lai, X., ETH Series in Information Processing, v. 1, Konstanz: Hartung-Gorre Verlag, 1992.

[MD5] “The MD5 Message Digest Algorithm”, Rivest, R., RFC 2104, April 1992.
URL: ftp://ftp.isi.edu/in-notes/rfc2104.txt

[PKCS1] “PKCS #1: RSA Encryption Standard”, version 1.5, RSA Laboratories, November 1993.

[RSA] “A Method for Obtaining Digital Signatures and Public-Key Cryptosystems”, Rivest, R., Shamir, A. and Adleman L.M., Communications of the ACM, v. 21, n. 2, Feb 1978, pp. 120-126.

[RC5] "The RC5, RC5-CBC, RC5-CBC-Pad, and RC5-CTS Algorithms", Baldwin, R. and Rivest R., RFC 2040, October 1996. URL: ftp://ftp.isi.edu/in-notes/rfc2040.txt

[P1363] "Standard Specifications For Public Key Cryptography", IEEE P1363 / D1a (Draft Version 1a), February 1998. URL: http://grouper.ieee.org/groups/1363/

[X9.62] "The Elliptic Curve Digital Signature Algorithm (ECDSA)", ANSI X9.62 Working Draft, November 1997.

3.2 Informative References

[WAPWSP] "Wireless Session Protocol Specification", WAP Forum, 30-April-1998.
URL: http://www.wapforum.org/

[GSM03.40] "European Digital Cellular Telecommunication System (phase 2+): Technical realization of Short Message Service (SMS) Point-to-Point (P)", ETSI.

[XDR] "XDR: External Data Representation Standard", Srinivansan, R., RFC-1832:, August 1995. URL: ftp://ftp.isi.edu/in-notes/rfc1832.txt

[ISO7498] "Information technology - Open Systems Interconnection - Basic Reference Model: The Basic Model", ISO/IEC 7498-1:1994.

[ISO10731] "Information Technology - Open Systems Interconnection - Basic Reference Model - Conventions for the Definition of OSI Services", ISO/IEC 10731:1994.

3.3 Acknowledgements

WTLS is derived from [TLS]. TLS is based on the SSL 3.0 specification.

4 Definitions and Abbreviations

4.1 Definitions

For the purposes of this specification the following definitions apply.

Abbreviated Handshake

A creation of a new *connection state* based on an existing *secure session*. See also *Session Resume*.

Connection State

The operating environment of the r*ecord protocol*. The connection state includes all parameters that are needed for the cryptographic operations (encryption/decryption and MAC calculation/verification). Each *secure connection* has a connection state

Datagram Transport

A transport service that does not guarantee that the sent transport SDUs are not lost, duplicated or delivered out of order.

Handshake

The procedure of agreeing on the protocol options to be used between a client and a server. It includes the negotiation of security parameters (eg, algorithms and key lengths), key exchange and authentication. Handshaking occurs in the beginning of each secure connection.

Handshake Protocol

The protocol that carries out the *handshake*.

Full Handshake

A creation of a new *secure session* between two peers. The full handshake includes the parameter negotiation and the exchange of public-key information between the client and server.

Optimised Handshake

A creation of a new *secure session* between two peers. Unlike in the *full handshake*, the server looks up the client certificate from its own source without requesting it over the air from the client.

Record

A protocol data unit (PDU) in the *record protocol* layer.

Record Protocol

The record protocol takes messages to be transmitted, optionally compresses the data, applies a MAC, encrypts and transmits the result. Received data is decrypted, verified, decompressed and then delivered to higher level clients. There are four record protocol clients described in this document: the handshake protocol, the alert protocol, the change cipher spec protocol and the application data protocol.

Secure Connection

The WTLS connection that has a *connection state*. Each secure connection is identified by the transport addresses of the communicating peers.

Secure Session

The secure session that is negotiated on a handshake. The items that are negotiated (eg, session identifier, algorithms and master secret) are used for creating *secure connections*. Each secure session is identified by a session ID allocated by the server.

Session Resume

A new *secure connection* can be established based on a previously negotiated *secure session*. So if there is an existing secure session it is not necessary to perform the full handshake and cryptographic calculations again. For example, a secure connection may be terminated and resumed later. Many secure connections can be established using the same secure session through the resumption feature of the WTLS *handshake protocol*.

Shared Secret Authentication

An authentication method based on a shared secret. This method works without public-key algorithms but requires that the premaster secret is implanted or entered manually into both client and server. The shared secret is sensitive information and, therefore, a secure channel is needed for the distribution.

4.2 Abbreviations

For the purposes of this specification the following abbreviations apply.

API	Application Programming Interface
CA	Certification Authority
CBC	Cipher Block Chaining
DH	Diffie-Hellman
EC	Elliptic Curve
ECC	Elliptic Curve Cryptography
ECDH	Elliptic Curve Diffie-Hellman
ECDSA	Elliptic Curve Digital Signature Algorithm
IV	Initialisation Vector
MAC	Message Authentication Code
ME	Management Entity
OSI	Open System Interconnection
PDU	Protocol Data Unit
PRF	Pseudo-Random Function
SAP	Service Access Point
SDU	Service Data Unit
SHA-1	Secure Hash Algorithm
SMS	Short Message Service
SSL	Secure Sockets Layer
TLS	Transport Layer Security
WAP	Wireless Application Protocol
WDP	Wireless Datagram Protocol
WSP	Wireless Session Protocol
WTLS	Wireless Transport Layer Security
WTP	Wireless Transaction Protocol

4.3 Document Conventions

This specification uses the same keywords as specified in RFC 2119 [RFC2119] for defining the significance of each particular requirement. These words are:

MUST

This word, or the terms "REQUIRED" or "SHALL", mean that the definition is an absolute requirement of the specification.

MUST NOT

This phrase, or the phrase "SHALL NOT", mean that the definition is an absolute prohibition of the specification.

SHOULD

This word, or the adjective “RECOMMENDED”, means that there may exist valid reasons in particular circumstances to ignore a particular item, but the full implications must be understood and carefully weighed before choosing a different course.

SHOULD NOT

This phrase, or the phrase "NOT RECOMMENDED" mean that there may exist valid reasons in particular circumstances when the particular behavior is acceptable or even useful, but the full implications should be understood and the case carefully weighed before implementing any behavior described with this label.

MAY

This word, or the adjective “OPTIONAL”, means that an item is truly optional. One vendor may choose to include the item because a particular marketplace requires it or because the vendor feels that it enhances the product while another vendor may omit the same item. An implementation which does not include a particular option MUST be prepared to interoperate with another implementation which does include the option, though perhaps with reduced functionality. In the same vein an implementation which does include a particular option MUST be prepared to interoperate with another implementation which does not include the option (except, of course, for the feature the option provides.)

5 WTLS Architectural Overview

5.1 Reference model

A model of layering the protocols in WAP is illustrated Figure 1: Wireless Application Protocol Reference Model. The layering of WAP protocols and their functions is similar to that of the ISO OSI Reference Model [ISO7498] for upper layers. Layer Management Entities handle protocol initialisation, configuration, and error conditions (such as loss of connectivity due to the mobile terminal roaming out of coverage) that are not handled by the protocol itself.

WTLS is designed to function on connection-oriented and/or datagram transport protocols. Security is assumed to be an optional layer above the transport layer. The security layer preserves the transport service interfaces. The session or application management entities are assumed to provide additional support required to manage (eg, initiate and terminate) secure connections.

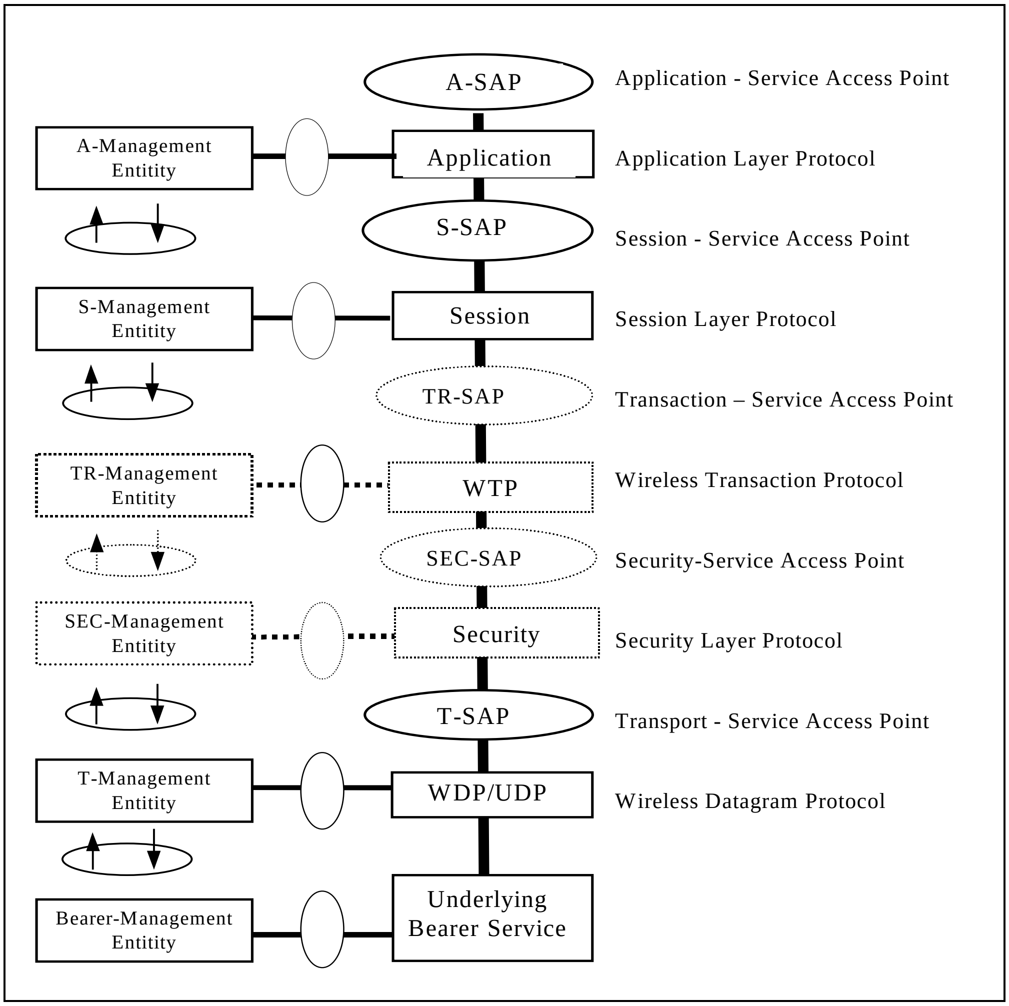
A-SAP
Application - Service Access Point
A-Management Entitity
Application
Application Layer Protocol
S-SAP
Session - Service Access Point
S-Management Entitity
Session
Session Layer Protocol
TR-SAP
Transaction – Service Access Point
TR-Management Entitity
WTP
Wireless Transaction Protocol
SEC-SAP
Security-Service Access Point
SEC-Management Entitity
Security
Security Layer Protocol
T-SAP
Transport - Service Access Point
T-Management Entitity
WDP/UDP
Wireless Datagram Protocol
Bearer-Management Entitity
Underlying Bearer Service

Figure 1: Wireless Application Protocol Reference Model

6 WTLS Elements for Layer-to-Layer Communication

6.1 Notations Used

6.1.1 Definition of Service Primitives and Parameters

Communication between layers is accomplished by means of service primitives. Service primitives represent, in an abstract way, the logical exchange of information and control between the security layer and adjacent layers.

Service primitives consist of commands and their respective responses associated with the services requested of another layer. The general syntax of a primitive is:

X-Service.type (Parameters)

where X designates the layer providing the service. For this specification X is “ SEC” for the Security layer.

Service primitives are not the same as an application programming interface (API) and are not meant to imply any specific method of implementing an API. Service primitives are an abstract means of illustrating the services provided by the protocol layer to the layer above. The mapping of these concepts to a real API and the semantics associated with a real API are an implementation issue and are beyond the scope of this specification.

6.1.2 Time Sequence Charts

The behaviour of service primitives is illustrated using time sequence charts, which are described in [ISO10731].

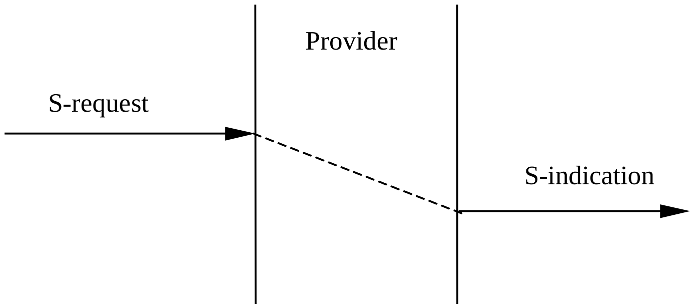

Figure 2: A Non-confirmed Service

Figure 2 illustrates a simple non-confirmed service, which is invoked using a request primitive and results in an indication primitive in the peer. The dashed line represents propagation through the provider over a period of time indicated by the vertical difference between the two arrows representing the primitives.

6.1.3 Primitive Types

The primitives types defined in this specification are:

Type	**Abbreviation**	**Description**
request	req	Used when a higher layer is requesting a service from the next lower layer
indication	ind	A layer providing a service uses this primitive type to notify the next higher layer of activities related to the request primitive type of the peer (such as the invocation of the request primitive) or to the provider of the service (such as a protocol generated event)
response	res	A layer uses the response primitive type to acknowledge receipt of the indication primitive type from the next lower layer
confirm	cnf	The layer providing the requested service uses the confirm primitive type to report that the activity has been completed successfully

6.1.4 Service Parameter Tables

The service primitives are defined using tables indicating which parameters are possible and how they are used with the different primitive types. For example, a simple confirmed primitive might be defined using the following:

Primitive	**S-primitive**			
Parameter	***req***	***ind***	***res***	***cnf***
Parameter 1	M	M(=)		
Parameter 2			O	C(=)

If some primitive type is not possible, the column for it will be omitted. The entries used in the primitive type columns are defined in the following table:

M	Presence of the parameter is mandatory – it MUST be present
C	Presence of the parameter is conditional depending on values of other parameters
O	Presence of the parameter is a user option – it MAY be omitted
P	Presence of the parameter is a service provider option – an implementation MAY not provide it
	The parameter is absent
*	Presence of the parameter is determined by the lower layer protocol
(=)	The value of the parameter is identical to the value of the corresponding parameter of the preceding service primitive

In the example table above, *Parameter 1* is always present in *S-primitive.request* and corresponding *S-primitive.indication*. *Parameter 2* MAY be specified in *S-primitive.response* and in that case it MUST be present and have the equivalent value also in the corresponding *S-primitive.confirm*; otherwise, it MUST NOT be present.

6.2 WTLS Transport Service

6.2.1 Service Primitives

6.2.1.1 SEC-Unitdata

This primitive is used to exchange user data between the peers. SEC-Unitdata can only be invoked when there is an existing secure connection between the transport addresses of the peers.

Primitive	SEC-Unitdata	
Parameter	***req***	***ind***
Source Address	M	M(=)
Source Port	M	M(=)
Destination Address	M	O(=)
Destination Port	M	O(=)
User Data	M	M(=)

Source Address identifies the originator.

Source Port identifies the port from which the message is sent.

Destination Address identifies the peer to which the user data is sent.

Destination Port identifies the port to which the message is sent.

User Data is the data to be transmitted.

6.3 WTLS Connection Management

6.3.1 Overview

WTLS Connection management allows a client to connect with a server and to agree on protocol options to be used. The secure connection establishment consists of several steps and either client or server can interrupt the negotiation at will (eg, if the parameters proposed by the peer are not acceptable). The negotiation may include the security parameters (eg, cryptographic algorithms and key lengths), key exchange and authentication. Either the server or client service user can also terminate the connection at any time.

The primitive sequence for establishing a secure session (full handshake) is shown in Figure 3.

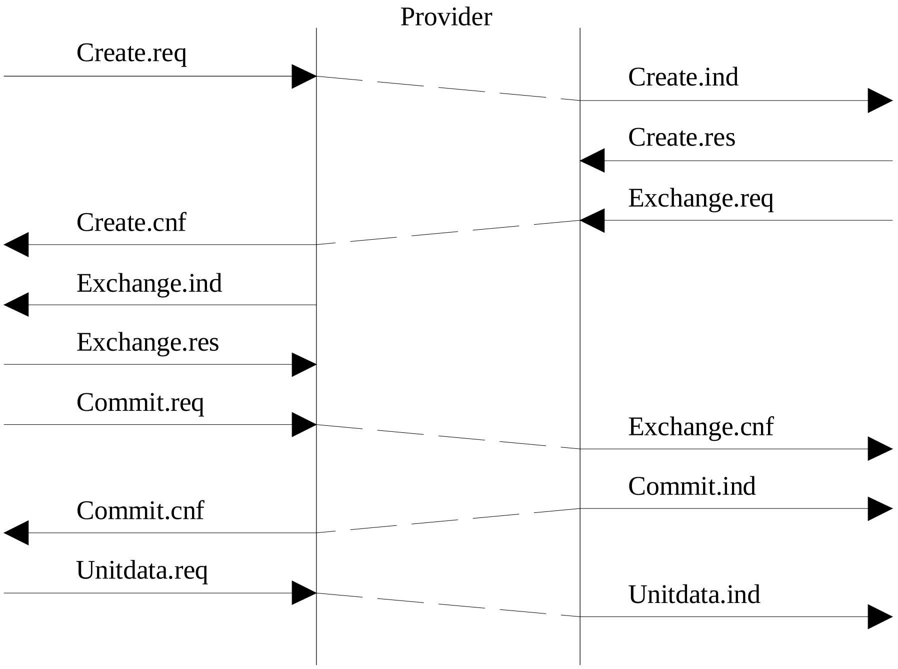

Figure 3: Full Handshake

The primitive sequence for establishing a secure session in an optimised or abbreviated way is shown in Figure 4.

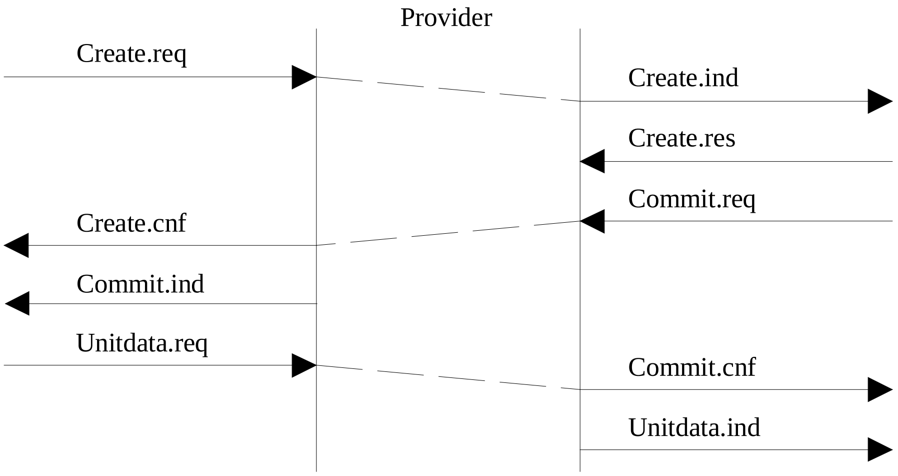

Figure 4: Abbreviated or Optimised Handshake

6.3.2 Service Primitives

6.3.2.1 SEC-Create

This primitive is used to initiate a secure connection establishment.

Primitive **Parameter**	**SEC-Create**			
	req	***ind***	***res***	***cnf***
Source Address	M	M(=)		
Source Port	M	M(=)	-	-
Destination Address	M	O(=)	-	-
Destination Port	M	O(=)		
Client Identities	O	C(=)	-	-
Proposed Key Exchange Suites	M	M(=)	-	-
Proposed Cipher Suites	M	M(=)	-	-
Proposed Compression Methods	M	M(=)	-	-
Sequence Number Mode	O	C(=)	M	M(=)
Key Refresh	O	C(=)	M	M(=)
Session Id	O	C(=)	M	M(=)
Selected Key Exchange Suite	-	-	M	M(=)
Selected Cipher Suite	-	-	M	M(=)
Selected Compression Method	-	-	M	M(=)
Server Certificate	-	-	O	C(=)

Source Address identifies the originator.

Source Port identifies the port from which the message is sent.

Destination Address identifies the peer to which the user data is sent.

Destination Port identifies the port to which the message is sent.

Client Identities identify the originator in a transport independent way. This parameter may be used by the server to look up the corresponding client certificate. Client can send several identities corresponding to different keys or certificates.

Proposed Key Exchange Suites include the key exchange suites proposed by the client.

Proposed Cipher Suites include the cipher suites proposed by the client.

Proposed Compression Methods include the compression methods proposed by the client.

Sequence Number Mode defines how sequence numbers are used in this secure connection.

Key Refresh defines how often the encryption and protection keys are refreshed within a secure connection.

Session Id identifies the secure session. It is unique per server.

Selected Key Exchange Suite identifies the key exchange suite selected by the server.

Selected Cipher Suite identifies the cipher suite selected by the server.

Selected Compression Method identifies the compression method chosen by the server.

Server Certificate is the public-key certificate of the server.

6.3.2.2 SEC-Exchange

This primitive is used in a secure connection creation if the server wishes to perform public-key authentication or key exchange with the client.

Primitive	SEC-Exchange			
Parameter	***req***	***ind***	***res***	***cnf***
Client Certificate	-	-	M	M(=)

Client Certificate is the public-key certificate of the client.

6.3.2.3 SEC-Commit

This primitive is initiated when the handshake is completed and either peer requests to switch into the newly negotiated connection state.

Primitive	SEC-Commit			
Parameter	***req***	***ind***	***res***	***cnf***
-	-	-	-	-

6.3.2.4 SEC-Terminate

This primitive is used to terminate the connection.

Primitive	SEC-Terminate	
Parameter	***req***	***ind***
Alert Description	M	M(=)
Alert Level	M	M(=)

Alert Description identifies the reason that caused the termination.

Alert Level defines whether the session (fatal) or just a connection (critical) is terminated.

6.3.2.5 SEC-Exception

This primitive is used to inform the other end about warning level alerts.

Primitive	SEC-Notify	
Parameter	***req***	***ind***
Alert Description	M	M(=)

Alert Description identifies what caused the warning.

6.3.2.6 SEC-Create-Request

This primitives is used by the server to request the client to initiate a new handshake.

Primitive	SEC-Create-Request	
Parameter	***req***	***ind***
Source Address	O	C(=)
Source Port	O	C(=)
Destination Address	O	C(=)

Destination Port	O	C(=)

Source Address identifies the originator.

Source Port identifies the port from which the message is sent.

Destination Address identifies the client to which the data is sent. This parameter is needed when the primitive is used in a NULL session state.

Destination Port identifies the port to which the data is sent.

6.3.3 Constraints on Using the Service Primitives

The following tables define the permitted primitive sequences on the service interface. The client and server have separate tables, since the service is asymmetric.

Only the permitted primitives are listed on the rows; the layer prefix is omitted for brevity. The table entries are interpreted as follows:

Table 1: Table Entry Legend

Entry:	**Description**
	The indication or confirm primitive cannot occur.
N/A	Invoking this primitive is an error. The appropriate action is a local implementation matter.
STATE_NAME	Primitive is permitted and moves the service interface view to the named state.

Table 2: Permitted Client Security Layer Primitives

CLIENT	**Session States**							
SEC-Primitive	NULL	CREATING	CREATED	EXCHANGE	COMMIT1	COMMIT2	OPENING	OPEN
Create.req	CREATING	N/A	N/A	N/A	N/A	N/A	N/A	CREATING
Commit.req	N/A	N/A	N/A	N/A	COMMIT2	N/A	N/A	N/A
Terminate.req	N/A	NULL	NULL	NULL	NULL	NULL	NULL	NULL
Exception.req	N/A	CREATING	CREATED	EXCHANGE	COMMIT1	COMMIT2	OPENING	OPEN
Unitdata.req	N/A	N/A	N/A	N/A	N/A	N/A	OPENING	OPEN
Exchange.res	N/A	N/A	N/A	COMMIT1	N/A	N/A	N/A	N/A
Exchange.ind			EXCHANGE					
Commit.ind			OPENING					
Terminate.ind		NULL				NULL	NULL	NULL
Exception.ind		CREATING				COMMIT2	OPEN	OPEN
Create-Request.ind	NULL						OPEN	OPEN
Unitdata.ind							OPEN	OPEN
Create.cnf		CREATED						
Commit.cnf						OPEN		

Table 3: Permitted Server Security Layer Primitives

SERVER	**Session States**						
SEC-Primitive	NULL	CREATING	CREATED	EXCHANGE	COMMIT	OPENING	OPEN
Exchange.req	N/A	N/A	EXCHANGE	N/A	N/A	N/A	N/A
Commit.req	N/A	N/A	COMMIT	N/A	N/A	N/A	N/A
Create-Request.req	NULL	N/A	N/A	N/A	N/A	N/A	OPEN
Terminate.req	N/A	NULL	NULL	NULL	NULL	NULL	NULL
Exception.req	N/A	CREATING	CREATED	EXCHANGE	COMMIT	OPENING	OPEN
Unitdata.req	N/A	N/A	N/A	N/A	N/A	N/A	OPEN
Create.res	N/A	CREATED	N/A	N/A	N/A	N/A	N/A
Commit.ind						OPEN	
Create.ind	CREATING			CREATING	CREATING		CREATING
Terminate.ind				NULL	NULL		NULL
Exception.ind	NULL			EXCHANGE	COMMIT		OPEN
Unitdata.ind							OPEN
Exchange.cnf				OPENING			
Commit.cnf					OPEN		

7 WTLS State Tables

The following state tables define the actions of WTLS on a datagram transport service provider.

WTLS PDUs are identified in *italics*.

By default, all WTLS PDUs will be processed under the state that is currently in use.

If any PDUs other than the ones listed under Conditions are received, the receiver may generate an alert depending on the severity of the case. See Section 10.2 for more detailed information.

Although the state tables provided are helpful to understand the WTLS protocol, they are not the formal and complete definition. Those tables tend to be concise and readable so that certain level of details are not reflected. It is therefore essential that the textual description of this specification is the unique and complete definition of the WTLS protocol.

7.1 Client State Tables

The following tables show the protocol states and event processing on the client.

Client Secure Session NULL			
Event	**Conditions**	**ACTION**	**Next State**
SEC-Create.req		T-Unitdata.req (*ClientHello*) The sequence number is present during a handshake	CREATING
T-Unitdata.ind	*HelloRequest*	SEC-Create-Request.ind The client may initiate a handshake with SEC-Create.req, initiate an *alert (no_renegotiation)* or ignore the request.	NULL

Client Secure Session CREATING

Event	Conditions	ACTION	Next State
SEC-Terminate.req		T-Unitdata.req (*Alert (fatal or critical)*)	NULL
SEC-Exception.req		T-Unitdata.req (*Alert (warning)*)	CREATING
T-Unitdata.ind	*ServerHello* *Certificate** *ServerKeyExchange** *CertificateRequest** *ServerHelloDone*	SEC-Create.cnf SEC-Exchange.ind	EXCHANGE
	ServerHello *Certificate** *ChangeCipherSpec* *Finished*	SEC-Create.cnf SEC-Commit.ind The pending state is made current by *ChangeCipherSpec* so that *Finished* is processed under the new state and the sequence numbers are set to zero	CREATED
	Alert (critical or fatal)	SEC-Terminate.ind	NULL
	Alert (warning)	SEC-Exception.ind	CREATING
Retransmission timer expires		T-Unitdata.req (*ClientHello*) The last buffer resent without incrementing the sequence number The retransmission timer is cleared The retransmission counter is incremented	CREATING
Retransmission counter exceeds the maximum value		SEC-Terminate.ind	NULL

*) Whether these messages are present or not depends on the chosen key exchange method.

Client Secure Session EXCHANGE

Event	Conditions	ACTION	Next State
SEC-Exchange.res		Create a buffer with: *ClientKeyExchange** *CertificateVerify**	COMMIT1
SEC-Terminate.req		T-Unitdata (*Alert (critical or fatal)*)	NULL
SEC-Exception.req		T-Unitdata (*Alert (warning)*)	EXCHANGE

*) Whether these messages are present or not depends on the chosen key exchange method.

Client Secure Session COMMIT1

Event	Conditions	ACTION	Next State
SEC-Commit.req		Append to the buffer: *ChangeCipherSpec* *Finished* The pending state is made current by *ChangeCipherSpec* so that *Finished* is processed under the new negotiated state and the sequence number is set to zero Send the buffer out with T-Unitdata.req	COMMIT2
SEC-Terminate.req		T-Unitdata (*Alert (critical or fatal)*)	NULL
SEC-Exception.req		T-Unitdata (*Alert (warning)*)	COMMIT1

Client Secure Session COMMIT2

Event	Conditions	ACTION	Next State
SEC-Terminate.req		T-Unitdata.req (*Alert (critical or fatal)*)	NULL
SEC-Exception.req		T-Unitdata (*Alert (warning)*)	COMMIT2
T-Unitdata.ind	*Alert (critical or fatal)*	SEC-Terminate.ind	NULL
	Alert (warning)	SEC-Exception.ind	COMMIT2
	Finished	SEC-Commit.cnf	OPEN
Retransmission timer expires	No response from the server is received	T-Unitdata.req The last buffer is resent without incrementing the sequence number The retransmission timer is cleared The retramission counter is incremented	COMMIT2
Retransmission counter exceeds the maximum value		SEC-Terminate.ind	NULL

Client Secure Session CREATED

Event	Conditions	ACTION	Next State
	Impelementation may send *Finished* immediately without user data	Create a buffer with: *Finished* Send it out with T-Unitdata.req	OPENING
	Implementation may delay sending *Finished* and prepend it to user data (if any)	Create a buffer with: *Finished* Set up a Finished prepending timer	OPENING
SEC-Terminate.req		T-Unitdata.req (*Alert (critical or fatal l)*)	NULL
SEC-Exception.req		T-Unitdata.req (*Alert (warning)*)	CREATED

Client Secure Session OPENING

Event	Conditions	ACTION	Next State
SEC-Unitdata.req		Prepend buffer to user data and call T-Unitdata.req	OPENING
SEC-Terminate.req		T-Unitdata.req (*Alert (critical or fatal)*)	NULL
SEC-Exception.req		T-Unitdata.req (*Alert (warning)*)	OPENING
	Finished prepending timer is set	Prepend buffer to user data and call T-Unitdata.req Remove the Finished prepending timer.	OPENING
Finished prepending timer expires		Send buffer out it T-Unitdata.req	OPENING
T-Unitdata.ind	User data is received	SEC-Unitdata.ind	OPEN
	Alert (duplicate_ finished_received)		OPEN
	Alert (critical or fatal)	SEC-Terminate.ind	NULL
	Alert (warning)	SEC-Exception.ind	OPEN
	HelloRequest	SEC-Create-Request.ind The client may initiate a handshake with SEC-Create.req, initiate an *alert (no_renegotiation)* or ignore the request.	OPEN

Client Secure Session OPEN			
Event	**Conditions**	**ACTION**	**Next State**
SEC-Create.req		T-Unitdata.req (*ClientHello*)	CREATING
SEC-Terminate.req		T-Unitdata.req (*Alert (critical or fatal)*)	NULL
SEC-Exception.req		T-Unitdata.req (*Alert (warning)*)	OPEN
SEC-Unitdata.req		T-Unitdata.req	OPEN
T-Unitdata.ind	User data received	SEC-Unitdata.ind	OPEN
	Alert (critical or fatal)	SEC-Terminate.ind	NULL
	Alert (warning)	SEC-Exception.ind	OPEN
	HelloRequest	SEC-Create-Request.ind The client may initiate a handshake with SEC-Create.req, initiate an *alert (no_renegotiation)* or ignore the request.	OPEN

7.2 Server State Tables

The following tables show the protocol states and event processing on the client.

Server Secure Session NULL			
Event	**Conditions**	**ACTION**	**Next State**
SEC-Create-Request.req		T-Unitdata.req (*HelloRequest*) The rate at which HelloRequests are sent should be limited.	NULL
T-Unitdata.ind	*ClientHello*	SEC-Create.ind	CREATING
	Alert (no_renegotiation)	SEC-Exception.ind	NULL

Server Secure Session CREATING			
Event	**Conditions**	**ACTION**	**Next State**
SEC-Terminate.req		T-Unitdata.req (*Alert (critical or fatal)*)	NULL
SEC-Exception.req		T-Unitdata.req (*Alert (warning)*)	CREATING
SEC-Create.res		Create a buffer with: *ServerHello* *Certificate**	CREATED

*) Whether this message is present or not depends on the chosen key exchange method.

Server Secure Session CREATED			
Event	**Conditions**	**ACTION**	**Next State**
SEC-Exchange.req	Full handshake	Append to the buffer: *ServerKeyExchange** *CertificateRequest** *ServerHelloDone* Send it out with T-Unitdata.req	EXCHANGE
SEC-Commit.req	Optimized or abbreviated handshake	Append to the buffer: *ChangeCipherSpec* *Finished* The pending state is made current by *ChangeCipherSpec* so that *Finished* is processed under the new negotiated state and sequence numbers are set to zero Send the buffer out with T-Unitdata.req	COMMIT
SEC-Terminate.req		T-Unitdata.req (*Alert (critical or fatal)*)	NULL
SEC-Exception.req		T-Unitdata.req (*Alert (warning)*)	CREATED

*) Whether these messages are present or not depends on the chosen key exchange method.

Server Secure Session EXCHANGE			
Event	**Conditions**	**ACTION**	**Next State**
SEC-Terminate.req		T-Unitdata.req (*Alert (critical or fatal)*)	NULL
SEC-Exception.req		T-Unitdata.req (*Alert (warning)*)	EXCHANGE
T-Unitdata.ind	*ClientHello* A record identical to the previous one is received	Resend last buffer with T-Unitdata.req	EXCHANGE
	ClientHello A record not identical to the previous one is received	SEC-Create.ind	CREATING
	Alert (critical or fatal)	SEC-Terminate.ind	NULL
	Alert (warning)	SEC-Exception.ind	EXCHANGE
	*Certficate** *ClientKeyExchange** *CertificateVerify** *ChangeCipherSpec* *Finished*	SEC-Exchange.cnf SEC-Commit.ind The pending state is made current after *ChangeCipherSpec* so that *Finished* is processed under the new negotiated state and sequence numbers are set to zero Create a new buffer with: *Finished* Send it out with T-Unitdata.req	OPENING

*) Whether these messages are present or not depends on the chosen key exchange method.

Server Secure Session COMMIT

Event	Conditions	ACTION	Next State
SEC-Terminate.req		T-Unitdata.req (*Alert (critical or fatal)*)	NULL
SEC-Exception.req		T-Unitdata.req (*Alert (warning)*)	COMMIT
T-Unitdata.ind	*ClientHello* A record identical to the previous one is received	Resend last buffer with T-Unitdata.req	COMMIT
	ClientHello A record not identical to the previous one is received	SEC-Create.ind	CREATING
	Alert (critical or fatal)	SEC-Terminate.ind	NULL
	Alert (warning)	SEC-Exception.ind	COMMIT
	Finished	SEC-Commit.cnf	OPEN
	Finished and user data	SEC-Commit.cnf SEC-Unitdata.ind	OPEN

Server Secure Session OPENING

Event	Conditions	ACTION	Next State
SEC-Create-Request.req		T-Unitdata (*HelloRequest*)	OPENING
SEC-Terminate.req		T-Unitdata (*Alert (critical or fatal)*)	NULL
SEC-Exception.req		T-Unitdata.req (*Alert (warning)*)	OPENING
SEC-Unitdata.req		T-Unitdata.req	OPENING
T-Unitdata.ind	*ClientHello*	SEC-Create.ind	CREATING
	Alert (critical or fatal)	SEC-Terminate.ind	NULL
	Alert (warning)	SEC-Exception.ind	OPENING
	User data received	SEC-Unitdata.ind	OPEN
	*Certficate** *ClientKeyExchange** *CertificateVerify** *ChangeCipherSpec* *Finished* A group of records identical to the previous one is received	Resend last buffer with T-Unitdata.req	OPENING

Server Secure Session OPEN			
Event	**Conditions**	**ACTION**	**Next State**
SEC-Create-Request.req		T-Unitdata (*HelloRequest*)	OPEN
SEC-Terminate.req		T-Unitdata (*Alert (critical or fatal)*)	NULL
SEC-Exception.req		T-Unitdata.req (*Alert (warning)*)	OPEN
SEC-Unitdata.req		T-Unitdata.req	OPEN
T-Unitdata.ind	*ClientHello*	SEC-Create.ind	CREATING
	Alert (critical or fatal)	SEC-Terminate.ind	NULL
	Alert (warning)	SEC-Exception.ind	OPEN
	User data received	SEC-Unitdata.ind	OPEN
	Finished A record identical to the previous *Finished* is received	T-Unitdata (*Alert (duplicate_finished_received)*)	OPEN
	Finished and user data A record identical to the previous *Finished* is received	SEC-Unitdata.ind T-Unitdata (*Alert (duplicate_finished_received)*)	OPEN

8 Presentation Language

This document deals with the formatting of data in an external representation similar to TLS. The following very basic and somewhat casually defined presentation language 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 WTLS only, not to have general application beyond that particular point.

8.1 Basic Block Size

The representation of all data items is explicitly specified. The basic block size is one byte (ie 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.

8.2 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.

8.3 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. Then 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 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, sufficient to represent the value 400 (see Section 8.4). 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 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 */
```

The notation

```
A = B[first..last];
```

indicates that vector A is assigned to be the elements from first to last of B.

8.4 Numbers

The basic numeric data type is unsigned byte (uint8). All larger numeric data types are formed from fixed length series of bytes concatenated as described in Section 8.1 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” or “big-endian” order; the uint32 represented by the hex bytes 01 02 03 04 is equivalent to the decimal value 16909060.

8.5 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;
```

Enumerateds occupy 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;
```

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;
```

8.6 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 using a syntax much like that available for enumerateds. For example, T.f2 refers to the second filed of the previous declaration. Structure definitions may be embedded.

8.6.1 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, or a default arm for those elements missing. 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;
  Td fd;
  select (E) {
    case e1: Te1;
    case e2: Te2;
    …
    case en: Ten;
    default: TeDefault;
  } [[fv]];
} [[Tv]];
```

For example:

```
enum { apple, orange } 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: V2;       /* VariantBody, tag = orange */
  } variant_body;
} VariantRecord;
```

Variant structures may be qualified (narrowed) by specifying a value for the selector prior to the type. For example, a

```
orange VariantRecord
```

is a narrowed type of VariantRecord containing a variant_body of type V2.

8.7 Cryptographic Attributes

The four cryptographic operations digital signing, stream cipher encryption, block cipher encryption, and public key encryption are designated digitally-signed, stream-ciphered, block-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 9.1).

In digital signing, one-way hash functions are used as input for a signing algorithm. A digitally-signed element is encoded as an opaque vector <0..2^{16}-1>, where the length is specified by the signing algorithm and key.

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 which are block-ciphered will be an exact multiple of the cipher block length.

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 signing algorithm and key.

In the following example:

```
block-ciphered struct {
  uint8 field1;
  uint8 field2;
  digitally-signed opaque hash[20];
} UserType;
```

The contents of hash are used input for the signing algorithm, then the entire structure is encrypted with a block cipher. The length of this structure, in bytes would be exact multiple of the cipher block length.

8.8 Constants

Typed constants can be defined for purpose 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 */
```

8.9 String Constants

A string constant must be interpreted as a vector of bytes (uint8) with a fixed length. Strings are enclosed with quotation marks. Unlike in C, no terminating nulls are implied. ASCII coding must be used.

```
For example,

block = H(parameter, “key expansion”);
  /* string length is 13 bytes (no terminating null) */
```

9 Record Protocol Specification

The WTLS Record Protocol is a layered protocol. The Record Protocol takes messages to be transmitted, optionally compresses the data, applies a MAC, encrypts, and transmits the result. Received data is decrypted, verified, and decompressed, then delivered to higher level clients.

Four record protocol clients are described in this document: the change cipher spec protocol, the handshake protocol, the alert protocol, and the application data protocol. If a WTLS implementation receives a record type it does not understand, it should ignore it.

Several records can be concatenated into one transport SDU. For example, several handshake messages can be transmitted in one transport SDU. This is particularly useful with packet-oriented transports such as GSM short messages.

9.1 Connection State

A WTLS connection state is the operating environment of the WTLS Record Protocol. It specifies a compression algorithm, encryption algorithm and MAC algorithm. In addition, the parameters for these algorithms are known: the MAC secret and the bulk encryption keys and IVs for the secure connection in both the read and the write directions.

Logically, there are always two connection states outstanding: the current state and the pending state. All records are processed under the current state. The security parameters for the pending state are set by the WTLS Handshake Protocol. The Handshake Protocol must make the pending state current. The pending state is then reinitialised to an empty state. The initial current state always specifies that no encryption, compression, or MAC will be used.

The security parameters for a WTLS connection state are set by providing the following values. Note that the following values are agreed on in a handshake procedure between a client and server when a secure session is negotiated (for more information see Chapter 10):

These parameters are defined in the presentation language as:

```
enum { server(1), client(2) } ConnectionEnd;

uint8  BulkCipherAlgorithm;

enum { stream(1), block(2), (255) } CipherType;

enum { true, false } IsExportable;

uint8  MACAlgorithm;

enum { off(0), implicit(1), explicit(2), (255) } SequenceNumberMode;

uint8 CompressionMethod;
```

```
struct {
   ConnectionEnd       entity;
   BulkCipherAlgorithm bulk_cipher_algorithm;
   CipherType          cipher_type;
   uint8               key_size; /* bytes */


   uint8               iv_size; /* bytes */
   uint8               key_material_length; /* bytes */
   IsExportable        is_exportable;
   MACAlgorithm        mac_algorithm;
   uint8               mac_key_size; /* bytes */
   uint8               mac_size; /* bytes */
   opaque              master_secret[20];
   opaque              client_random[16];
   opaque              server_random[16];
   SequenceNumberMode  sequence_number_mode;
   uint8               key_refresh;
   CompressionMethod   compression_algorithm;
} SecurityParameters;
```

Item	**Description**
Connection End	Whether this entity is considered a client or a server in this secure session.
Bulk Cipher Algorithm	An algorithm to be used for bulk encryption. This specification includes the key size of this algorithm, how much of that key is secret, whether it is a block or stream cipher, the block size of the cipher (if appropriate), and whether it is concidered as an “export cipher”. Bulk cipher algorithms are listed in Appendix A.
MAC Algorithm	An algorithm to be used for message authentication. This specification includes the size of the key used for MAC calculation and the size of the hash which is returned by the MAC algorithm. MAC algorithms are listed in Appendix A.
Compression Algorithm	The algorithm to compress data prior to encryption. This specification must include all information the algorithm requires to do compression.
Master Secret	A 20 byte secret shared between the two peers in the secure connection.
Client Random	A 16 byte value provided by the client.
Server Random	A 16 byte value provided by the server.

Item	**Description**
Sequence Number Mode	Which scheme is used to communicate sequence numbers in this secure connection: Implicit sequence numbering Sequence numbers will be used as an input to MAC calculations. This requires that a reliable transport protocol is used. Explicit sequence numbering The sequence number will be sent in plaintext with record layer messages and it is used as an input to MAC calculations. This option MUST be used when operating on a datagram transport protocol. Note that in this case sequence numbers do not have to be in unbroken sequence, but they have to be sent in monotonic way (the sequence number of each sent record is greater than the previous one). If the verification fails, *bad_record_mac* alert is sent as normally. Off No sequence numbers will be used. This option is not recommended and choosing it makes the system vulnerable for playback attacks. In this case, protection against such attacks must be provided by upper protocol layers.
Key Refresh	Defines how often some connection state parameters (encryption key, MAC secret, and IV) are updated New keys are calculated at every $n = 2^{key\_refresh}$ messages, ie, when the sequence number is 0, n, 2n, 3n etc. For example, if three is chosen as a value for key_refresh, a new set of keys is generated for every eight (2^8) messages, ie, messages with sequence numbers 0, 8, 16 etc. If zero is chosen, a new key set is generated for each message (2^0).

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:

Item	Description
Compression State	The current state of the compression algorithm. Note that a stateful compression cannot be used when operating on top of a datagram protocol. If a stateful compression is used, there are separate states for both directions.
Client write MAC secret	The secret used for MAC calculation/verification for records sent by the client. The secret must be updated according to the key refresh parameter.
Client write encryption key	The key used for encryption/decryption of records sent by the client. The key must be updated according to the key refresh parameter.
Client write IV	The base IV used to calculate a record level IV for block ciphers running in CBC mode for records sent by the client.
Client write sequence number	The sequence number used for records sent by the client. Sequence numbers are of type uint16 and may not exceed 2^{16}-1. When a new connection state is established the sequence number of the first record is zero.
Server write MAC secret	The secret used for MAC calculation/verification for records sent by the server. The secret must be updated according to the key refresh parameter.
Server write encryption key	The key used for encryption/decryption of records sent by the server. The key must be updated according to the key refresh parameter.
Server write IV	The base IV used to calculate a record level IV for block ciphers running in CBC mode for records sent by the server.
Server write sequence number	The sequence number used for records sent by the server. Sequence numbers are of type uint16 and may not exceed 2^{16}-1. When a new connection state is established the sequence number first record is zero.

9.2 Record Layer

The WTLS Record Layer receives uninterpreted data from higher layers in non-empty blocks of size maximum of 2^{16}-1.

9.2.1 Fragmentation

Unlike in TLS, the record layer does not fragment information blocks. It is assumed that the transport layer takes care of the necessary fragmentation and reassembly.

```
enum {
   change_cipher_spec(1), alert(2), handshake(3),
   application_data(4), (15)
} ContentType;


enum { without(0), with(1) } SequenceNumberIndication;


enum { without(0), with(1) } FragmentLengthIndication;
```

```
struct {
   opaque record_type[1];
   select (SequenceNumberIndication) {
       case without: struct {};
       case with: uint16 sequence_number;
   }
   select (FragmentLengthIndication) {
       case without: struct {};
       case with: uint16 length;
   }   opaque fragment[WTLSPlaintext.length];
} WTLSPlaintext;
```

Description of WTLSPlaintext fields:

Item	**Description**		
record_type	Defines the higher level protocol used to process the enclosed fragment. Contains also information about the existence of optional fields in the record and an indication about ciphering state.		
	Bits	**Length**	**Description**
	0 – 3	4 bits	Content type
	4	1 bit	Reserved for future use
	5	1 bit	Cipher spec indicator defines whether this record is transmitted under a cipher spec different from null 0 = null cipher spec used 1 = current, different from null, cipher spec is used Null cipher spec means that no compression, MAC protection or encryption is used. Its usage is restricted to handshake messages starting a new session and certain alerts sent in cleartext (see Section 10.2.2).
	6	1 bit	Sequence number field indicator defines whether the next byte in this record contains a sequence number field: 0 = no sequence number field 1 = sequence number included The sequence number field MUST be used with datagram transports (see Section 9.2.3.1 for explicit sequence numbering).

Item	Description		
	7	1 bit	Record length field indicator defines whether the record contains a length field: 0 = no record length field 1 = record length field included In some circumstances, it is possible to avoid sending the record length in the record layer. This reduces the amount of overhead two bytes per record. The requirements for leaving the field out are: 1. The receiver must be able to determine the size of the transport SDU. 2. This is the last (or the only) record in this transport SDU. If both requirements are met, each peer can decide per message whether they use the record length field or not. If possible the record length field should be left out.
sequence_number	An optional sequence number of the record. Note that this field MUST be used with datagram transports (see Section 9.2.3.1 for explicit sequence numbering).		
length	The optional length (in bytes) of the following WTLSPlaintext.fragment. This field MUST be used if several records are concatenated into one transport SDU.		
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.		

Note: Data of different WTLS Record layer content types may be interleaved. Application data is generally of lower precedence for transmission than other content types.

9.2.2 Record Compression and Decompression

All records are compressed using the compression algorithm defined in the current connection state. There is always an active compression algorithm; however, initially it is defined as NULL. Note that a stateful compression algorithm can not be used if WTLS is ran on top of a datagram transport.

The compression algorithm translates a WTLSPlaintext structure into a WTLSCompressed structure. This means that the WTLSPlaintext.fragment is compressed and copied. Other fields (such as the fragment length) are updated if needed.

```
struct {
   opaque record_type[1];
   select (SequenceNumberIndication) {
       case without: struct {};
       case with: uint16 sequence_number;
   }
   select (FragmentLengthIndication) {
       case without: struct {};
       case with: uint16 length;
   }
   opaque fragment[WTLSCompressed.length];
} WTLSCompressed;
```

Description of WTLSCompressed fields:

Item	**Description**
record_type	As in Section 9.2.1.
sequence_number	As in Section 9.2.1.
length	The optional length (in bytes) of the following WTLSCompressed.fragment (See Section 9.2.1).
fragment	The compressed form of WTLSPlaintext.fragment.

9.2.3 Record Payload Protection

The encryption and MAC functions translate a WTLSCompressed structure into a WTLSCiphertext. The decryption functions reverse the process.

```
struct {
   opaque record_type[1];
   select (SequenceNumberIndication) {
       case without: struct {};
       case with: uint16 sequence_number;
   }
   select (FragmentLengthIndication) {
       case without: struct {};
       case with: uint16 length;
   }
   select (SecurityParameters.cipher_type) {
       case stream: GenericStreamCipher;
       case block: GenericBlockCipher;
   } fragment;
} WTLSCiphertext;
```

Item	**Description**
record_type	As in Section 9.2.1.
sequence_number	As in Section 9.2.1.
length	The optional length (in bytes) of the following WTLSCiphertext.fragment (See Section 9.2.1).
fragment	The encrypted form of WTLSCompressed.fragment.

9.2.3.1 Explicit Sequence Numbering

When explicit sequence numbering is used, record verification and decryption require special measures. Explicit sequence numbering MUST be used with a datagram transport protocols meaning that records can be lost, duplicated, or received out of order.

The receiver MUST keep books about received records in order to discard duplicated records. This can be implemented using a sliding window. For example, a window size of 32 can be used. Using this window, the receiver can keep books on received messages with sequence numbers in the range n – 32 … n

where n is the current (expected) sequence number. Records with sequence numbers n – 32 MUST be discarded

When a handshake starts with plain text message exchanges, sequence numbers start from zero and are incremented by one in each handshake message. When a handshake starts on a secure connection, the current sequence numbers for the secure connection is used for handshake messages and are incremented by one for each handshake message. They are set to zero after ChangeCipherSpec message for either cases. In retransmissions, sequence numbers remain the same as in the original messages. When the sequence number exceeds 2^{16}-1 the secure connection MUST be closed.

In handshake messages, sequence numbers MUST be used (even on connection oriented transports). After negotiation, sequence numbers are either used or not. Note that with datagram transport protocols, sequence numbers MUST always be used.

9.2.3.2 Null or Standard Stream Cipher

Stream ciphers (including BulkCipherAlgorithm NULL) convert WTLSCompressed.fragment structures to and from stream WTLSCiphertext.fragment structures.

```
stream-ciphered struct {
   opaque content[WTLSCompressed.length];
   opaque MAC[SecurityParameters.mac_size];
} GenericStreamCipher;
```

The MAC is generated as:

```
HMAC_hash ( MAC_secret, seq_number + WTLSCompressed.record_type +
            WTLSCompressed.length + WTLSCompressed.fragment )
```

where “+” denotes concatenation. If WTLSCompressed.length is not available, the actual length of the compressed fragment should be used instead. If sequence numbers are not used at all, a value of zero is assumed.

Note that no stream ciphers except BulkCipherAlgorithm NULL are defined in the current WTLS specification.

9.2.3.3 CBC Block Cipher

For block ciphers (such as RC5 and DES), the encryption and MAC functions convert WTLSCompressed.fragment structures to and from block WTLSCiphertext.fragment structures.

```
block-ciphered struct {
   opaque content[WTLSCompressed.length];
   opaque MAC[SecurityParameters.hash_size];
   uint8 padding[padding_length];
   uint8 padding_length;
} GenericBlockCipher;
```

The MAC is generated as described in Section 9.2.3.2.

Item	**Description**
padding	Padding that is added to force the length of the plaintext to be a multiple of the block cipher’s block length. Each uint8 in the padding data vector MUST be filled with the padding length value.
padding_length	The padding length should 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.

The encrypted data length (WTLSCiphertext.length) is one more than the sum of WTLSCompressed.length, CipherSpec.hash_size, and padding_length.

Example: If the block length is 8 bytes, the content length (WTLSCompressed.length) is 59 bytes, and 10 bytes of the MAC are used, the length before padding is 70 bytes. Since 70 mod 8 is 6, 2 bytes of padding are required.

Generation of the encryption key and the initialization vector (IV) is explained in the Section 11.2.

10 Handshake Protocol Specification

The WTLS Handshake Protocol is composed of three sub-protocols which are used to allow peers to agree upon security parameters for the record layer, authenticate themselves, instantiate negotiated security parameters, and report error conditions to each other.

The Handshake Protocol is responsible for negotiating a secure session, which consists of the following items:

Item	Description
Session Identifier	An arbitrary byte sequence chosen by the server to identify an active or resumable secure session.
Protocol Version	WTLS protocol version number.
Peer Certificate	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 bulk data encryption algorithm (such as null, RC5, DES, etc.) and a MAC algorithm (such as SHA-1). It also defines cryptographic attributes such as the mac_size.
Master Secret	20-byte secret shared between the client and server.
Sequence Number Mode	Which sequence numbering scheme (off, implicit, or explicit) is used in this secure connection.
Key Refresh	Defines how often some connection state values (encryption key, MAC secret, and IV) calculations are performed.
Is Resumable	A flag indicating whether the secure session can be used to initiate new secure connections.

These items are then used to create security parameters for use by the Record Layer when protecting application data. Many secure connections can be instantiated using the same secure session through the resumption feature of the WTLS Handshake Protocol.

10.1 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 change cipher spec is sent either by the client or server to notify the other party that subsequent records will be protected under the newly negotiated CipherSpec and keys. In practise, it means that both peers should immediately make the pending state current. The change cipher spec message is sent during the handshake after the security parameters have been agreed upon, but before the verifying finished message is sent. Implementations MUST check that the change cipher spec message is sent or received before sending or receiving the verifying finished message, so that the finished and subsequent messages are protected under the newly negotiated Cipher Spec and keys.

10.2 Alert Protocol

One of the content types supported by the WTLS Record layer is the alert type. Alert messages convey the severity of the message and a description of the alert.

Alert messages with a level of fatal result in the immediate termination of the secure connection. In this case, other connections using the secure session MAY continue, but the session identifier MUST be invalidated, preventing the failed secure session from being used to establish new secure connections.

Alert messages with a level of critical result in the immediate termination of the secure connection. Other connections using the secure session MAY continue and the session identifier MAY be preserved to be used for establishing new secure connections.

An alert message is either sent as specified by the current connection state (ie, compressed and encrypted), or under null cipher spec (ie, without compression or encryption).

A 4-byte checksum is used in alerts. The checksum is calculated from the last record (ie, WTLSCiphertext structure) received from the other party, in the following way:

1. Pad the record with zero bytes so that its length is modulo 4

2. Devide the result into 4-byte blocks

3. XOR these blocks together

The receiver of the alert SHOULD verify that the checksum matches with the message earlier sent by him.

```
enum { warning(1), critical(2), fatal(3), (255) } AlertLevel;
enum {
   connection_close_notify(0),
   session_close_notify(1)
   no_connection(5),
   unexpected_message(10),
   bad_record_mac(20),
   decryption_failed(21),
   record_overflow(22),
   decompression_failure(30),
   handshake_failure(40),
   bad_certificate(42),
   unsupported_certificate(43),
   certificate_revoked(44),
   certificate_expired(45),
   certificate_unknown(46),
   illegal_parameter(47),
   unknown_ca(48),
   access_denied(49),
   decode_error(50),
   decrypt_error(51),
   unknown_key_id(52),
   disabled_key_id(53),
   key_exchange_disabled(54),
   session_not_ready(55),
   unknown_parameter_index(56),
   duplicate_finished_received(57),
   export_restriction(60),
   protocol_version(70),
   insufficient_security(71),
   internal_error(80),
   user_canceled(90),
   no_renegotiation(100), (255)
} AlertDescription;
```

```
struct {
   AlertLevel level;
   AlertDescription description;
   opaque checksum[4]
} Alert;
```

10.2.1 Closure Alerts

The client and the server must share knowledge that the secure connection is ending. Either party may initiate the exchange of closing messages.

Alert	Description
connection_close_notify	This message notifies the recipient that the sender will not send any more messages using this connection state.
session_close_notify	This message notifies the recipient that the sender will not send any more messages using this connection state or the secure session.

Either party may initiate a close by sending a *connection_close_notify* or *session_close_notify* alert. Any data received after a closure alert is ignored. It is required that the other party responds with a *connection_close_notify* or *session_close_notify* alert of its own, respectively, and close down the secure connection immediately, discarding any pending writes. It the case of a *session_close_notify*, the receiver MUST also invalidate the session identifier. It is not required for the initiator of the close to wait for the responding *connection_close_notify* or *session_close_notify* alert before closing the read side of the secure connection.

10.2.2 Error Alerts

Error handling in the WTLS 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 secure connection. Servers and clients are required to forget any session identifiers, keys, and secrets associated with a failed secure connection. Upon transmission or receipt of a critical alert message, both parties immediately close the secure connection but MAY preserve the session-identifiers and use that for establishing new secure connections. The following error alerts are defined:

Alert	**Description**
no_connection	A message was received while there is no secure connection with the sender. This message is fatal or critical.The message is sent in cleartext.
unexpected_message	An inappropriate message was received. This alert SHOULD be fatal or critical.
bad_record_mac	This alert is returned if a record is received with an incorrect MAC. This message is generally a warning. The message is sent in cleartext.
decryption_failed	A WTLSCiphertext decrypted in an invalid way: either it wasn't a multiple of the block length or its padding values, when checked, weren't correct. This message is generally a warning. The message is sent in cleartext.
record_overflow	A WTLSCiphertext record was received which had a length more than allowed bytes, or a record decrypted to a WTLSCompressed record with more than allowed bytes. This message is generally a warning. The message is sent in cleartext.
decompression_failure	The decompression function received improper input (eg, data that would expand to excessive length). This message is generally a warning. The message is sent in cleartext.
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.
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. Note that certificate revocation is likely to be checked by servers only.
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 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

Alert	**Description**
	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 fatal or critical.
decrypt_error	A handshake cryptographic operation failed, including being unable to correctly verify a signature, decrypt a key exchange, or validate a finished message. This message SHOULD be sent as fatal.
unknown_key_id	None of the client key_id's listed in ClientHello.client_key_ids is known or recognized to the server, or the client did not supply any items, if the server has the policy that requires recognition of client_key_id's. This is generally a fatal alert.
disabled_key_id	All the client_key_id's listed in ClientHello.client_key_ids are disabled administratively. This is generally a critical alert.
key_exchange_disabled	To protect the outcome of the anonymous key exchange from being overriding by the undesirable subsequent anonymous key exchanges, key exchange is administratively disabled.
session_not_ready	The secure session is not ready to resume new secure connections due to administrative reasons such as that the session is temporarily not available due to maintenance in the server. This is generally a critical alert.
unknown_parameter_ index	The client has suggested a key exchange suite that could be supported by the server, but the server does not know the key exchange parameter index supplied. When receiving this alert, the client may initiate a new handshake and suggest another parameter index, supply the parameters explicitly or let the server supply the parameters.
duplicate_finished_ received	In an abbreviated or optimised handshake, the client has sent a second (resent) finished message. This message is generally a warning.
export_restriction	A negotiation not in compliance with export restrictions was detected. This message is always fatal.
protocol_version	The protocol version the client (or server) has attempted to negotiate is recognised, but not supported by the server (or client). (For example, old protocol versions might be avoided for security reasons). This message is always fatal.
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 makes it impossible to continue (such as a memory allocation failure). This message is fatal or critical.
user_canceled	This handshake is being cancelled for some reason unrelated to a protocol failure. If the user cancels an operation after the handshake is complete, just closing the secure connection by sending a *connection_close_notify* is more appropriate. This alert should be followed by a *connection_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;

Alert	Description
	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 secure connection.

For all errors where an alert level is not explicitly specified, the sending party may determine at its discretion whether this is a fatal or critical error or a warning; if an alert with a level of warning or critical is received, the receiving party may decide at its discretion whether to treat this as a fatal error or not. However, all messages which are transmitted with a level of fatal MUST be treated as fatal messages.

Implementations MAY maintain a count of received alerts with a level of warning or critical, and treat them as fatal when a certain configurable limit is exceeded.

A fatal alert only terminates the session to be created and leaves the existing session intact if the handshaking is conducted on an existing secure session. However, there may be some cases that closing the existing session is desirable. A *session_close_notify* MUST be sent to the peer if one of the parties decide to terminate the existing session immediately after a fatal alert is sent or received during a handshake that intends to create a new session. Under any other circumstances, a fatal alert is treated normally as described at the beginning of this section.

10.3 Handshake Protocol Overview

The cryptographic parameters of the secure session are produced by the WTLS Handshake Protocol, which operates on top of the WTLS Record Layer. When a WTLS 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 a shared secret.

The WTLS Handshake Protocol involves the following steps:

- Exchange hello messages to agree on algorithms, exchange random values.
- Exchange the necessary cryptographic parameters to allow the client and server to agree on a pre-master secret.
- Exchange certificates and cryptographic information to allow the client and server to authenticate themselves.
- Generate a master secret from the pre-master 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.

These goals are achieved by the handshake protocol, which can be summarised as follows: The client sends a client hello message to which the server must respond with a server hello message, or else a fatal error will occur and the secure connection will fail. The client hello and server hello are used to establish security enhancement capabilities between client and server. The client hello and server hello establish the following attributes: Protocol Version, Key Exchange Suite, Cipher Suite, Compression Method, Key Refresh, and Sequence Number Mode. Additionally, two random values are generated and exchanged: ClientHello.random and ServerHello.random.

The actual key exchange uses up to four messages: the server certificate, the server key exchange, the client certificate, and the client key exchange. New key exchange methods can be created by specifying a format for these messages and defining the use of the messages to allow the client and server to agree upon a shared secret. This secret should be quite long. For wireless environments, 20 bytes can be considered suitable.

Following the hello messages, the server will send its certificate, if it is to be authenticated. Additionally, a server key exchange message may be sent, if it is required (eg, the server does not have a certificate, or if its certificate is for signing only). The server may request a certificate from the client (or get the certificate from some certificate distribution service), if that is appropriate to the key exchange suite selected. Now the server will send the server hello done message, indicating that the hello-message phase of the handshake is complete. (The previous handshake messages are combined in one lower layer message.) The server will then wait for a client response. If the server has sent a certificate request message, the client must send the certificate message. The client key exchange message is now sent if the client certificate does not contain enough data for key exchange or if it is not sent at all. The content of that message will depend on the public key algorithm selected between the client hello and the server hello. If the client is to be authenticated using a certificate with a signing capability (eg, RSA), a digitally-signed certificate verify message is sent to explicitly verify the certificate.

At this point, a change cipher spec 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. From now on, the Cipher Spec indicator is set to 1 in the messages. When the server receives the change cipher spec message it also copies the pending Cipher Spec into the current Cipher Spec. In response, the server will send its own 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.)

```
Client                                         Server
____________________________________________________________________


ClientHello                --------->
                                               ServerHello
                                               Certificate*
                                               ServerKeyExchange*
                                               CertificateRequest*
                           <---------          ServerHelloDone
Certificate*
ClientKeyExchange*
CertificateVerify*
[ChangeCipherSpec]
Finished                   --------->
                           <---------          Finished

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

Figure 5. Message flow for a full handshake

* Indicates optional or situation-dependent messages that are not always sent.

When the client and server decide to resume a previous secure session instead of negotiating new security parameters the message flow is as follows:

The client sends a ClientHello using the Session ID of the secure session to be resumed. The server then checks its secure session cache for a match. If a match is found, and the server is willing to re-establish the secure connection under the specified secure session, it will send a ServerHello with the same Session ID value. At this point, the server must send a change cipher spec message and proceed directly to the finished message to which the client should response with its own finished message. 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.

Note that many simultaneous secure connections can be instantiated under one secure session. Each secure connection established from the same secure session shares some parameters with the others (eg, master secret).

```
Client                                          Server


ClientHello              -------->
                                                ServerHello
                                                [ChangeCipherSpec]
                         <--------              Finished
Finished
Application Data         -------->

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

Figure 6. Message flow for an abbreviated handshake

The shared-secret handshake means that the new secure session is based on a shared secret already implanted in both ends (eg, physically). In this case, the shared-secret KeyExchangeSuite is requested by the client. The message flow is similar to the abbreviated handshake in Figure 6.

Another variation is that the server, after receiving the ClientHello, can retrieve client's certificate using a certificate distribution service or from its own sources. In a Diffie-Hellman type key exchange method, assuming the Diffie-Hellman parameters are provided in the certificates, the server can calculate the pre-master secret and master secret at this point. In this case, the server sends its certificate, a Change Cipher Spec, and a Finished message.

```
Client                                          Server


ClientHello              -------->
                                                ServerHello
                                                Certificate
                                                [ChangeCipherSpec]
                         <--------              Finished
Finished
Application Data         -------->

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

Figure 7. Message flow for an optimised full handshake

10.4 Handshake Reliability over Datagrams

In the datagram environment, handshake messages may be lost, out of order, or duplicated. To make the handshake reliable over datagrams, WTLS requires that the handshake messages going in the same direction must be concatenated in a single transport Service Data Unit (SDU) for transmission, that the client retransmits the handshake messages if necessary, and that the server MUST appropriately respond the retransmitted messages from the client.

The handshake may consist of multiple messages to be delivered in one direction before any responses are required from the other end. Those messages must be concatenated into a single transport SDU for transmission or retransmission to guarantee that all the messages in the same SDU arrive in order. For instance, ServerHello, ChangeCipherSpec, and Finished messages can be sent in a single transport SDU for the abbreviated handshake. The maximum size of SDU for the underlying transport service layer must be sufficient to contain all those messages.

For the full handshake, the client must retransmit ClientHello and Finished messages if the expected response messages are not received from the server for a predefined time-out period. Note that the whole transport SDU which contains the

Finished message must be retransmitted. After the number of retransmissions exceeds the maximum predefined retransmission counter, the client terminates the handshake. Those predefined time-out and counter values may be obtained from the WTP stack through the management entity if the WTP stack is present above the WTLS stack.

For the optimized and abbreviated handshakes, like the full handshake, the client retransmits ClientHello, if necessary. In addition, the client must also prepend Finished message with the Application Data message until an Application Data message from the server is received and decrypted successfully or a *duplicated_finished_received* alert (warning) is received from the server. However, the first Finished message can be either sent alone or prepend with the Application Data message, if any.

For the full handshake, the server MUST retransmit the transport SDU which contains the ServerHello message upon receiving a duplicated ClientHello message. However, if the ClientHello is new, the server MUST start a new handshake and SEC-Create.ind service primitive MUST be generated. The server MUST also retransmit the transport SDU which contains the Finished message upon receiving a duplicated Finished message from the client.

For the optimized and abbreviated handshakes, the server behaves the same as that in the full handshake for handling the duplicated or new ClientHello messages. In addition, the server MUST ignore duplicated Finished message and keep the committed secure connection intact. If the server has no Application Data to send to the client, it SHOULD send *duplicated_finished_received* alert (warning).

10.5 Handshake Protocol

The WTLS Handshake Protocol is one of the defined higher level clients of the WTLS Record Protocol. This protocol is used to negotiate the secure attributes of a secure session. Handshake messages are supplied to the WTLS Record Layer, where they are encapsulated within one or more WTLSPlaintext structures, which are processed and transmitted as specified by the current active connection 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 */
   uint16 length;          /* bytes in message */
   select (msg_type) {
       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;
```

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 which is not bound by these ordering rules is the Hello Request 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.

10.5.1 Hello Messages

The hello phase messages are used to agree on used security parameters between the client and server. When a new secure session begins, the connection state (encryption, hash, and compression algorithms) is initialised to null. The Cipher Spec indicator is set to 0 in the records.

10.5.1.1 Hello Request

When this message will be sent:

> The hello request message may be sent by the server at any time.

Meaning of this message:

> Hello request is a simple notification that the client should begin the negotiation process anew by sending a client hello message when convenient. This message will be ignored by the client if the client is currently negotiating a secure session. This message MAY be ignored by the client if it does not wish to make a new handshake, 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 hello request but does not receive a client hello in response, it MAY close the secure connection with a fatal alert.

After sending a hello request, servers should not repeat the request until the subsequent handshake negotiation is complete. However, if the client does not respond in a reasonable time, the message MAY be sent again.

Structure of this message:

```
struct { } HelloRequest;
```

Note: This message must not be included in the message hashes which are maintained throughout the handshake and used in the finished messages and the certificate verify message.

10.5.1.2 Client Hello

When this message will be sent:

> When a client first connects to a server it is required to send the client hello as its first message. The client can also send a client hello in response to a hello request or on its own initiative in order to renegotiate the security parameters in an existing secure connection.

Structure of this message:

The key exchange list contains the cryptographic key exchange algorithms supported by the client in decreasing order of preference. In addition, each entry defines the certificate or public key the client wishes to use. The server will select one or, if no acceptable choices are presented, return a *handshake_failure* alert and close the secure connection. The trusted authorities list with a similar format identifies the trusted certificates known by the client.

```
struct {
   uint32 gmt_unix_time;
   opaque random_bytes[12];
} Random;
```

Item	Description
gmt_unix_time	The current time and date in standard UNIX 32-bit format (seconds since the midnight starting Jan 1, 1970, GMT) according to the sender's internal clock. Clocks are not required to be set correctly by the basic WTLS Protocol (so, if client has no date and time available it can place null here); higher level or application protocols may define additional requirements.
random_bytes	12 bytes generated by a secure random number generator. This value will be used later in the protocol.

```
uint8 KeyExchangeSuite;    /* Key exchange suite selector */

struct {
   uint8 dh_e;
   opaque dh_p<1..2^16-1>;
   opaque dh_g<1..2^16-1>;
} DHParameters;
```

Item	Description
dh_e	The exponent length in bytes. The value 0 indicates that the default length is used (ie, the same length as the prime).
dh_p	The prime modulus used for the Diffie-Hellman operation.
dh_g	The generator used for the Diffie-Hellman operation.

```
enum { ec_prime_p(1), ec_characteristic_two(2), (255) } ECFieldID;

enum { ec_basis_onb, ec_basis_trinomial, ec_basis_pentanomial } ECBasisType;

struct {
   opaque a <1..2^8-1>;
   opaque b <1..2^8-1>;
   opaque seed <0..2^8-1>;
} ECCurve;
```

Item	Description
a, b	These parameters specify the coefficients of the elliptic curve. Each value shall be the octet string representation of a field element following the conversion routine in [X9.62], section 4.3.1.

Item	Description
seed:	This is an optional parameter used to derive the coefficients of a randomly generated elliptic curve.

```
struct {
   opaque point <1..2^8-1>;
} ECPoint;
```

Item	Description
point	This is the octet string representation of an elliptic curve point following the conversion routine in [X9.62], section 4.4.2.a. The representation format is defined following the definition in [X9.62], section 4.4.

```
struct {
   ECFieldID   field;
   select (field) {
   case ec_prime_p: opaque prime_p <1..2^8-1>;
   case ec_characteristic_two:
       uint16      m;
       ECBasisType basis;
       select (basis) {
           case ec_basis_onb:
               struct { };
           case ec_trinomial:
               opaque  k <1..2^8-1>;
           case ec_pentanomial:
               opaque  k1 <1..2^8-1>;
               opaque  k2 <1..2^8-1>;
               opaque  k3 <1..2^8-1>;
       };
   };
   ECCurve     curve;
   ECPoint     base;
   opaque      order <1..2^8-1>;
   opaque      cofactor <1..2^8-1>;
} ECParameters;
```

Item	Description
field	This identifies the finite field over which the elliptic curve is defined.
prime_p	This is the odd prime defining the field F_p.
m	This is the degree of the characteristic-two field $F_{2 m}$
k	The exponent k for the trinomical basis representation $x^m + x^k + 1$.
k1, k2, k3	The exponents for the pentanomial representation $x^m + x^{k3} + x^{k2} + x^{k1} + 1$.
Curve	Specifies the coefficients a and b of the elliptic curve E.
base	The base point P on the elliptic curve.

Item	Description
order	The order n of the base point. The order of a point P is the smallest possible integer n such that nP = 0 (the point at infinity).
cofactor	The integer h = #E(Fq)/n, where #E(Fq) represents the number of points on the elliptic curve E defined over the field Fq.

```
uint8 ParameterIndex;

enum { rsa, diffie_hellman, elliptic_curve } PublicKeyAlgorithm;

struct {
   select (PublicKeyAlgorithm) {
       case rsa: struct {};
       case diffie_hellman: DHParameters params;
       case elliptic_curve: ECParameters params;
   }
} ParameterSet;

struct {
   ParameterIndex parameter_index;
   select (parameter_index) {
       case 255: ParameterSet parameter_set;
       default: struct {};
   }
} ParameterSpecifier;
```

Item	Description
parameter_index	Indicates parameters relevant for this key exchange suite 0 = not applicable, or specified elsewhere. 1…254 = assigned number of a parameter set, defined in Appendix A 255 = explicit parameters are present in the next field
parameter_set	Explicit parameters, eg, Diffie-Hellman or ECDH parameters. Implementations SHOULD use parameter indexes instead of explicit parameters.

```
enum { null(0), text(1), binary(2), key_hash_sha(254), x509_name(255)}
   IdentifierType;

uint16 CharacterSet;
```

```
struct {
   IdentifierType identifier_type;
   select (identifier_type) {
       case null: struct {};
       case text:
           CharacterSet character_set;
           opaque name<1.. 2^8-1>;
       case binary: opaque identifier<1..2^8-1>;
       case key_hash_sha: opaque key_hash[20];
       case x509_name: opaque distinguished_name<1..2^8-1>;
} Identifier;
```

Item	**Description**
identifier_type	Type of identifier used 0 = no identity supplied 1 = textual name with character set 2 = binary identity 254 = SHA-1 hash of the public key 255 = X.509 distinguished name
character_set	Maps to IANA defined character set.
name	Textual name.
identifier	Binary identifier.
key_hash	Hash calculated over the public key of the key pair which the client intends to use in the handshake to prove its identity.
distinguished_name	X.509 distinguished name.

```
struct {
   KeyExchangeSuite key_exchange_suite;
   ParameterSpecifier parameter_specifier;
   Identifier identifier;
} KeyExchangeId;
```

Item	**Description**
key_exchange_suite	Assigned number of the key exchange suite, defined in Appendix A.
parameter_specifier	Specifies parameters relevant for this key exchange suite. Value zero of a parameter index for a key exchange suite using parameters, indicates that the server MUST supply parameters
identifier	Identifies the client in a relevant way for the key exchange suite. The server can use this information to fetch a client certificate from a database.

The CipherSuite list, passed from the client to the server in the client hello message, contains the combinations of symmetric cryptographic algorithms supported by the client in order of the client’s preference (favourite choice first). Each

CipherSuite defines a bulk encryption algorithm (including secret key length) and a MAC algorithm. The server will select a cipher suite or, if no acceptable choices are presented, return a *handshake_failure* alert and close the secure connection.

```
struct {
   BulkCipherAlgorithm bulk_cipher_algorithm;
   MACAlgorithm        mac_algorithm;
} CipherSuite
```

Item	**Description**
bulk_cipher_algorithm	Assigned number of the bulk cipher algorithm, defined in Appendix A.
mac_algorithm	Assigned number of the MAC algorithm, defined in Appendix A.

```
opaque SessionID<0..8>;
```

The client hello includes a list of compression algorithms supported by the client, ordered according to the client's preference.

```
uint8 CompressionMethod;


struct {
   uint8 client_version;
   Random random;
   SessionID session_id;
   KeyExchangeId client_key_ids<3..2^16-1>;
   KeyExchangeId trusted_key_ids<0..2^16-1>;
   CipherSuite cipher_suites<2..2^8-1>;
   CompressionMethod compression_methods<1..2^8-1>;
   SequenceNumberMode sequence_number_mode;
   uint8 key_refresh;
} ClientHello;
```

Item	**Description**
client_version	The version of the WTLS protocol by which the client wishes to communicate during this secure session. This should be the latest (highest valued) version supported by the client. For this version of the specification, the version will be 1.
random	A client-generated random structure.
session_id	The ID of a secure session the client wishes to use for this secure connection. This field should be empty if no session_id is available or the client wishes to generate new security parameters.
client_key_ids	A list of cryptographic key exchange options and identities supported by the client, with the client's first preference first.
trusted_key_ids	A list of trusted certificates known by the client, with the client's first preference first.

Item	Description
cipher_suites	This is a list of the cryptographic options supported by the client, with the client's first preference first.
compression_methods	This is a list of the compression methods supported by the client, sorted by client preference. 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.
sequence_number_mode	This value indicates how sequence numbering should be used in record layer messages.
key_refresh	Defines how often some connection state parameters (encryption key, MAC secret, and IV) are updated. See Section 9.1.

After sending the client hello message, the client waits for a server hello message. Any other handshake message returned by the server except for a hello request is treated as a critical or fatal error.

When the client has an existing session_id and is initiating an abbreviated handshake, it MAY omit key exchange related items (client_key_ids, trusted_key_ids) from the client hello message. In this case, if the server is not willing to resume the session and is not able to continue with a full handshake, and it MUST to return an *unknown_key_id* alert.

10.5.1.3 Server Hello

When this message will be sent:

> The server will send this message in response to a client hello message when it was able to find an acceptable set of algorithms. If it cannot find such a match, it must respond with a *handshake_failure* alert.

Structure of this message:

```
struct {
   uint8 server_version;
   Random random;
   SessionID session_id;
   uint8 client_key_id;
   CipherSuite cipher_suite;
   CompressionMethod compression_method;
   SequenceNumberMode sequence_number_mode;
   uint8 key_refresh;
} ServerHello;
```

Item	Description
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 1.
random	This structure is generated by the server and must be different from (and independent of) ClientHello.random.
session_id	This is the identity of the secure session corresponding to this secure connection. If the ClientHello.session_id was non-empty, the server will look in its secure session cache for

Item	Description
	a match. If a match is found and the server is willing to establish the new secure connection using the specified secure session, the server will respond with the same value as was supplied by the client. This indicates a resumed secure session and dictates that the parties must proceed directly to the finished messages. Otherwise this field will contain a different value identifying the new secure session. The server MAY return an empty session_id to indicate that the secure session will not be cached and therefore cannot be resumed. If a secure session is resumed, it must be using the same cipher suite it was originally negotiated with.
client_key_id	The number of the key exchange suite selected by the server from the list in ClientHello.client_key_ids. For example, value one indicates that the first entry was selected.
cipher_suite	The single cipher suite selected by the server from the list in ClientHello.cipher_suites.
compression_method	The single compression algorithm selected by the server from the list in ClientHello.compression_methods.
sequence_number_mode	If the client suggested usage of sequence numbers then the server MUST confirm the value. If the client did not suggest usage the server can confirm that choice or indicate that sequence numbering should be used. So, if any party wishes to use sequence numbers then they have to be used.
key_refresh	This value indicates how many bits of the sequence number the server wishes to use to trigger key refresh. The value can be equal to what the client suggested or less . So, lower choice is used resulting in more frequent key refresh and thus higher security.

10.5.2 Server Certificate

When this message will be sent:

> If sent this message must always immediately follow the server hello message.

Meaning of this message:

> The certificate type must be appropriate for the selected key exchange suite’s algorithm. It can a X.509v3 certificate [X509] or a WTLS certificate which is optimised for size. Other certificate types may be added in the future. It must contain a key which matches the key exchange method, as follows. Unless otherwise specified, the signing algorithm for the certificate must be the same as the algorithm for the key carried in the certificate. Unless otherwise specified, the public key may be of any length.

As KeyExchangeSuites which specify new key exchange methods are specified for the WTLS Protocol, they will imply certificate format and the required encoded keying information.

Structure of this message:

```
enum {  WTLSCert(1), X509Cert(2), (255) } CertificateFormat;

opaque ASN1Cert<1..2^16-1>;

enum { anonymous(0), ecdsa_sha(1), rsa_sha(2), (255)} SignatureAlgorithm;
enum { rsa(2), ecdh(3), ecdsa(4), (255) } PublicKeyType;


ECPoint ECPublicKey;
```

Item	Description
ECPublicKey	The EC public key W = sG [P1363].

```
struct {
   opaque rsa_exponent<1..2^16-1>;
   opaque rsa_modulus<1..2^16-1>;
} RSAPublicKey;
```

Item	Description
rsa_exponent	The exponent of the server's RSA key.
rsa_modulus	The modulus of the server's RSA key.

```
struct {
   select (PublicKeyType) {
       case ecdh: ECPublicKey;
       case ecdsa: ECPublicKey;
       case rsa:  RSAPublicKey;
} PublicKey;


struct {
   uint8 certificate_version;
   SignatureAlgorithm signature_algorithm;
   Identifier issuer;
   uint32 valid_not_before;
   uint32 valid_not_after;
   Identifier subject;
   PublicKeyType public_key_type;
   ParameterSpecifier parameter_specifier;
   PublicKey public_key;
} ToBeSignedCertificate;
```

Item	Description
certificate_version	Version of the certificate. For this specification, the version is 1.
signature_algorithm	Algorithm used to sign the certificate.
issuer	Issuer of the certificate. Defines who signed the certificate. Certificates are usually signed by Certification Authorities (CA)
valid_not_before	Beginning of the validity period of the certificate, expressed in standard UNIX 32-bit format (seconds since the midnight starting Jan 1, 1970, GMT)
valid_not_after	End of the validity period of the certificate, expressed in standard UNIX 32-bit format (seconds since the midnight starting Jan 1, 1970, GMT)
subject	Owner of the key, associated with the public key being certified.

Item	Description
public_key_type	Type (algorithm) of the public key.
parameter_specifier	Specifies parameter relevant for the public key.
public_key	Public key that is being certified.

The hash value and the signature is calculated from ToBeSignedCertificate using the algorithms defined in CertificateSignatureAlgorithm.

```
select( SignatureAlgorithm )
{
   case anonymous: { };
   case ecdsa_sha:
       digitally-signed struct {
           opaque sha_hash[20]; /* SHA-1 hash of data to be signed */
       }
   case rsa_sha:
       digitally-signed struct {
           opaque sha_hash[20]; /* SHA-1 hash of data to be signed */
       }
} Signature;
```

```
struct {
   ToBeSignedCertificate to_be_signed_certificate;
   Signature signature;
} WTLSCertificate;
```

```
struct {
   CertificateFormat certificate_format;
   select (certificate_format)     {
       case X.509:  ASN1Cert;
       case WTLSCert:   WTLSCertificate;
   }
} Certificate;
```

```
struct {
   Certificate certificate_list<0..2^16-1>;
} Certificates;
```

Item	Description
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 must be distributed independently, the self-signed certificate which specifies the root certificate authority is 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.

To optimise the traffic and client processing, the chain should have minimal length. For server certificates, it is possible to have only one certificate: the server certificate certified by a CA public key of which is distributed independently.

Client certificate chain is likely to contain several certificates. However, this is acceptable because this chain is processed by the server. Also, server may get the client certificate from a certificate distribution service.

In a certificate chain, all certificates must use algorithms appropriate for the selected key exchange suite. Eg,

- for RSA, all certificates carry RSA keys signed with RSA
- for ECDH_ECDSA, the first certificate contains an ECDH key signed with ECDSA, and the following certificates carry ECDSA keys signed with ECDSA

10.5.3 Server Key Exchange Message

When this message will be sent:

> This message will be sent immediately after the server certificate message (or the server hello message, if this is an anonymous negotiation).

The server key exchange 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 pre-master secret. This is true for the following key exchange methods:

- ECDH_anon
- RSA_anon
- DH_anon

The server key exchange message MUST NOT be sent for the following key exchange methods:

- ECDH_ECDSA (fixed parameters)
- RSA

Meaning of this message:

> This message conveys cryptographic information to allow the client to communicate the pre-master secret: either an RSA public key to encrypt a secret with, or EC Diffie-Hellman parameters with which the client can complete a key exchange (with the result being the pre-master secret). As additional Key Exchange Suites are defined for WTLS which 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 pre-master secret.

Structure of this message:

```
enum { rsa, rsa_anon, dh_anon, ecdh_anon } KeyExchangeAlgorithm;
```

```
struct {
   opaque dh_Y<1..2^16-1>;
} DHPublicKey;
```

Item	Description
dh_Y	The Diffie-Hellman public value (Y).

```
struct {
   ParameterSpecifier parameter_specifier;
   select (KeyExchangeAlgorithm) {
       case rsa_anon:
           RSAPublicKey params;
       case diffie_hellman_anon:
           DHPublicKey params;
       case ec_diffie_hellman_anon:
           ECPublicKey params;
   };
} ServerKeyExchange;
```

Item	Description
parameter_specifier	Specifies parameters relevant for this key exchange suite. Value zero of a parameter index for a key exchange suite using parameters, indicates that the server is willing to use those parameters indicated by the client. If the client has not indicated parameters then the server MUST indicate them.
params	The server's key exchange parameters (RSA, ECDH or DH public key).

10.5.4 Certificate Request

When this message will be sent:

> A server can optionally request a certificate from the client, if appropriate for the selected cipher suite. This message, if sent, will immediately follow the Server Certificate message and Server Key Exchange message (if sent).

Structure of this message:

```
struct {
   KeyExchangeId trusted_authorities<0..2^16-1>;
} CertificateRequest;
```

Item	Description
trusted_authorities	A list of the names and types of acceptable certificate authorities. These names may specify a desired id for a root CA or for a subordinate CA; thus, this message can be used both to describe known roots and a desired authorisation space. If no authorities are

Item	Description
	sent, client may send any certificate.

10.5.5 Server Hello Done

When this message will be sent:

> The server hello done message is sent by the server to indicate the end of the server hello 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 server hello done 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;
```

10.5.6 Client Certificate

When this message will be sent:

> This message the client can be sent after receiving a server hello done 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. If client authentication is required by the server for the handshake to continue, it MAY respond with a fatal *handshake_failure* alert. Client certificates are sent using the Certificate structure defined previously for server certificates.

10.5.7 Client Key Exchange Message

When this message will be sent:

> This message will immediately follow the client certificate message, if it is sent. Otherwise it will be the first message sent by the client after it receives the server hello done message.

Meaning of this message:

> With this message, the pre-master secret is set, either through direct transmission of the RSA-encrypted secret, or by the transmission of EC Diffie-Hellman public key which will allow each side to agree upon the same pre-master secret. When the key exchange method is ECDH, client certification has been requested, and the client was able to respond with a certificate that contained EC Diffie-Hellman parameters matched those specified by the server in its certificate, this message is omitted.

Structure of this message:

> The structure of the message depends on which key exchange method has been selected.

```
struct {
   select (KeyExchangeAlgorithm) {
       case rsa: RSAEncryptedSecret param;;
       case rsa_anon: RSAEncryptedSecret param;
       case dh_anon: DHPublicKey param; /* client public value*
       case ecdh_anon: ECPublicKey param; /* client public value */
   } exchange_keys;
} ClientKeyExchange;
```

10.5.7.1 RSA Encrypted Secret Message

Meaning of this message:

> If RSA is being used for key agreement and authentication, the client generates a 20 byte secret, encrypts it using the public key from the server's certificate and sends the result in an encrypted secret message.

Structure of this message:

```
struct {
   uint8 client_version;
   opaque random[19];
} Secret;
```

Item	Description
client_version	The latest (newest) version supported by the client. This is used to detect version roll-back attacks. Upon receiving the secret, the server should check that this value matches the value transmitted by the client in the client hello message.
random	19 securely-generated random bytes.

```
struct {
   public-key-encrypted Secret secret;
} EncryptedSecret;
```

Item	Description
secret	This random value is generated by the client. This value appended with the public key is used as the pre-master secret which is used to generate the master secret, as specified in Chapter 11.

10.5.7.2 Client EC Diffie-Hellman Public Value

Meaning of this message:

> This message conveys the client's EC Diffie-Hellman public key if it was not already included in the client's certificate. This structure is a variant of the client key exchange message, not a message in itself.

10.5.7.3 Client Diffie-Hellman Public Value

Meaning of this message:

This message conveys the client's Diffie-Hellman public key if it was not already included in the client's certificate. This structure is a variant of the client key exchange message, not a message in itself.

10.5.8 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 by the client following a client certificate that has signing capability (ie, RSA certificates). When sent, it will immediately follow the client key exchange message.

Structure of this message:

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

Item	Description
signature	The hash value to be signed is calculated as follows: H(handshake_messages); Here handshake_messages refers to all handshake messages sent or received starting at client hello up to but not including this message, in the order they were sent by the client or by the server, including the data visible at the handshake layer, ie, also the type and length fields of the handshake messages. This is the concatenation of all the Handshake structures as defined in Section 10.5 exchanged this far. The hash algorithm used is the one agreed during the handshake.

10.5.9 Finished

When this message will be sent:

A finished message is always sent at the end of the handshake to verify that the key exchange and authentication processes were successful. Both ends must change finished messages immediately after a change cipher spec message.

Meaning of this message:

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

Structure of this message:

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

Item	**Description**
verify_data	The value is calculated as follows: PRF(master_secret, finished_label, H(handshake_messages)) [0..11];
	finished_label For Finished messages sent by the client, the string "client finished". For Finished messages sent by the server, the string "server finished". handshake_messages All of the data from all handshake messages up to but not including this message, in the order they were sent by the client or by the server. 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 10.5 exchanged thus far.

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

The value handshake_messages includes all handshake messages starting at client hello up to, but not including, this finished message. The handshake_messages for the finished message sent by the client will be different from that for the finished message sent by the server, because the one which is sent second will include the prior one.

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

11 Cryptographic Computations

11.1 Computing the Master Secret

In order to begin message protection, the WTLS Record Protocol requires specification of a suite of algorithms, a master secret, and the client and server random values. The encryption and MAC algorithms are determined by the cipher_suite selected by the server and revealed in the server hello message. The key exchange and authentication algorithms are determined by the key_exchange_suite also revealed in the server hello. 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.

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..19];
```

The master secret is always exactly 20 bytes in length. The length of the pre-master secret will vary depending on key exchange method.

11.1.1 RSA Encryption Scheme

When RSA is used for server authentication and key exchange, a 20-byte secret value 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 secret value . The pre_master_secret is the secret value appended with the server's public key. Both parties then convert the pre_master_secret into the master_secret, as specified above.

In RSA signing, a 20-byte structure of SHA-1 [SHA] hash is signed (encrypted with the private key), using PKCS #1 [PKCS1] block type 1.

RSA public key encryption is performed using PKCS #1 block type 2.

11.1.2 Diffie-Hellman

The 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.

11.1.3 EC Diffie-Hellman

The EC 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.

Elliptic curve calculations are performed according to [P1363].

EC parameters may be transmitted explicitly or using an algorithm definition which specifies pre-defined parameters (see Appendix A).

EC points are represented according to [P1363] Elliptic Curve Point to Octet String Primitive (EC2OSP). Implementations SHOULD use point compression.

ECDSA signature and verification is performed according to [P1363] Elliptic Curve Signature Scheme with Appendix (ECSSA) using

- EMSA-hash with SHA-1, for calculating the hash of the data to be signed
- the Elliptic Curve Signature Primitive, DSA version (ECSP-DSA) for signature, and the Elliptic Curve Verification Primitive, DSA version (ECVP-DSA) for verification
- output format for ECSSA, for output of the signature as an octet string

ECDH calculation of the key Z is performed according to [P1363]

- using the Elliptic Curve Secret Value Derivation Primitive, Diffie-Hellman version (ECSVDP-DH), for generating a shared secret value z as a field element
- converting the shared secret value *z* to an octet string *Z* using Field Element to Octet String Conversion Primitive (FE2OSP)

11.1.4 Session resume

In a session resume, the master_secret is not recalculated. This means that a resumed session uses the same master_secret as the previous one.

Note that although the same master_secret is used, new ClientHello.random and ServerHello.random values are exchanged in the abbreviated handshake. These randoms are taken into account in key block generation (see Section 11.2) meaning that each secure connection starts up with different key material.

11.2 Key Calculation

A connection state (see Section 9.1) is the operating environment of the Record Protocol. An algorithm is required to generate the connection state (encryption keys, IVs, and MAC secrets) from the secure session parameters provided by the handshake protocol.

A new connection state is calculated in the following way:

The master secret is hashed into a sequence of secure bytes, which are assigned to the MAC secrets, encryption keys, and IVs. To generate the key material, compute

```
key_block = PRF (SecurityParameters.master_secret,
                 expansion_label, seq_num +
                 SecurityParameters.server_random +
                 SecurityParameters.client_random);
```

until the needed amount of output has been generated.

A new key block generation takes place at intervals of the sequence number, corresponding to key refresh frequency. The sequence number used in the calculation is the first one that mandates key refresh.

Different value of expansion_label is used for client write keys and server write keys. So, the key_block generated with `“client expansion”` as expansion_label, is partitioned as follows:

```
client_write_MAC_secret[SecurityParameters.hash_size]
client_write encryption_key[SecurityParameters.key_material]
client_write IV[SecurityParameters.IV_size]
```

The key_block generated with `“server expansion”` as expansion_label, is partitioned as follows:

```
server_write_MAC_secret[SecurityParameters.hash_size]
server_write encryption_key[SecurityParameters.key_material]
server_write IV[SecurityParameters.IV_size]
```

In WTLS many connection state parameters can be recalculated during a secure connection. This feature is called the key refresh. It is performed in order to minimise the need for new handshakes. In the key refresh, the values of MAC secret, encryption key, and IV will change due to the sequence number. The frequency of these updates depends on the key refresh parameter. For example, the key refresh may be performed for every four records.The seq_num parameters used in the above calculation is the sequence number of the record that triggers key refresh. If sequence numbering is not used at all, the sequence numbers are equal to zero in all calculations.

Exportable encryption algorithms (for which SecurityParameters.is_exportable is true) require additional processing as follows to derive their final write keys:

```
final_client_write_encryption_key =
   PRF(SecurityParameters.client_write_encryption_key, "client write key",
   SecurityParameters.client_random + SecurityParameters.server_random);


final_server_write_encryption_key =
   PRF(SecurityParameters.server_write_encryption_key, "server write key",
   SecurityParameters.client_random + SecurityParameters.server_random);
```

Exportable encryption algorithms derive their IVs solely from the random values from the hello messages:

```
iv_block = PRF("", "IV block", seq_num +
   SecurityParameters.client_random + SecurityParameters.server_random);
```

The iv_block is partitioned into two initialization vectors as thekey_block was above:

```
client_write_IV[SecurityParameters.IV_size]
server_write_IV[SecurityParameters.IV_size]
```

Note that the PRF is used without a secret in this case: this just means that the secret has a length of zero bytes and contributes nothing to the hashing in the PRF.

For CBC mode block ciphers, the IV (initialisation vector) for each record is calculated in the following way:

```
record_IV = IV XOR S
```

where IV is the original IV (client_write_IV or server_write_IV) and S is obtained by concatenating the 2-byte sequence number of the record needed number of times to obtain as many bytes as in IV. It is also possible that an encryption algorithm supports using a sequence number as input. Then the record sequence number is used as the algorithm sequence number.

11.3 HMAC and the Pseudorandom Function

A number of operations in the WTLS record and handshake layer require a keyed MAC; this is a secure digest of some data protected by a secret.

In addition, a construction is required to do expansion of secrets into blocks of data for the purposes of key generation or validation. This pseudo-random function (PRF) takes as input a secret, a seed, and an identifying label and produces an output of arbitrary length.

11.3.1 MAC Calculation

HMAC [HMAC] can be used with a variety of different hash algorithms. For example, SHA-1 [SHA] or MD5 [MD5] could be used. The cryptographic hash function is denoted by H. In addition, a secret key K is required. We assume H to be a cryptographic hash function where data is hashed by iterating a basic compression function on blocks of data. We denote by B the byte-length of such blocks (B=64 for all the above mentioned examples of hash functions), and by L the byte-length of hash outputs (L=16 for MD5, L=20 for SHA-1). The authentication key K can be of any length up to B, the block length of the hash function. Applications that use keys longer than B bytes will first hash the key using H and then use the resultant L byte string as the actual key to HMAC. In any case, the minimal recommended length for K is L bytes (as the hash output length).

We define two fixed and different strings ipad and opad as follows (the 'i' and 'o' are mnemonics for inner and outer):

```
ipad = the byte 0x36 repeated B times
opad = the byte 0x5C repeated B times.
```

To compute HMAC over the data we perform

```
H( K XOR opad + H( K XOR ipad + data ) )
```

where + indicates concatenation.

Namely,

1. Append zeros to the end of K to create a B byte string (eg, if K is of length 20 bytes and B=64, then K will be appended with 44 zero bytes 0x00).
2. XOR (bitwise exclusive-OR) the B byte string computed in step (1) with ipad.
3. Append the data to the B byte string resulting from step (2).
4. Apply H to the data generated in step (3).
5. XOR (bitwise exclusive-OR) the B byte string computed in step (1) with opad.
6. Append the H result from step (4) to the B byte string resulting from step (5).
7. Apply H to the data generated in step (6) and output the result.

11.3.2 Pseudo-random Function

In the TLS standard, two hash algorithms were used in order to make the PRF as secure as possible. In order to save resources, WTLS can be implemented using only one hash algorithm. Which hash algorithm is actually used, is agreed during the handshake as a part of the cipher spec.

First, we define a data expansion function, P_hash(secret, data) using a single hash function to expand a secret and seed into an arbitrary quantity of output:

```
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(0) = seed
A(i) = HMAC_hash( secret, A(i-1) )
```

P_hash can be iterated as many times as is necessary to produce the required quantity of data. For example, if P_SHA was being used to create 64 bytes of data, it would have to be iterated 5 times (through A(4)), creating 80 bytes of output data; the last 16 bytes of the final iteration would then be discarded, leaving 64 bytes of output data.

Then,

```
PRF( secret, label, seed ) = P_hash( secret, label + seed )
```

Appendix A Algorithm Definitions

Table 4. The Available Key Exchange Suites

Key Exhange Suite	Assigned Number	Description	Key Size Limit (bits)
NULL	0	No key exchange is done. A zero length pre-master secret is used. Master secret and Finished messages are used for error checking purposes only.	N/A
SHARED_SECRET	1	Symmetric-key based handshake. Parties share a secret key that is used as the pre-master key as such.	None
DH_anon	2	Diffie-Hellman key exchange without authentication. Parties send each other (temporary) DH public keys. Each party calculates the pre-master secret based on one's own private key and counterpart's public key .	None
DH_anon_512	3	As DH_anon, but with a limited length DH key.	512
DH_anon_768	4	As DH_anon, but with a limited length DH key.	768
RSA_anon	5	RSA key exchange without authentication. The server sends its RSA public key. The client generates a secret value, encrypts it with the server's public key and sends it to the server. The pre-master secret is the secret value appended with the server's public key.	None
RSA_anon_512	6	As RSA_anon, but with a limited length server public key.	512
RSA_anon_768	7	As RSA_anon, but with a limited length server public key.	768
RSA	8	RSA key exchange with RSA based certificates. The server sends a certificate that contains its RSA public key. The server certificate is signed with RSA by a third party trusted by the client. The client extracts server's public key from received certificate, generates a secret value , encrypts it with the server's public key and sends it to the server. The pre-master secret is the secret value appended with the server's public key. If the client is to be authenticated it signs some data (messages send during the handshake) with its RSA private key and sends its certificate and the signed data.	None
RSA_512	9	As RSA, but with a limited length of certified server public key.	512

Key Exhange Suite	**Assigned Number**	**Description**	**Key Size Limit (bits)**
RSA_768	10	As RSA, but with a limited length of certified server public key.	768
ECDH_anon	11	EC Diffie-Hellman key exchange without authentication. Parties send each other (temporary) ECDH public keys. Each party calculates the pre-master secret based on one's own private key and counterpart's public key .	None
ECDH_anon_113	12	As ECDH_anon, but with a limited length ECDH key.	113
ECDH_anon_131	13	As ECDH_anon, but with a limited length ECDH key.	131
ECDH_ECDSA	14	EC Diffie-Hellman key exchange with ECDSA based certificates. The server sends a certificate that contains its ECDH public key. The server certificate is signed with ECDSA by a third party trusted by the client. Depending whether the client is to be authenticated or not, it sends its certificate containing its ECDH public key signed with ECDSA by a third party trusted by the server, or just its (temporary) ECDH public key. Each party calculates the pre-master secret based on one's own private key and counterpart's public key received as such or contained in a certificate.	None

Note that regarding to some key exchange suites, export restrictions may apply.

Table 5. The Available Bulk Encryption Algorithms

Cipher	Assigned Number	Is Export-able	Type	Key Material (bytes)	Expanded Key Material (bytes)	Effective Key Bits (bits)	IV Size (bytes)	Block Size (bytes)
NULL	0	True	Stream	0	0	0	0	N/A
RC5_CBC_40	1	True	Block	5	16	40	8	8
RC5_CBC_56	2	True	Block	7	16	56	8	8
RC5_CBC	3	False	Block	16	16	128	8	8
DES_CBC_40	4	True	Block	5	8	40	8	8
DES_CBC	5	False	Block	8	8	56	8	8
3DES_CBC_EDE	6	False	Block	24	24	168	8	8
IDEA_CBC_40	7	True	Block	5	16	40	8	8
IDEA_CBC_56	8	True	Block	7	16	56	8	8
IDEA_CBC	9	False	Block	16	16	128	8	8

Field	Description
IsExportable	Encryption algorithms for which IsExportable is true have a limited effective key length in order to comply with certain export regulations. For them, an additional key expansion is performed and the initialization vector is derived in a special way (Chapter 11.2). This specification does not imply whether it is actually legal to export these algorithms (or illegal to export algorithms for which IsExportable is false) from one specific country to another.
Type	Indicates whether this is a stream cipher of a block cipher running in CBC mode.
Key Material	The number of bytes from the key_block that are used for generating the write keys.
Expanded Key Material	The number of bytes in the write keys.
Effective Key Bits	How much entropy material is in the key material being fed into the encryption routines.
IV Size	How much data needs to be generated for the initialization vector. Zero for stream ciphers; equal to the block size for block ciphers.
Block Size	The amount of data a block cipher enciphers in one chunk; a block cipher running in CBC mode can only encrypt a multiple of its block size.

RC5 [RC5] is a family of block cipher algorithms. RC5 implementations can be designated as RC5-w/r/b, where w is the word size in bits (and also the half of the block size), r is the number of rounds, and b is the length of the key in bytes. Using this notation, the cipher RC5_CBC is RC5-32/16/16. The cipher RC5_CBC_40 is implemented as an export cipher, using 5 bytes as key material and expanding that to 16 bytes, and then applying RC5-32/12/16. The cipher RC5_CBC_56 is implemented as an export cipher, using 7 bytes as key material and expanding that to 16 bytes, and then applying RC5-32/12/16.

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

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

Table 6. The Available Keyed MAC Algorithms

Hash Function	Assigned Number	Description	Key Size (bytes)	Hash Size (bytes)
SHA_0	0	No keyed MAC is calculated. Note than in other than keyed MAC operations (eg, PRF) the full-length SHA-1 is used.	0	0
SHA_40	1	The keyed MAC is calculated using SHA-1 but only the first 5 bytes of the output are used. Note that in other than keyed MAC operations (eg, PRF) the full-length SHA-1 is used.	20	5
SHA_80	2	The keyed MAC is calculated using SHA-1 but only the first half of the output (10 bytes) is used. Note that in other than keyed MAC operations (eg, PRF) the full-length SHA-1 is used.	20	10
SHA	3	The keyed MAC is calculated using SHA-1.	20	20
SHA_XOR_40	4	A 5-byte XOR checksum. The input data is first divided into the multiple blocks of 5 bytes. Then all blocks are XOR'ed one after another. If the last block is less than 5 bytes, it is padded with 0x00. SHA is much stronger than XOR for generating MAC's, although there were no significant attacks reported on XOR MAC's, which must be encrypted and is only used for CBC mode block ciphers. XOR is only intended for some devices with very limited CPU resources. Warning: With exportable grade of encryption (eg, RC5_40), XOR can not provide as strong message integrity protection as SHA can. It is recommended that the security consequence should be carefully evaluated before XOR MAC is adopted in those environments. In other than MAC operations for message integrity (eg, PRF) the full-length SHA-1 is used.	0	5
MD5_40	5	The keyed MAC is calculated using MD5 but only the first 5 bytes of the output are used. Note than in other than keyed MAC operations (eg, PRF) the full-length MD5 is used.	16	5
MD5_80	6	The keyed MAC is calculated using MD5 but only the first 10 bytes of the output are used. Note than in other than keyed MAC operations (eg, PRF) the full-length MD5 is used.	16	10

Hash Function	Assigned Number	Description	Key Size (bytes)	Hash Size (bytes)
MD5	7	The keyed MAC is calculated using MD5.	16	16

Field	Description
Key Size	The number of bytes used as the HMAC key.
Hash Size	The number of bytes used in the MAC.

Table 7. The Available Compression Algorithms

Compression Algorithm	**Assigned Number**	**Description**
NULL	0	No compression.

Table 8. Elliptic Curve Parameters For Selected Curves

Parameter	**Value**
Assigned number	1
Field size	113
Elliptic curve E	$y^2 + x y = x^3 + a x^2 + b$ over $F(2^{113})$
Curve parameter a	1
Curve parameter b	1
Generating point G	01667979A40BA497E5D5C270780617, 00F44B4AF1ECC2630E08785CEBCC15
The order of G	00FFFFFFFFFFFFFFFDBF91AF6DEA73
The cofactor k	2

Parameter	**Value**
Assigned number	2
Field size	131
Elliptic curve E	$y^2 + x y = x^3 + a x^2 + b$ over $F(2^{131})$
Curve parameter a	0
Curve parameter b	1
Generating point G	043A891E4FD64F01E60F8831C3D7E195B22FF19BEE, 04035AB7114A900F460549987F48C3B1F00B5A1D58
The order of G	02000000000000000004D4FDD5703A3F269
The cofactor k	4

Parameter	**Value**
Assigned number	3
Field size	163
Elliptic curve E	$y^2 + x y = x^3 + a x^2 + b$ over $F(2^{163})$
Curve parameter a	1
Curve parameter b	1
Generating point G	02FE13C0537BBC11ACAA07D793DE4E6D5E5C94EEE8, 0289070FB05D38FF58321F2E800536D538CCDAA3D9
The order of G	04000000000000000000020108A2E0CC0D99F8A5EF
The cofactor k	2

Table 9. Predefined Diffie-Hellman Parameters

Parameter	**Value**
Assigned number	1
Exponent bits	160
Prime modulus (512 bits)	FAF30C63D171E54A8131CD331D7C8D6C 8AED41B0354E1A29D8DAD03E2E67FF8E 00053A07FD28A1EE6AF199FD70330EA8 C4C602B86EDFBF47FD1D7BFB6456BD57
Generator (512 bits)	E7734EBBCF50893C760181B2AA2DB0AC F2D5B6E775EE88BAFC7AA5A6BB20A64E B9F54301141F90291B7B375135394504 81C9F9CB2BA3E67B4580E2153FD22B80

Parameter	**Value**
Assigned number	2
Exponent bits	160
Prime modulus (768 bits)	85DB5DB185090AED3BDB3BABFCB46669F9563E681EDB4359 9241FEF6AA9B5DF9EFE39C0CB7994A04F2BD8F57B5B22AF7 5E360526216420BCA08FCDF98FF6417DCFDD1C40E4FFB183 260E3B28EF0B31A3633788C988B1BC6734A81B31A28CD6FB
Generator (760 bits)	1B15C3C57263B0DD1A9D996768B88370ED458D7B0081A220 054EFDD23B9CD8298B719FD3B67CB093817332D033642D21 130F83D9CB2CC5ACDD36E6E6DDB2410AB30311CDBEE9222C CFE644443B0C7204F2D12F7A3719C8866A20A0E778EBBA

Appendix B Implementation Notes

The following implementation notes are provided to identify areas where implementation choices may impact the security, performance and effectiveness of the WTLS protocols. The implementation notes provide guidance to implementers of the protocols.

B.1 Negotiating Null Cipher Spec

Null cipher spec can be negotiated to be used in a session. The NULL key exchange suite may be used for that purpose, so that no key exchange actually takes place. The master secret is calcucated with a zero length pre-master secret. The message flow is like in the abbreviated handshake.

Implementations MUST be careful when accepting a null cipher spec since it offers no security.

B.2 Anonymous handshakes

Completely anonymous sessions can be established using RSA or Diffie-Hellman for key exchange. With anonymous RSA, the client generates a secret value and encrypts it with the server's uncertified public key extracted from the server key exchange message. The result is sent in a client key exchange message. Since eavesdroppers do not know the server's private key, it will be infeasible for them to decode the secret value. (The pre_master_secret is this value appended with server's public key.)

With Diffie-Hellman, the server's public value is contained in the server key exchange message and the client's is sent in the client key exchange message. Eavesdroppers who do not know the private values are not able to find the Diffie-Hellman result (ie, the pre_master_secret).

Warning: Completely anonymous handshakes (ie, where neither the client nor the server is authenticated) only provide protection against passive eavesdropping. The active eavesdroppers, or the active man-in-the-middle attackers may replace the finished messages with their own during the handshaking process for creating sessions. However, there are known methods that may effectively defeat those active attacks in environments where those attacks are a concern. For instance, server authentication, or using an independent tamper-proof channel to verify that the finished messages were not replaced by the attacker. When the handshaking process is complete and authenticated or verified, the established sessions should be secure and protected against both passive and active man-in-the-middle attacks or eavesdroppers.

B.3 Key refresh

The passive key refresh mechanism of WTLS makes it possible to update keys in a secure connection without handshaking.

Key refresh makes cryptoanalysis less attractive for an attacker because keys will be invalidated regularly and the material that can be gained is limited. This is particularly useful in environments, where export-restricted encryption is used and handshaking is expensive (ie, connections with long lifetimes are desirable).

The frequency of key refresh is agreed on during the handshake. This parameter defines how many messages are sent before key refresh is triggered. For example, key refresh may be triggered after each four messages.

In key refresh, a new key block is generated using the master secret as a source of entropy and the message sequence number as an additional parameter (along with other parameters) in the pseudorandom function. The generated key block is used for message protection keys: MAC keys, encryption keys and initialization vectors.

Note that key refresh can always be done also by performing/forcing a new handshake.

B.4 Denial-of-Service Attacks

Since WTLS operates on top of datagrams, the implementation should pay special attention to preventing denial-of-service attacks. It should take into account that in some networks transport addresses may be forged relatively easy.

In order to make denial-of-service attacks harder to accomplish, it may not be possible for an attacker to break up an existing connection/session by sending a single message in plaintext from a forged address.

In addition, the server should be careful in accepting new connection requests in plain text within an existing secure connection. Note that the server cannot just ignore them because eg, ClientHello in plain text may be sent by a client whose connection state was lost. Special care must be taken with arbitrated and optimized handshakes in which the server switches the pending state current immediately after responding to ClientHello message. In such a case, the old active state should be kept intact until the new handshake is accomplished. In other words, the server should not discard the old active state until the client responds with Finished and the handshake is completed successfully. The old active state should be restored to the current state if it is evidenced that the handshake started is invalid.

For the same reason, when a client receives a plaintext ServerHello on its secure connection, it should not cause the existing secure connection broken because of the unexpected message. It should keep the existing secure connection and send the *unexpected_message* as a warning.

Appendix C Implementation Classes

WTLS implementations may have support for various features. This appendix defines classes guiding implementors to select these features. A class may have mandatory (M) or optional (O) support for a certain feature. Certain features are not yet defined in the current version of the specification.

The current version of the WTLS specification covers all features in class 1.

Table 10. WTLS Classes

Feature	Class 1	Class 2	Class 3
Public-key exchange	M	M	M
Server certificates	O	M	M
Client certificates	O	O	M
Shared-secret hanshake	O	O	O
Compression	-	O	O
Encryption	M	M	M
MAC	M	M	M
Smart card interface	-	O	O

Appendix D Requirements for the WTLS Protocol

The common requirements set by wireless mobile networks are described below.

Item	Description
Datagram transport protocol	Both datagram and connection oriented transport layer protocols must be supported. It must be possible to cope with, for example, lost, duplicated, or out of order datagrams without breaking the connection state.
Slow interactions	The protocol must take into account that round-trip times with some bearers (eg, SMS [GSM03.40]) can be long. For example, sending a query and receiving a response might require more than 10 seconds. This must be taken into account in the protocol design.
Low transfer rate	The slowness of some bearers is a major constraint. Therefore, the amount of overhead must be kept in the minimum. For example, with SMS the effective transfer rate may be lower than 100 bit/s.
Limited processing power	The processing power of many mobile terminals is quite limited. This must be taken into account when cryptographic algorithms are chosen.
Limited memory capacity	The memory capacity of most mobile terminals is very modest. Therefore, the number of cryptographic algorithms must be minimised and small-sized algorithms must be chosen. Especially the RAM requirements must be as low as possible.
Restrictions on exporting and using cryptography	International restrictions and rules for using, exporting, and importing cryptography must be taken into account. This means that it must be possible to achieve the best permitted security level according to the legislation of each area. For example, in many cases, strong authentication can be used although strong encryption is prohibited.