IEEE STANDARDS ASSOCIATION

IEEE Standard for Wireless Access in Vehicular Environments—Security Services for Applications and Management Messages

IEEE Vehicular Technology Society

Sponsored by the
Intelligent Transportation Systems Committee

IEEE
3 Park Avenue
New York, NY 10016-5997
USA

26 April 2013

IEEE Std 1609.2™-2013
(Revision of
IEEE Std 1609.2-2006)

IEEE Std 1609™.2-2013
(Revision of
IEEE Std 1609.2-2006)

IEEE Standard for Wireless Access in Vehicular Environments—Security Services for Applications and Management Messages

Sponsor

Intelligent Transportation Systems Committee
of the
IEEE Vehicular Technology Society

Approved 6 February 2013

IEEE-SA Standards Board

Abstract: Secure message formats and processing for use by Wireless Access in Vehicular Environments (WAVE) devices, including methods to secure WAVE management messages and methods to secure application messages are defined in this standard. It also describes administrative functions necessary to support the core security functions.
Keywords: cryptography, IEEE 1609.2™, security, WAVE, wireless access in vehicular environments

The Institute of Electrical and Electronics Engineers, Inc.
3 Park Avenue, New York, NY 10016-5997, USA

PDF: ISBN 978-0-7381-8287-2 STD98169
Print: ISBN 978-0-7381-8288-9 STDPD98169

Notice and Disclaimer of Liability Concerning the Use of IEEE Documents: IEEE Standards documents are developed within the IEEE Societies and the Standards Coordinating Committees of the IEEE Standards Association (IEEE-SA) Standards Board. IEEE develops its standards through a consensus development process, approved by the American National Standards Institute, which brings together volunteers representing varied viewpoints and interests to achieve the final product. Volunteers are not necessarily members of the Institute and serve without compensation. While IEEE administers the process and establishes rules to promote fairness in the consensus development process, IEEE does not independently evaluate, test, or verify the accuracy of any of the information or the soundness of any judgments contained in its standards.

Use of an IEEE Standard is wholly voluntary. IEEE disclaims liability for any personal injury, property or other damage, of any nature whatsoever, whether special, indirect, consequential, or compensatory, directly or indirectly resulting from the publication, use of, or reliance upon any IEEE Standard document.

IEEE does not warrant or represent the accuracy or content of the material contained in its standards, and expressly disclaims any express or implied warranty, including any implied warranty of merchantability or fitness for a specific purpose, or that the use of the material contained in its standards is free from patent infringement. IEEE Standards documents are supplied "**AS IS**."

The existence of an IEEE Standard does not imply that there are no other ways to produce, test, measure, purchase, market, or provide other goods and services related to the scope of the IEEE standard. Furthermore, the viewpoint expressed at the time a standard is approved and issued is subject to change brought about through developments in the state of the art and comments received from users of the standard. Every IEEE standard is subjected to review at least every ten years. When a document is more than ten years old and has not undergone a revision process, it is reasonable to conclude that its contents, although still of some value, do not wholly reflect the present state of the art. Users are cautioned to check to determine that they have the latest edition of any IEEE standard.

In publishing and making its standards available, IEEE is not suggesting or rendering professional or other services for, or on behalf of, any person or entity. Nor is IEEE undertaking to perform any duty owed by any other person or entity to another. Any person utilizing any IEEE Standards document, should rely upon his or her own independent judgment in the exercise of reasonable care in any given circumstances or, as appropriate, seek the advice of a competent professional in determining the appropriateness of a given IEEE standard.

Translations: The IEEE consensus development process involves the review of documents in English only. In the event that an IEEE standard is translated, only the English version published by IEEE should be considered the approved IEEE standard.

Official Statements: A statement, written or oral, that is not processed in accordance with the IEEE-SA Standards Board Operations Manual shall not be considered the official position of IEEE or any of its committees and shall not be considered to be, nor be relied upon as, a formal position of IEEE. At lectures, symposia, seminars, or educational courses, an individual presenting information on IEEE standards shall make it clear that his or her views should be considered the personal views of that individual rather than the formal position of IEEE.

Comments on Standards: Comments for revision of IEEE Standards documents are welcome from any interested party, regardless of membership affiliation with IEEE. However, IEEE does not provide consulting information or advice pertaining to IEEE Standards documents. Suggestions for changes in documents should be in the form of a proposed change of text, together with appropriate supporting comments. Since IEEE standards represent a consensus of concerned interests, it is important to ensure that any responses to comments and questions also receive the concurrence of a balance of interests. For this reason, IEEE and the members of its societies and Standards Coordinating Committees are not able to provide an instant response to comments or questions except in those cases where the matter has previously been addressed. Any person who would like to participate in evaluating comments or revisions to an IEEE standard is welcome to join the relevant IEEE working group at http://standards.ieee.org/develop/wg/.

Comments on standards should be submitted to the following address:

Secretary, IEEE-SA Standards Board
445 Hoes Lane
Piscataway, NJ 08854
USA

Photocopies: Authorization to photocopy portions of any individual standard for internal or personal use is granted by The Institute of Electrical and Electronics Engineers, Inc., provided that the appropriate fee is paid to Copyright Clearance Center. To arrange for payment of licensing fee, please contact Copyright Clearance Center, Customer Service, 222 Rosewood Drive, Danvers, MA 01923 USA; +1 978 750 8400. Permission to photocopy portions of any individual standard for educational classroom use can also be obtained through the Copyright Clearance Center.

Notice to users

Laws and regulations

Users of IEEE Standards documents should consult all applicable laws and regulations. Compliance with the provisions of any IEEE Standards document does not imply compliance to any applicable regulatory requirements. Implementers of the standard are responsible for observing or referring to the applicable regulatory requirements. IEEE does not, by the publication of its standards, intend to urge action that is not in compliance with applicable laws, and these documents may not be construed as doing so.

Copyrights

Updating of IEEE documents

Users of IEEE Standards documents should be aware that these documents may be superseded at any time by the issuance of new editions or may be amended from time to time through the issuance of amendments, corrigenda, or errata. An official IEEE document at any point in time consists of the current edition of the document together with any amendments, corrigenda, or errata then in effect. In order to determine whether a given document is the current edition and whether it has been amended through the issuance of amendments, corrigenda, or errata, visit the IEEE-SA Website at http://standards.ieee.org/index.html or contact the IEEE at the address listed previously. For more information about the IEEE Standards Association or the IEEE standards development process, visit IEEE-SA Website at http://standards.ieee.org/index.html.

Errata

Errata, if any, for this and all other standards can be accessed at the following URL: http://standards.ieee.org/findstds/errata/index.html. Users are encouraged to check this URL for errata periodically.

Patents

Attention is called to the possibility that implementation of this standard may require use of subject matter covered by patent rights. By publication of this standard, no position is taken by the IEEE with respect to the existence or validity of any patent rights in connection therewith. If a patent holder or patent applicant has filed a statement of assurance via an Accepted Letter of Assurance, then the statement is listed on the IEEE-SA Website at http://standards.ieee.org/about/sasb/patcom/patents.html. Letters of Assurance may indicate whether the Submitter is willing or unwilling to grant licenses under patent rights without compensation or under reasonable rates, with reasonable terms and conditions that are demonstrably free of any unfair discrimination to applicants desiring to obtain such licenses.

Essential Patent Claims may exist for which a Letter of Assurance has not been received. The IEEE is not responsible for identifying Essential Patent Claims for which a license may be required, for conducting inquiries into the legal validity or scope of Patents Claims, or determining whether any licensing terms or conditions provided in connection with submission of a Letter of Assurance, if any, or in any licensing agreements are reasonable or non-discriminatory. Users of this standard are expressly advised that determination of the validity of any patent rights, and the risk of infringement of such rights, is entirely their own responsibility. Further information may be obtained from the IEEE Standards Association.

Participants

At the time this standard was completed, the Dedicated Short Range Communication Working Group had the following membership:

Thomas M. Kurihara, *Chair*
William Whyte, *Vice Chair*

Scott Andrews
Lee Armstrong
Roger Berg
William Brownlow
Susan Dickey
Hans-Joachim Fischer
Wayne Fisher
Jon Friedman
Ramez Gerges
Paul Gray
Gloria Gwynne
Ron Hochnadel
Stanley Hsu
Carl Kain
Doug Kavner
David Kelley
John Kenney
Jeremy Landt
Michael Li
Chih-Che (Mike) Lin
Julius Madey
Alastair Malarky
Justin McNew
John Moring
Satoshi Oyama
Gary Pruitt
James D. Randall
Steve Randall
Güner Refi-Tugrul
Randal Roebuck
Richard Roy
Steve Sill
François Simon
Hsieh Tien-Yuan
George Vlantis
Timothy Weil
Andre Weimerskirch
Aaron Weinfield

The following members of the individual balloting committee voted on this standard. Balloters may have voted for approval, disapproval, or abstention.

Iwan Adhicandra
Lee Armstrong
H. Stephen Berger
Mark Bushnell
William Byrd
Kai T. Chen
Keith Chow
Michael Coop
Patrick Diamond
Sourav Dutta
Richard Edgar
Wayne Fisher
Andre Fournier
Ramez Gerges
Paul Gray
Ron Greenthaler
Randall Groves
Tugrul Guener
Gloria Gwynne
Ronald Hochnadel
Werner Hoelzl
Chung-Hsien Hsu
David Hunter
Noriyuki Ikeuchi
Piotr Karocki
John Kenney
Stuart Kerry
Max Kicherer
Thomas M. Kurihara
Paul Lambert
Richard Lancaster
Jeremy Landt
Hsia-Hsin Li
William Lumpkins
Greg Luri
Julius Madey
Alastair Malarky
Sean Maschue
Edward Mccall
Justin Mcnew
John Moring
Michael S. Newman
Satoshi Obara
Satoshi Oyama
Venkatesha Prasad
Markus Riederer
Robert Robinson
Randal Roebuck
Richard Roy
Randall Safier
Bartien Sayogo
Gil Shultz
Di Dieter Smely
Walter Struppler
Dale Sumida
Jasja Tijink
Thomas Tullia
Dmitri Varsanofiev
John Vergis
George Vlantis
Stephen Webb
Hung-Yu Wei
Andre Weimerskirch
William Whyte
Oren Yuen

When the IEEE-SA Standards Board approved this standard on 6 February 2013, it had the following membership:

John Kulick, *Chair*
Richard H. Hulett, *Past Chair*
Konstantinos Karachalios, *Secretary*

Masayuki Ariyoshi	Mark Halpin	Ron Peterson
Peter Balma	Gary Hoffman	Gary Robinson
Farooq Bari	Paul Houzé	Jon Walter Rosdahl
Ted Burse	Jim Hughes	Adrian Stephens
Wael William Diab	Michael Janezic	Peter Sutherland
Stephen Dukes	Joseph L. Keopfinger*	Yatin Trivedi
Jean-Phillippe Faure	David J. Law	Phil Winston
Alexander Gelman	Oleg Logvinov	Yu Yuan

*Member Emeritus

Also included are the following nonvoting IEEE-SA Standards Board liaisons:

Richard DeBlasio, *DOE Representative*
Michael Janezic, *NIST Representative*

Michelle Turner
IEEE Standards Program Manager, Document Development

Michael Kipness
IEEE Standards Program Manager, Technical Program Development

Introduction

This introduction is not part of IEEE Std 1609.2-2013, IEEE Standard for Wireless Access in Vehicular Environments—Security Services for Applications and Management Messages.

5.9 GHz Dedicated Short Range Communications for Wireless Access in Vehicular Environments (DSRC/WAVE, hereafter simply WAVE), as specified in a range of standards including those generated by the IEEE P1609 working group, enables vehicle-to-vehicle (V2V), and vehicle-to-infrastructure (V2I) wireless communications. This connectivity makes possible a range of applications that rely on communications between road users, including vehicle safety, public service, commercial fleet management, tolling, and other operations.

With improved communications come increased risks, and the safety-critical nature of many WAVE applications makes it vital that services be specified that can be used to protect messages from attacks such as eavesdropping, spoofing, alteration, and replay. Additionally, the fact that the wireless technology will be deployed in personal vehicles, whose owners have a right to privacy, means that inasmuch as possible the security services should respect that right and not leak personal, identifying, or linkable information to unauthorized parties.

With this in mind, at the time that IEEE P1609 was established to develop the standards for the WAVE wireless networking protocols, the IEEE also established IEEE P1556™ (later renumbered as IEEE 1609.2) to develop standards for the security techniques that will be used to protect the services that use these protocols. These applications face unique constraints. Many of them, particularly safety applications, are time-critical: the processing and bandwidth overhead due to security must be kept to a minimum, to improve responsiveness and decrease the likelihood of packet loss. For many applications, the potential audience consists of all vehicles on the road in North America; therefore, the mechanism used to authenticate messages must be as flexible and scalable as possible, and must accommodate the smooth removal of compromised WAVE devices from the system. Additionally, as mentioned above, the privacy of privately owned and operated vehicles must be respected as far as technically and administratively feasible.

This document specifies a range of security services for use by WAVE devices. Mechanisms are provided to authenticate WAVE management messages, to authenticate messages from non-anonymous users, and to encrypt messages to a known recipient. Mechanisms to provide anonymous authentication, particularly anonymous broadcast, will be provided in a separate document.

Contents

IEEE Standard for Wireless Access in Vehicular Environments—Security Services for Applications and Management Messages

1. Overview

1.1 Scope

This standard defines secure message formats and processing for use by Wireless Access in Vehicular Environments (WAVE) devices, including methods to secure WAVE management messages and methods to secure application messages. It also describes administrative functions necessary to support the core security functions.

1.2 Purpose

The safety-critical nature of many WAVE applications makes it vital that services be specified that can be used to protect messages from attacks such as eavesdropping, spoofing, alteration, and replay. Additionally, the fact that the wireless technology will be deployed in communication devices in personal vehicles as well as other portable devices, whose owners have an expectation of privacy, means that in as much as possible the security services must be designed to respect privacy and not leak personal, identifying, or linkable information to unauthorized parties. This standard describes security services for WAVE management messages and application messages designed to meet these goals.

1.3 Document organization

Clause 1 provides an overview of the document. Clause 2 contains the normative references. Clause 3 contains definitions and abbreviations. Clause 4 provides a general description of WAVE Security Services and their use. Clause 5 specifies how WAVE Security Services may be used by calling entities to execute specific security use cases. Clause 6 specifies the encoding and structure of messages generated and consumed by WAVE Security Services. Clause 7 defines the primitives used to communicate between WAVE Security Services and other functional entities.

Annex A provides a PICS proforma. Annex B provides a description of the IEEE 1609.2 security profile and a proforma that may be used by developers of applications (or other entities that invoke WAVE Security Services) to specify options for how those applications are to interact with WAVE Security Services. Annex C provides an IEEE 1609.2 security profile for the Basic Safety Message (BSM) specified by the Society of Automotive Engineers (SAE). Annex D provides examples, including examples of encoded protocol data united (PDU) and sample process flows from the point of view of an entity invoking WAVE Security Services rather than from the point of view of WAVE Security services. Annex E provides an informative rationale and FAQ. Annex F is a copyright statement covering the text from Clause 6 that was excerpted from other sources. Annex G provides an informative bibliography.

1.4 Document conventions

Unless otherwise stated, conventions follow those in IEEE Std 802.11™ [B10][1], including conventions for the ordering of information elements within data streams.

Numbers are decimal unless otherwise noted. Numbers preceded by 0x are to be read as hexadecimal, so that 0xFF is equivalent to "FF hexadecimal." Occasionally, this standard includes representations of octet strings in hexadecimal form; these strings are indicated as hexadecimal on a case-by-case basis.

Figures are used for illustration and are informative, unless otherwise noted.

1.5 Note to implementers

Future versions of this standard may differ significantly from the current version. In particular, international harmonization efforts, advances in cryptanalysis, and other considerations make it likely that future versions of this standard will not preserve backwards compatibility with the current version. Changes of data structures in future versions of the standards will be reflected by changes in the version numbers within those data structures.

2. Normative references

The following referenced documents are indispensable for the application of this document (i.e., they must be understood and used, so each referenced document is cited in text and its relationship to this document is explained). For dated references, only the edition cited applies. For undated references, the latest edition of the referenced document (including any amendments or corrigenda) applies.

Federal Information Processing Standard (FIPS) 180-3, Secure Hash Standard, October 2008.[2]

Federal Information Processing Standard (FIPS) 186-3, Digital Signature Standard.

[1] The numbers in brackets correspond to those of the bibliography in Annex G.
[2] FIPS publications are available from the National Technical Information Service (NTIS), U. S. Dept. of Commerce, 5285 Port Royal Rd., Springfield, VA 22161 (http://www.ntis.org/).

IEEE Std 1363™-2000, IEEE Standard Specifications for Public Key Cryptography. [3, 4]

IEEE Std 1363a™-2004, IEEE Standard Specifications for Public Key Cryptography: Additional Techniques.

IEEE P1609.0™ /D1.0, April 2011, Draft Guide for Wireless Access in Vehicular Environments (WAVE) – Architecture.[5]

IEEE Std 1609.3™-2010, Standard for Wireless Access in Vehicular Environments (WAVE) – Networking Services.

IEEE Std 1609.12™-2012, Standard for Wireless Access in Vehicular Environments (WAVE) – Identifier Allocations.

IETF Request for Comments: 3629, UTF-8, A Transformation Format of ISO 10646.[6]

NIMA Technical Report TR8350.2, "Department of Defense World Geodetic System 1984, Its Definition and Relationships With Local Geodetic Systems."[7]

NIST Special Publication SP 800-38C, Recommendation for Block Cipher Modes of Operation: the CCM Mode for Authentication and Confidentiality.[8]

Standards for Efficient Cryptography Group, "SEC 1: Elliptic Curve Cryptography version 2.0."[9]

Standards for Efficient Cryptography Group, "SEC 4: Elliptic Curve Qu-Vanstone Implicit Certificate Scheme (ECQV)," Working Draft Version 0.97, March 2011.

3. Definitions, abbreviations, and acronyms

3.1 Definitions

For the purposes of this document, the following terms and definitions apply. The *IEEE Standards Dictionary Online* should be consulted for terms not defined in this clause. [10]

Advanced Encryption Standard (AES): A U.S. government standard specifying a symmetric block cipher; also, the block cipher specified in that standard.

asymmetric cryptographic algorithm: A cryptographic algorithm that uses two related keys, a public key and a private key, such that the public key is derived from the private key but, given only the public key, it is computationally infeasible to derive the private key.

[3] IEEE publications are available from the Institute of Electrical and Electronics Engineers, 445 Hoes Lane, Piscataway, NJ 08855-1331, USA (http://standards.ieee.org/).
[4] The IEEE standards or products referred to in this clause are trademarks of the Institute of Electrical and Electronics Engineers, Inc.
[5] This IEEE standards project was not approved by the IEEE-SA Standards Board at the time this publication went to press. For information about obtaining a draft, contact the IEEE.
[6] IETF publications are available from http://www.ietf.org.
[7] This technical report is available from http://earth-info.nga.mil/GandG/tr8350/tr8350_2.html.
[8] NIST special publications are available from http://csrc.nist.gov/publications/nistpubs/.
[9] SECG publications are available from http://www.secg.org
[10] *IEEE Standards Dictionary Online* subscription is available at: http://www.ieee.org/portal/innovate/products/standard/standards_dictionary.html.

authenticated channel: A logical communications channel such that the receiver has assurance that the sender is who they claim to be, and that modifications to the data can be detected. A channel may be authenticated by applying cryptographic mechanisms to the channel itself, or by checking the transmitted protocol data units by some out-of-band mechanism.

authentication: A cryptographic service that provides assurance that the sender of a communication is who they claim to be.

Basic Safety Message (BSM): A message, described in SAE J2735, used to announce the status of an object such as a vehicle, pedestrian, or roadside worker.

block cipher: A symmetric encryption algorithm that processes data in blocks, typically of 8 or 16 octets.

calling entity: A functional entity that invokes WAVE Security Services.

certificate: *See* **digital certificate**.

certificate authority (CA): An entity that issues certificates. A correctly functioning certificate authority has procedures in place to ensure that certificates are only issued to entities that are entitled to them.

certificate authority certificate (CA certificate): A certificate that is used to verify other certificates; a type of *security management certificate*.

certificate chain: An ordered list of certificates such that each certificate in the list (except for the first) was issued by the certificate before it in the list. *See also*: **full certificate chain**; **partial certificate chain**.

certificate holder: The entity authorized to use a particular digital certificate to establish trust. The certificate holder can carry out operations using the private key corresponding to the certificate's public key.

Certificate Management Entity (CME): The management entity responsible for managing the certificate information and corresponding key information on a device.

Certificate Revocation List (CRL): A list identifying certificates that have been revoked. *See*: **revocation**.

Certificate Revocation List distribution center (CRL distribution center): A networked entity that stores and distributes certificate revocation lists (CRLs).

Certificate Revocation List series (CRL series): An integer used to assign different certificates issued by the same certificate authority to distinct sets, such that the certificates in different sets appear on different revocation lists if revoked.

Certificate Signing Request (CSR): A communication from an entity to a certificate authority, requesting that the certificate authority issue a certificate on behalf of the entity.

communications certificate: A certificate used to sign either secure data or a *WAVE Service Advertisement*.

computationally infeasible: An operation that requires an exceptionally large amount of computing power to complete. The threshold for computational infeasibility will rise over time as computing power at a given price point increases.

Counter Mode with Cipher Block Chaining Message Authentication Code (CCM): A mode of operation of a block cipher in which the data is encrypted with a keystream, which in turn is generated by encrypting an incrementing counter, and in turn authenticated with a message authentication code calculated using cipher block chaining mode.

cryptographic verification: The process of determining whether a signature on a signed communication is consistent with the communication and the private key.

cryptographically secure hash function: A function that maps an arbitrary-length input into a fixed-length output (the hash value) such that (a) it is computationally infeasible to find an input that maps to a specific hash value and (b) it is computationally infeasible to find two inputs that map to the same hash value. All hash functions used in this document are cryptographically secure hash functions.

cryptomaterial: A private key, a public key, or a certificate.

Cryptomaterial Handle: A reference to a private key and the associated public key or certificate, stored within the security processing services.

decrypt: To convert unreadable, encrypted data to readable, decrypted data using a decryption algorithm and a key. *Contrast*: **decode, encrypt.**

decryption algorithm: An algorithm that takes as input ciphertext and a key and (if the correct key is provided) produces the original plaintext.

delta Certificate Revocation List (delta CRL): A certificate revocation list that carries information only about certificates that were revoked within a certain time period. *Contrast*: **full Certificate Revocation List**.

digital certificate: A digitally signed document binding a public key to an identity and/or a set of permissions.

dubious certificate: A certificate for which the most recent certificate revocation list has not yet been received.

Elliptic Curve Cryptography (ECC): A form of public-key cryptography based on the problem of finding discrete logarithms in a group defined over elliptic curves.

Elliptic Curve Digital Signature Algorithm (ECDSA): A specific digital signature mechanism based on the elliptic curve discrete logarithm problem.

Elliptic Curve Integrated Encryption Scheme (ECIES): A public-key encryption mechanism based on the elliptic curve discrete logarithm problem.

encode: To convert a data structure into an array of octets. *Contrast*: **decode, encrypt.**

encrypt: To convert readable data to unreadable, encrypted data using an encryption algorithm and a key. *Contrast*: **decrypt, encode.**

encryption algorithm: An algorithm that takes as input plaintext and a key and produces ciphertext.

encryption certificate: A certificate that contains an encryption key.

end-entity certificate: Any certificate other than one used by a CA for certificate management operations.

enrolment certificate: A certificate used to sign a Certificate Signing Request. A type of *security management certificate*. *Contrast*: **operational certificate**.

explicit certificate: A certificate that contains a public key and the certificate authority's signature

full certificate chain: A certificate chain in which the first certificate is a root certificate and the last certificate is an end-entity certificate.

full Certificate Revocation List (full CRL): A certificate revocation list that carries information about certificates that were revoked and have not expired, whenever the revocation took place. *Contrast*: **delta Certificate Revocation List**.

hash function: *See*: **cryptographically secure hash function.**

hash value: The output of a hash function.

IEEE 1609.2™ security profile: A specification of the options and parameters provided to the 1609.2 security services by a particular consumer of those services.

implicit certificate: A digital certificate that does not explicitly contain the certificate holder's public key, but instead allows the public key to be reconstructed from a reconstruction value and the certificate authority's public key

inherited permissions: Permissions within a subordinate certificate that are communicated by reference to the issuing certificate, rather than stated explicitly within the subordinate certificate.

issuing certificate: If the public key from certificate A can be used to verify the signature on certificate B, then A is the *issuing certificate* for B. *Contrast*: **subordinate certificate.**

Local Service Index for Security (LSI-S): An identifier used to identify a secure communications entity to the security services.

operational certificate: Any certificate other than an enrolment certificate.

operational permissions: The actions a certificate holder is allowed to take as stated in their certificate. Expressed in this standard using provider service identifiers.

partial certificate chain: A certificate chain that is not a full certificate chain (i.e., the first certificate is not a root certificate or the last certificate is not an end-entity certificate).

plaintext: Unencrypted data.

properly formed certificate: A certificate that can be parsed correctly according to the data structures and encoding defined in this standard.

provider: Advertiser of a WAVE service. See also: **user**; **WAVE Service Advertisement**.

Provider Service Identifier (PSID): An octet string that identifies a service provided by a higher layer entity.

public-key digital signature: A cryptographically secure checksum that is generated using a private key and verified using a public key.

reconstruction value: A value in an implicit certificate that allows the associated public key to be recovered.

replay attack: An attack in which an attacker retransmits or delays data that was originally validly transmitted.

revocation: The publication by a relevant authority of the information that a particular certificate is no longer to be trusted.

root certificate: A self-signed certificate that can be used as a trust anchor to verify other certificates.

secure communications entity: A functional entity that uses IEEE 1609.2 services to secure communications for any purpose other than security management.

secure data exchange certificate: A certificate that is used to verify the signature on data other than a *WAVE service advertisement*. *Contrast*: **WSA certificate**, **communications certificate**.

Secure Data Exchange Entity (SDEE): A secure communications entity that uses IEEE 1609.2 services to secure any communications other than *WAVE Service Advertisements*.

secure provider service: A higher layer entity that requests that its entry in a WAVE Service Advertisement be secured.

security management: Operations that support acquiring or establishing the validity of 1609.2 certificates.

security management certificate: A certificate used to authenticate security management protocol data units. *Enrolment certificates* and *certificate authority certificates* are security management certificates. *Contrast*: **communications.**

security management message: A protocol data unit used to manage certificates or information about certificates.

self signed certificate: A certificate whose signature can be verified with the public key in the certificate.

Service Specific Permissions (SSP): A field that encodes permissions relevant to a particular certificate holder.

ServiceInfo: A field in an IEEE Std 1609.3 *WAVE service advertisement* (WSA) that announces the availability of a service. A WSA may contain multiple ServiceInfo fields.

signature: See **public key digital signature**.

store: A collection of data objects (typically certificates and certificate revocation lists in this standard) such that each object stays in the store until it expires, is superseded, or is revoked.

symmetric cryptographic algorithm: A cryptographic algorithm that uses a single key. Knowledge of a symmetric encryption key allows both encryption and decryption. Knowledge of a symmetric authentication key allows generation and verification of message authentication codes.

symmetric key: *See*: **symmetric cryptographic algorithm**.

Transport Layer Security (TLS): An Internet Engineering Task Force (IETF) standard providing for secure communications over Transmission Control Protocol/Internet Protocol (TCP/IP).

trust anchor: A certificate whose validity does not depend on the validity of other certificates.

user: Consumer of a WAVE service. See also: **provider**; **WAVE Service Advertisement**.

valid certificate: A certificate that is correctly formed, that has not been revoked or expired, and for which a certificate chain to a trust anchor can be constructed.

vehicle-to-infrastructure: Wireless communication between vehicles and roadside units in either direction

Wireless Access in Vehicular Environments (WAVE) device: A device that is compliant to IEEE Std 1609.3™, IEEE Std 1609.4™ and IEEE Std 802.11™, operating outside the context of a basic service set (BSS). (See IEEE Std 802.11 specification of "STA transmission of data frames outside the context of a BSS.")

Wireless Access in Vehicular Environments (WAVE) Management Entity (WME): A set of management functions, defined in IEEE Std 1609.3, required to provide WAVE Networking Services.

Wireless Access in Vehicular Environments (WAVE) Service Advertisement (WSA): A collection of data indicating the services that a given WAVE device is offering.

Wireless Access in Vehicular Environments (WAVE) Service Advertisement (WSA) certificate: A certificate used to verify a wireless access in *WAVE Service Advertisement*.

Wireless Access in Vehicular Environments (WAVE) Short Message (WSM): A packet consisting of a data payload and a *WAVE Short Message Protocol* header.

Wireless Access in Vehicular Environments (WAVE) Short Message Protocol (WSMP): A low-overhead protocol for exchange of messages in a rapidly varying radio frequency environment where low latency is also an important objective.

3.2 Abbreviations and acronyms

AES	Advanced Encryption Standard
ASN.1	Abstract Syntax Notation 1
BSM	Basic Safety Message
CA	certificate authority
CCM	counter with cipher block chaining message authentication code
CME	Certificate Management Entity
CMH	Cryptomaterial Handle
CRL	Certificate Revocation List
CSR	Certificate Signing Request
DSRC	Dedicated Short Range Communications
ECC	Elliptic Curve Cryptography
ECDSA	Elliptic Curve Digital Signature Algorithm
ECIES	Elliptic Curve Integrated Encryption Scheme

FIPS	Federal Information Processing Standard
GPS	Global Positioning System
IETF	Internet Engineering Task Force
IP	Internet Protocol
IPv6	Internet Protocol version 6
ITS	Intelligent Transportation Systems
LSI-S	Local Service Index for Security
MAC	medium access control, or message authentication code
NIST	National Institute for Standards and Technology
OBU	on-board unit
PDF	probability distribution function
PDU	protocol data unit
PHY	Physical Layer
PSC	Provider Service Context
PSID	Provider Service Identifier
PSSME	Provider Service Security Management Entity
RFC	Request for Comments
RSU	roadside unit
SAE	Society of Automotive Engineers
SAP	Service Access Point
SDEE	Secure Data Exchange Entity
SP	special publication
SSP	Service Specific Permissions
TAI	International Atomic Time
TLS	Transport Layer Security
TTP	Trusted Third Party
UTC	Coordinated Universal Time
UTF	Unicode Transformation Format

V2I vehicle-to-infrastructure

V2V vehicle-to-vehicle

WAVE Wireless Access in Vehicular Environments

WGS World Geodetic System

WME WAVE Management Entity

WSA WAVE Service Advertisement

WSM WAVE Short Message

WSMP WAVE Short Message Protocol

4. General description

4.1 WAVE protocol stack overview

Wireless Access in Vehicular Environments (WAVE) provides a communication protocol stack optimized for the vehicular environment, employing both customized and general-purpose elements as shown in Figure 1. See IEEE P1609.0[11] for an overview of the WAVE architecture and services, including a description of the use of the security services described in this document.

[11] Information on references can be found in Clause 2.

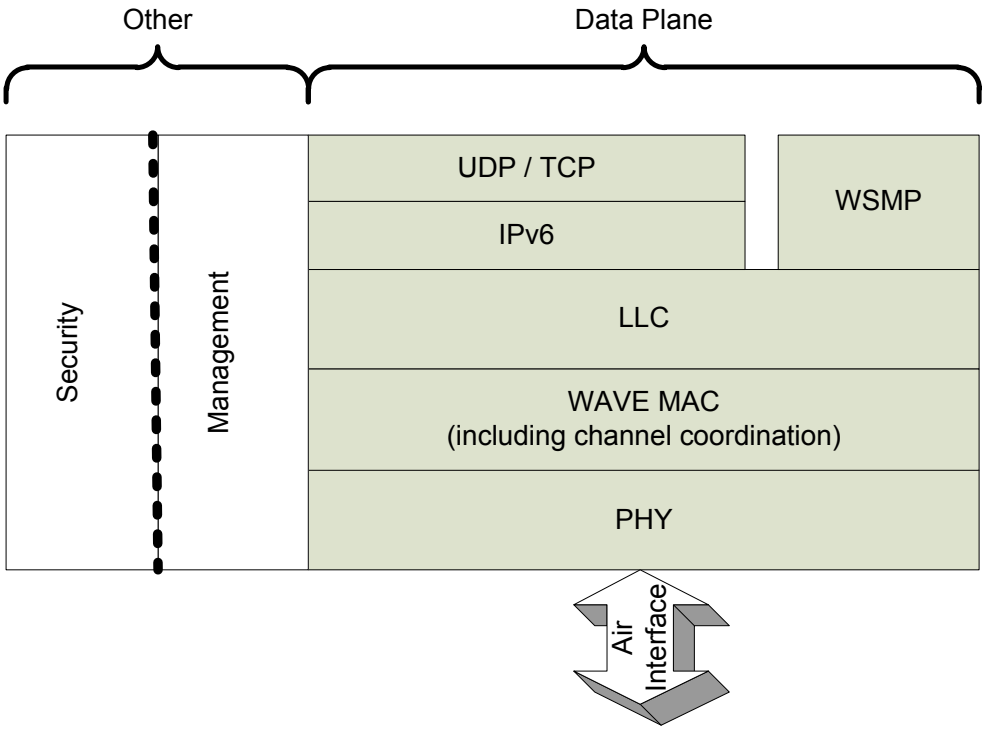


Figure 1—WAVE Protocol Stack

This standard specifies a set of management plane WAVE Security Services available to applications and processes running in the various layers of the WAVE protocol stack as shown in Figure 2. The WAVE Security Services consist of:

- *security processing services*:
  - provide processing that is performed to enable *secure communications that comprise* secure data and secure WAVE Service Advertisements (WSAs); and
- *security management services*:
  - provice certificate management services that are provided by the Certificate Management Entity (CME) and that manage information related to the validity of all certificates; and
  - provide service security management services that are provided by the Provider Service Security Management Entity (PSSME) and that manage information related to certificates and private keys that are used to send secured WSAs.

The services and entities within the WAVE Security Services are shown in Figure 2, which also shows Service Access Points (SAPs) that support communications between WAVE Security Services entities and other entities. This standard specifies the security processing via primitives defined at these SAPs. Horizontal boundaries within the WAVE Security Services in Figure 2 are abstract and do not correspond to horizontal boundaries within the Data Plane.

Data passed across the SAPs in this standard is assumed to be secure and trustworthy. This standard does not provide mechanisms to ensure the trustworthiness of this data.

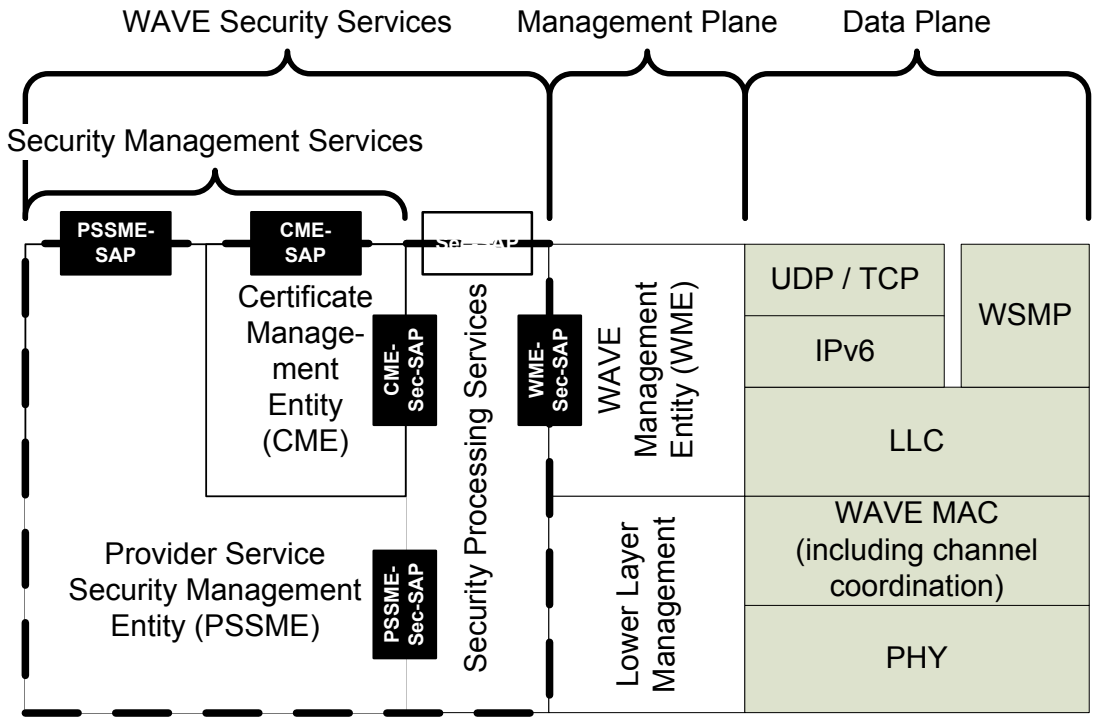


Figure 2—WAVE Protocol Stack showing detail of the security services

4.2 Generic security services

IEEE 1609 provides WAVE Security Services that in turn support the following generic security services:

- Confidentiality: Supported by the security processing services *Generate encrypted data* and *Decrypt encrypted data*.
- Authentication: Supported by the security processing services *Generate signed data* and *Verify signed data*.
- Authorization: Supported by the security processing services *Generate signed data* and *Verify signed data*.
- Integrity: Supported by the security processing services *Generate signed data* and *Verify signed data*.

4.3 Security processing services

4.3.1 General

The security processing services accept input requests from calling entities. The security processing services respond to each input request with a corresponding output confirmation containing the results of the request.

The security processing services consist of the following:

— Transforming unsecured Protocol Data Units (PDUs) into secured PDUs to be transferred using data-plane operations, and processing secured PDUs on reception, including transforming secured PDUs into unsecured PDUs. These processes together are referred to as "Secure data exchange." The security services specified in this document support the signing and/or encryption of PDUs prior to their transmission, and the decryption and verification (as appropriate) of PDUs on reception.

— Generating secured WSAs and processing signed WSAs on reception (hereafter, "Signed WSAs"). WSAs are defined in IEEE Std 1609.3. They originate at the WAVE Mangement Entity (WME) in the transmitting device, and are consumed by the WME in the receiving device. The WME is also defined in IEEE Std 1609.3. The security services specified in this document support the signing of WSAs prior to their transmission and the verification of WSAs on reception.

— Security processing for security management: ensuring access to cryptomaterial (private keys, public keys and certificates) as described in 4.4; generating certificate requests and processing responses; and validating Certificate Revocation Lists (CRLs).

— Storing private keys and their associated certificates.

An implementation of WAVE security services shall include at least one of the following security processing services:

— Generate signed data (see 5.3.2)

— Generate encrypted data (see 5.3.3)

— Verify signed data (see 5.3.6)

— Decrypt encrypted data (see 5.3.5)

— Generate signed WSA (see 5.4.1)

— Verify signed WSA on reception (see 5.4.2)

— Generate certificate request (see 5.6.1)

— Verify response to certificate request (see 5.6.2)

— Verify certificate revocation list (see 5.6.4)

4.3.2 Secure data exchange

The secure data exchange operations are:

— Generate signed data (see 5.3.2)

— Generate encrypted data (see 5.3.3)

— Verify signed data (see 5.3.6)

— Decrypt encrypted data (see 5.3.5)

The information elements used by the secure data exchange operations are specified in 7.2 as parameters to primitives. The specifier of an application may use the *IEEE 1609.2 security profile* as a compact way to communicate secure data exchange parameters to multiple implementers. The IEEE 1609.2 security profile is a set of fixed parameters or parameter ranges to be supplied to the secure data exchange operations when applying security to PDUs for a specific application or service. IEEE 1609.2 security profiles are described in Annex B and may be specified using the proforma given in B.3. Annex C provides a normative example of an IEEE 1609.2 security profile for a specific calling entity, namely one that sends or receives the Basic Safety Message (BSM) as specified in the Society of Automotive Engineers (SAE) J2735 standard.

Communication of IEEE 1609.2 security profiles is outside the scope of this standard and is discussed in Annex B.

4.3.3 Signed WSAs

The Signed WSA operations are:

- Generate signed WSA (see 5.4.1)
- Verify signed WSA on reception (see 5.4.2)

The information elements used by the signed WSA operations are specified in 7.3.

4.3.4 Processing for security management

The processing for security management operations are:

- Generate certificate request (see 5.6.1)
- Verify response to certificate request (see 5.6.2)
- Verify certificate revocation list (see 5.6.4)

The information elements used by these operations are specified in 7.2.

4.4 Cryptomaterial

4.4.1 General

Cryptographic operations within the security processing services make use of data of the following types:

- Private keys and associated public keys.
- Private keys and associated certificates.
- Digital certificates held by peer entities, for which no private key is stored by the security processing services.

This data is collectively referred to as *cryptomaterial*.

Private keys and associated certificates are used by the security processing services in the following operations:

- Generate signed data
- Decrypt encrypted data on reception
- Generate signed WSA

WAVE security services shall reject a request to carry out an operation that uses a private key and certificate if the private key and certificate provided to the operation do not form a valid pair as defined in 5.8.4.

Private keys and associated public keys are used during certificate request as described in 5.6.1.1.

WAVE security services shall reject a request to carry out an operation that uses a private key and a public key if the private key and public key provided to the operation do not form a valid pair as defined in 5.8.4.

Certificates held by peer entities are used in the following operations:

— Verify signed data on reception

— Generate encrypted data

— Verify signed WSA on reception

Private keys are held by the security processing services and managed through cryptomaterial handles, specified in 4.4.2. Certificates held by peer entities are managed by the CME, specified in 4.5.1.

The cryptographic mechanisms supported by this standard are the Elliptic Curve Digital Signature Algorithm (ECDSA) and the Elliptic Curve Integrated Encryption Scheme (ECIES). Cryptographic mechanisms are specified in 5.8.

4.4.2 Private key storage and cryptomaterial handles

Within this standard, private keys and the associated public keys and certificates are stored within the security processing services and referenced using a Cryptomaterial Handle (CMH). A CMH is an index used to reference storage within the security processing services. A CMH references a private key and a public key, or a private key and a certificate. Mechanisms for initializing a CMH and storing keys and certificates using a CMH are specified in 5.2.3. These mechanisms ensure that a private key and a public key, or a private key and a certificate, referenced by a CMH always form a valid pair.

Each secure data exchange entity (SDEE) involved in a secure data exchange operation directly references its key and certificate using a CMH. This is illustrated in Figure 3, where the dashed lines indicate the functionality specified in this standard.

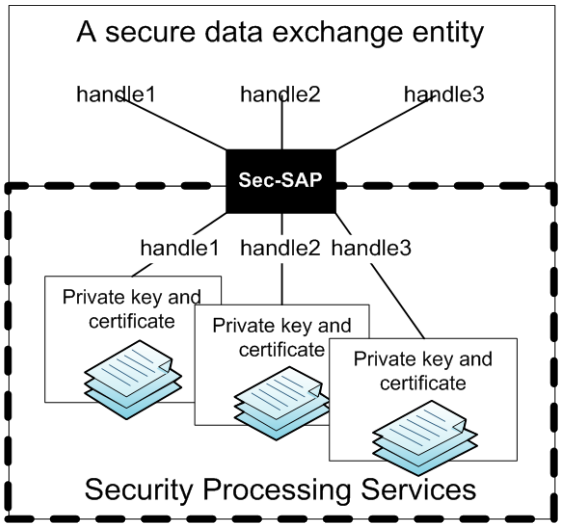


Figure 3—Secure data exchange entity and cryptomaterial handle

Private keys and certificates necessary to sign WSAs are stored and managed by the PSSME as described in 5.2.3. This is illustrated in Figure 4, where the dashed lines indicate the functionality specified in this standard. The PSSME references the private keys and certificates by means of CMHs.

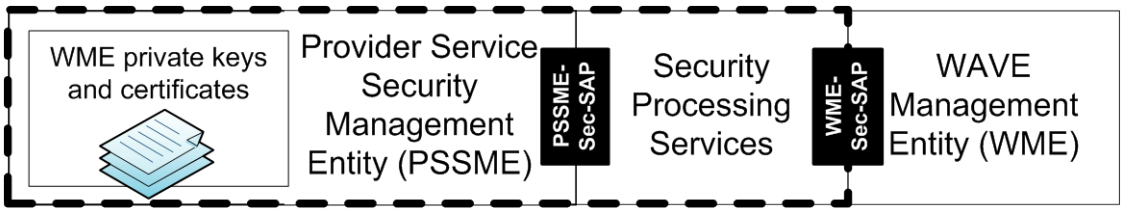


Figure 4—Relationship among WME, WSA private keys, and certificates, and the WAVE Security Services

NOTE—This standard does not provide a mechanism to remove CMHs or the keys associated with them if they are no longer needed. Implementations may provide this mechanism, for example to save storage space on the WAVE device, or to prevent the accidental use of expired or otherwise invalid cryptomaterial.

4.5 Security management services

4.5.1 Certificate Management Entity

The Certificate Management Entity (CME) manages the following information that allows the security processing services to determine the trustworthiness of certificates and received data:

— Revocation and other status information about certificates for which the corresponding private key is stored by the security processing services.

— Revocation and information about certificates for which the corresponding private key is not stored by the security processing services (i.e., those belonging to peer WAVE security services and to CAs).

— Certificates that may be used as a trust anchor.

Additionally, the CME manages information to allow it to determine whether or not a received secure PDU is an exact duplicate of a previously received PDU.

As illustrated in Figure 2, the CME has two Service Access Points (SAPs) through which other entities may communicate with it to obtain and update security management information: one for use by the security processing services (CME-Sec-SAP), and one for use by other entities (CME-SAP).

Certificate management information is generated by CAs and may be obtained by a number of different methods by entities that communicate with WAVE Security Services (see 5.6.4). This standard refers to certificates whose management information is available within the CME as certificates that are "known to" the CME.

NOTE—This standard specifies mechanisms for providing certificate management information to the CME but not for removing information that is no longer relevant. Implementations may implement such mechanisms.[12]

4.5.2 Provider Service Security Management Entity

The Provider Service Security Management Entity (PSSME) enables the sharing of security management information, including cryptomaterial, between the WME and higher layer entities that request to be advertised as secure provider services. The process of requesting to be advertised is specified in IEEE Std 1609.3. The PSSME provides the following functionality:

— Managing the assignment of a Local Service Index for Security (LSI-S) to allow the security processing services to distinguish between higher layer entities.

— Registration of permissions for secure provider services, to allow processes outside the PSSME to apply for WSA signing certificates.

— Communication of certificates and private keys to the security processing services, to allow the security processing services to sign WSAs on behalf of the WME.

— For each known WSA signing certificate, the PSSME may store the most recently accepted WSA signed by that certificate, to allow for fast acceptance of repeated WSAs.

PSSME operations are specified in 5.4 and 5.6.1. The PSSME's relationship with the security processing services and with the WME is illustrated in Figure 4.

[12] Notes in text, tables, and figures of a standard are given for information only and do not contain requirments needed to implement this standard.

5. Security services

5.1 General

Figure 5 shows the general model for security processing. Security services are invoked by a secure communications entity (SCE) and return their output to the same SCE.

The sending entity (which may be an application, another higher layer entity, the WME or any other entity) invokes the security services to perform sender-side security processing. The results of the processing are returned to the sending entity, which then transmits the resulting application protocol data unit (APDU).

The receiving entity receives the secure APDU. It then invokes the security services to perform security processing on the contents of the APDU. The security services return the result to the receiving entity for further processing which may include multiple invocations of the security services if necessary.

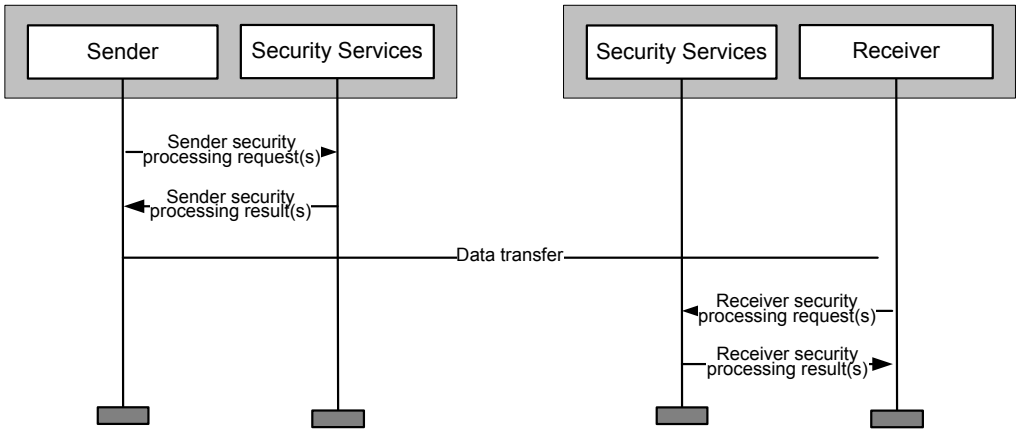


Figure 5—Process flow for use of 1609.2 security services

5.2 Preconditions for secure processing

5.2.1 Secure data exchange

The security processing services uniquely identify each SDEE using a Local Service Identifier for Security (LSI-S) which is an integer generated by the security processing services. The security services ensure that each LSI-S value is only allocated once. Implementations may choose a different mechanism to support unique identification of entities, as long as that mechanism supports all of the use cases for the LSI-S described in this standard.

NOTE—The LSI-S is only used by SDEEs that are going to request that the security processing services use replay protection on incoming messages. See E.5.8 for more discussion of replay protection. An SDEE that does not use replay protection on incoming messages need not request an LSI-S.

The process for obtaining an LSI-S is shown in Figure 6 where the dashed lines indicate functionality defined in this standard. Primitive names in the figure are abbreviated for compactness. The calling entity obtains its LSI-S via Sec-LocalServiceIndexForSecurity.request (Sec-LSIS.req in the diagram) and the LSI-S is returned via Sec-LocalServiceIndexForSecurity.confirm (Sec-LSIS.cfm).

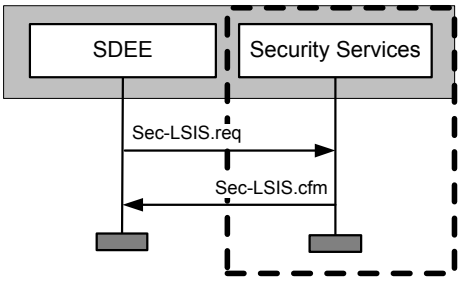


Figure 6—Process flow for obtaining a Local Service Identifier for Security (LSI-S) for a secure data exchange entity (SDEE)

5.2.2 Secure WSAs

5.2.2.1 Local service index for security

A higher layer entity that requests that its entry in a WSA is secured, is referred to as a *secure provider service*. The security processing services uniquely identify each secure provider service using a LSI-S which is a locally unique integer generated by the security processing services.

The calling entity obtains its LSI-S via PSSME-LocalServiceIndexForSecurity.request (PSSME-LSIS.req in the diagram) and the LSI-S is returned via PSSME-LocalServiceIndexForSecurity.confirm (PSSME-LSIS.cfm).

The process for obtaining an LSI-S is shown in Figure 7 where the dashed lines indicate functionality defined in this standard. Primitive names in the figure are abbreviated for compactness.

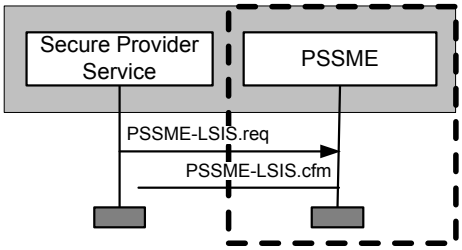


Figure 7—Process flow for obtaining a Local Service Identifier for Security (LSI-S) for a secure provider service

NOTE 1—The LSI-S for secure provider services and the LSI-S for secure data exchange are always used in different contexts and so in principle, a secure provider services LSI-S and a secure data exchange LSI-S may have the same value.

NOTE 2—The purpose of the LSI-S is to enable the WME to reference the secure provider service when requesting that the security processing services sign a WSA. This allows the security processing services to determine whether or not a valid WSA certificate exists for the combination of secure provider services in the WSA.

NOTE 3—The secure provider service relays the LSI-S to the WME via WME-ProviderService.request as defined in IEEE Std 1609.3.

NOTE 4—A sample process flow covering the inclusion of a secure provider service in a signed WSA, including certificate provision, is given in D.4.

5.2.2.2 Registering secure provider service permissions with PSSME

Each secure provider service has specified permissions associated with it. These are denoted by a Provider Service Identifier (PSID), a maximum allowable priority, and (optionally) a set of Service Specific Permissions (SSP). The mapping between permissions and certificates is managed by the PSSME.

The PSSME is requested to register secure provider service permissions via PSSME-SecuredProvider-Service.request. The information elements used by the PSSME are specified in 7.4.3.

The PSSME acknowledges the registration request via PSSME-SecuredProviderService.confirm.

The process for registering permissions is shown in Figure 8 where the dashed lines indicate functionality defined in this standard. Primitive names in the figure are abbreviated for compactness.

Sample process flows to obtain and use certificates for WSA signing are described in D.3.2, D.4.

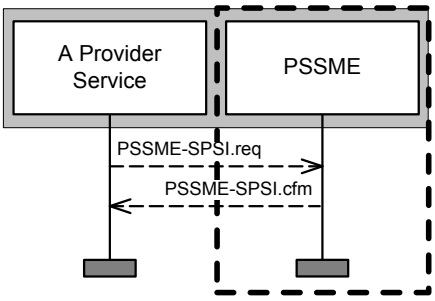


Figure 8—Process flow for registering secure provider service permissions

5.2.3 Cryptomaterial

5.2.3.1 General

In this standard, private keys and their associated public keys or certificates are stored internally to the security processing services. An invoking entity references a particular stored private key for use in a particular operation by passing a Cryptomaterial Handle (CMH) to the security processing services. The security processing services manage the state of the CMH in response to requests from external entities.

There are three states defined for a CMH, as follows:

— *Initialized*: A CMH in *Initialized* state does not reference any cryptomaterial.

— *Key Pair Only*: A CMH in *Key Pair Only* state references a private key and the corresponding public key.

— *Key and Certificate*: A CMH in *Key and Certificate* state references a private key and the corresponding certificate.

During the course of operations, the security processing services may directly access the certificate referenced by a CMH in *Key and Certificate* state or the public key referenced by a CMH in *Key Pair Only* state.

Figure 9 shows the state diagram for Cryptomaterial Handles, including an indication of the primitives that may be used to transition from one state to the next.

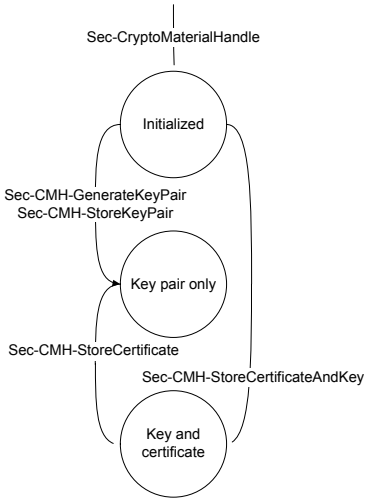


Figure 9—State diagram for Cryptomaterial Handles

5.2.3.2 Initialization

A CMH is created in the *Initialized* state as shown in Figure 10, where the dashed lines indicate functionality defined in this standard. Primitive names in the figure are abbreviated for compactness. The calling entity creates the CMH via Sec-CryptomaterialHandle.request (Sec-CMH.req in the diagram) and the CMH is returned via Sec-CryptomaterialHandle.confirm (Sec-CMH.cfm).

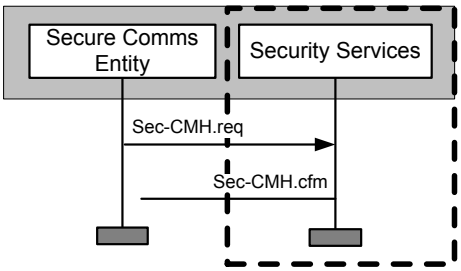


Figure 10—Process flow for obtaining a new Cryptomaterial Handle

5.2.3.3 Transition to *Key Pair Only* state

On invocation of Sec-CryptomaterialHandle-StoreKeyPair.request referencing a CMH that is in the *Initialized* state, the security processing services shall store the enclosed private key and place the CMH in the *Key Pair Only* state. The security processing services confirm the operation back to the invoking Secure Communications Entity via Sec-CryptomaterialHandle-StoreKeyPair.confirm. This is illustrated in Figure 11.

On invocation of Sec-CryptomaterialHandle-GenerateKeyPair.request for a CMH that is in the Initialized state, the security processing services shall generate a private key and public key pair, store them with the CMH and confirm the operation back to the invoking Secure Communications Entity in a Sec-CryptomaterialHandle-GenerateKeyPair.confirm, thus causing the CMH to enter the Key Pair Only state. This is illustrated in Figure 12.

A CMH shall only be placed in the *Key Pair Only* state if the private and public keys referenced by the CMH form a valid key pair for the given cryptographic algorithm. Key pair validity is established for ECDSA and ECIES as specified in 5.8.

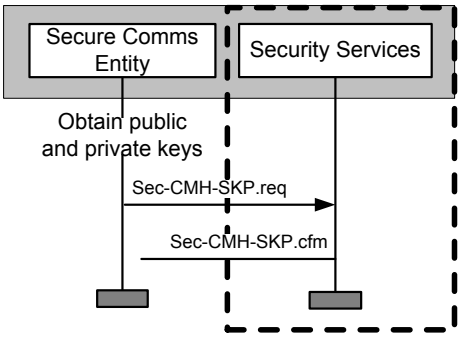


Figure 11—Transitioning a CMH from *Initialized* to *Key Pair Only* state with externally generated keys

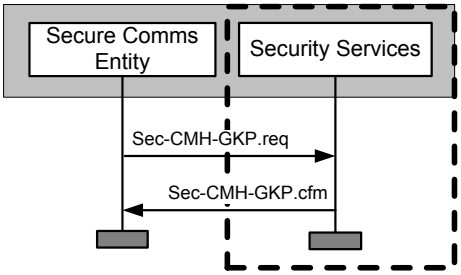


Figure 12—Transitioning a CMH from *Initialized* to *Key Pair Only* state with keys generated by the security services

5.2.3.4 Transition to *Key and Certificate* state

On invocation of Sec-CryptomaterialHandle-StoreCertificateAndKey.request for a CMH that is in the *Initialized* state, the security processing services shall store the enclosed private key and certificate and confirm the operation back to the invoking Secure Communications Entity via Sec-CryptomaterialHandle-StoreCertificateAndKey.confirm, thus causing the CMH to enter the *Key and Certificate* state. This is illustrated in Figure 13, where the primitive names are abbreviated to Sec-CMH-SCAK.req and Sec-CMH-SCAK.cfm respectively.

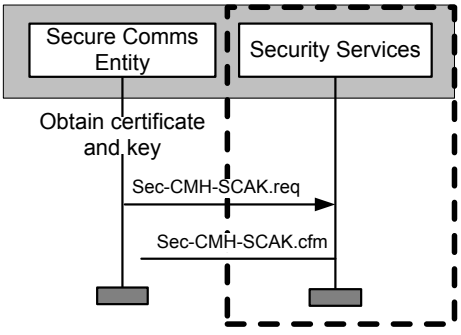


Figure 13—Transitioning a CMH directly from *Initialized* to *Key and Certificate* state

On invocation of a Sec-CryptomaterialHandle-StoreCertificate.request for a CMH that is in the *Key Pair Only* state, the security processing services shall store the enclosed certificate with the CMH and confirm the operation back to the invoking SCE in a Sec-CryptomaterialHandle-StoreCertificate.confirm, thus causing the CMH to enter the *Key and Certificate* state. This is illustrated in Figure 14.

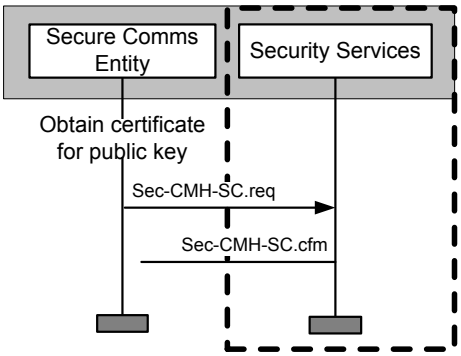


Figure 14 — Transitioning a CMH from *Initialized* to *Key and Certificate* state via *Key Pair Only* state with externally generated keys

A CMH shall only be placed in the *Key and Certificate* state if the private key and the public key indicated by the certificate form a valid key pair for the given cryptographic algorithm and if the certificate is part of a valid certificate chain terminating in a trust anchor. Key pair validity for ECDSA and ECIES may be assured using the processing specified for Sec-CryptomaterialHandle-StoreCertificate.request (7.2.9) and Sec-CryptomaterialHandle-StoreCertificateAndKey.request (7.2.11). The validity of the certificate chain shall be established according the criteria given in 5.5.2. This may also be assured using the processing specified for Sec-CryptomaterialHandle-StoreCertificate.request and Sec-CryptomaterialHandle-StoreCertificateAndKey.request.

NOTE 1—This standard intentionally does not provide primitives that may be used to extract the private key from a CMH.

NOTE 2—This standard does not provide a primitive that allows a private key to be imported to a CMH in encrypted form, but implementations may provide such a primitive.

5.3 Secure data exchange

5.3.1 General

This section specifies the process flow for the following specific uses of the security processing services for secure data exchange:

- Sign data
- Encrypt data
- Sign and encrypt data
- Process received secure data

All of these uses follow the same template:

— A calling entity invokes the security services via a request primitive to request processing of specific data.

— The security services carry out the requested processing and return the result to the calling entity via a confirm primitive. This returned values include a result code indicating whether the request succeeded or failed.

NOTE—This standard uses the term "reject a request" as shorthand for "return an error code in the confirm primitive corresponding to the request."

Actions taken by the calling entity before or after it invokes the security services are out of scope of this standard.

5.3.2 Sign data

The security processing services are requested to sign data via Sec-SignedData.request. The information elements used by the security processing services to generate signed data are specified in 7.2.13. These information elements include a CMH in *Key and Certificate* state.

The security processing services return the result of the request (either an octet string containing the signed data or an error code) to the SDEE via Sec-SignedData.confirm. On success, the octet string returned shall be a valid encoding of a 1609Dot2Data as defined in Clause 6.

The security processing services shall reject a request to sign data if the certificate is not currently valid or consistent with the data according to the criteria specified in 5.5 or if the CMH is not valid according to the criteria specified in 5.2.3.4, or if the CMH is not in *Key and Certificate* state. An implementation of WAVE security services may have an upper bound on full certificate chain length and may reject a request to sign data if the chain from the certificate to a trust anchor, as described in 5.5.2, is longer than that upper bound.[13]

The signing process is illustrated in Figure 15, where the dashed lines indicate entities and processing that are defined in this standard. Primitive names in the figure are abbreviated for compactness.

NOTE—Any changes made to the signed data after signing before sending it invalidate the digital signature, resulting in invalid signed data.

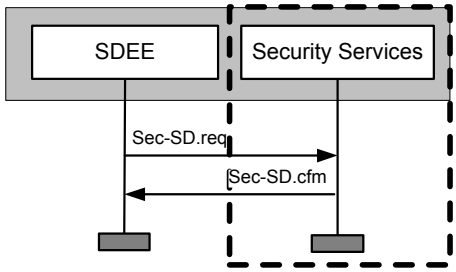


Figure 15—Process flow for signing data

[13] The Protocol Conformance Implementation Statement (PICS) in Annex A allows an implementation to state the maximum certificate chain length that it supports.

5.3.3 Encrypt data

The security processing services are requested to encrypt data via Sec-EncryptedData.request. The information elements used by the security processing services to generate encrypted data are specified in 7.2.15.

The security processing services return the result of the request (either an octet string containing the encrypted data, or an error code) via Sec-EncryptedData.confirm. On success, the octet string returned shall be a valid encoding of a 1609Dot2Data as defined in Clause 6.

The security processing services shall reject a request to encrypt data to a certificate that is not valid according to the criteria specified in 5.5. WAVE security services may reject a request to encrypt data if the number of requested recipients is greater than 6.

A single input PDU may be encrypted for multiple recipients, resulting in a single encrypted PDU that may be decrypted by any of those recipients.

The encryption process is illustrated in Figure 16, where the dashed lines indicate entities and processing that are defined in this standard.

NOTE—Encrypted data created by the security processing services includes an integrity check value. Changes made to the encrypted data before sending it may invalidate this check value resulting in the recipient rejecting the encrypted data.

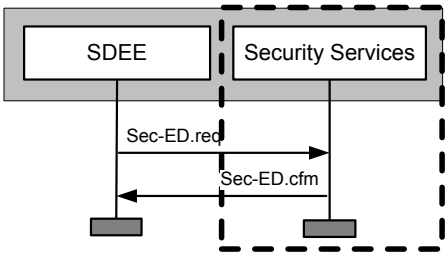


Figure 16—Process flow for encrypting data

5.3.4 Sign and encrypt data

The security processing services may be requested to create signed and encrypted data by successive invocations of Sec-SignedData.request and Sec-EncryptedData.request.

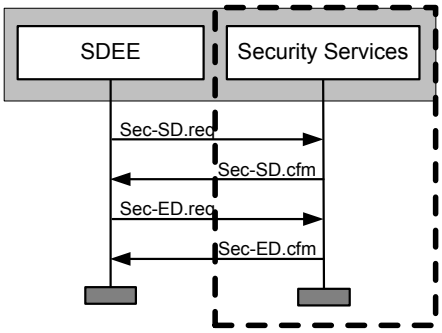


Figure 17 — Process flow for generating signed and encrypted data

5.3.5 Decrypt data

The security processing services are requested to decrypt data via Sec-SecureDataContent-Extraction.request. The information elements used by the security processing services for this primitive are specified in 7.2.17. These information elements include a CMH in *Key and Certificate* state. The security services take as input encrypted data that is a valid encoding of a 1609Dot2Data as defined in Clause 6.

The security processing services return the result of the request via Sec-SecureDataContentExtraction-.confirm. If all operations succeed, the result shall be the type of the data, the plaintext contents of the data and additional information elements that were included by the sending security services. If any of the operations fail, the result shall be an appropriate error code.

The security processing services may reject a request to decrypt data if the number of requested recipients is greater than 6.

While performing operations in response to Sec-SecureDataContentExtraction.request, the security processing services shall also extract and return the information elements specified in the specification of Sec-SecureDataContentExtraction.confirm.

The decryption process is illustrated in Figure 18, where the dashed lines indicate entities and processing that are defined in this standard. See D.2 for a sample process flow for receiving secure data.

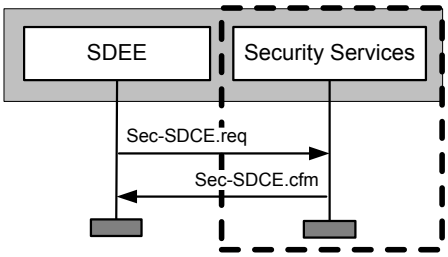


Figure 18—Process flow for decrypting data

5.3.6 Verify signed data

The security processing services are requested to verify signed data via Sec-SignedData-Verification.request. The information elements used by the security processing services for this primitive are specified in 7.2.19. The security services take as input signed data that is a valid encoding of a SignedData as defined in Clause 6.

The security processing services return the result of the request (either an indication of success with copies of relevant certificates or an indication of the reasons for failure) via Sec-SignedDataVerification.confirm.

The security processing services shall reject signed data as failing verification if the data and associated certificates fail the validity and consistency tests specified in 5.5. If requested, WAVE security services shall reject signed data as failing verification if the signed data fails any of the relevance tests specified in 5.5.5. The security processing services may reject signed data as failing verification if the full certificate chain from the certificate to a trust anchor, as described in 5.5.2, is longer than some threshold specified by the invoking SDEE.

While performing operations in response to Sec-SignedDataVerification.request the security processing services shall also extract information about the certificates associated with the message as described in the specification of Sec-SignedDataVerification.request. The security services return this information via Sec-SignedDataVerification.confirm.

The verification process is illustrated in Figure 19, where the dashed lines indicate entities and processing that are defined in this standard. Primitive names in the figure are abbreviated for compactness. See D.2 for a sample process flow for receiving secure data.

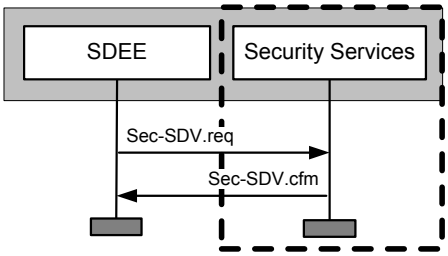


Figure 19—Process flow for verifying data

5.4 Signed WSAs

5.4.1 Sign WSA

The security processing services are requested to sign a WSA via WME-Sec-SignedWsa.request. The information elements that the security processing services require the WME to provide are specified in 7.3.2. The CMH in *Key and Certificate* state that is required to generate the signature shall be obtained from the PSSME via PSSME-Sec-CryptomaterialHandle.request. A full process flow covering the inclusion of a secure provider service in a signed WSA is illustrated in D.4.

The security processing services return the result of the request (either an octet string containing the signed WSA, or an error code) to the WME via in a WME-Sec-SignedWsa.confirm. On success, the octet string returned shall be a valid encoding of a 1609Dot2Data as defined in Clause 6. WME behavior following reception of WME-Sec-SignedWsa.confirm is specified in IEEE Std 1609.3.

WAVE security services shall reject a request to sign a WSA if it does not have access to a certificate that is currently valid according to the criteria specified in 5.5, if the CMH is not valid according to the criteria specified in 5.2.3.4, or if the certificate referenced by the CMH is not an explicit certificate (see 5.5.2). WAVE security services may reject a request to sign data if the full certificate chain from the certificate to a trust anchor, as described in 5.5.2, is longer than 5 certificates. WAVE security services may reject a request to sign a WSA if it contains more than 32 ServiceInfo entries (see IEEE Std 1609.3 for the definition of WSAs and ServiceInfo entries).

The WSA signing process is illustrated in Figure 20 where the dashed lines indicate entities and processing that are defined in this standard. Primitive names in the figure are abbreviated for compactness. Sample process flows to obtain and use certificates for WSA signing are described in D.3.2 and D.4.

NOTE—Any changes made to the signed WSA before sending it invalidate the digital signature, resulting in the recipient rejecting the signed WSA.

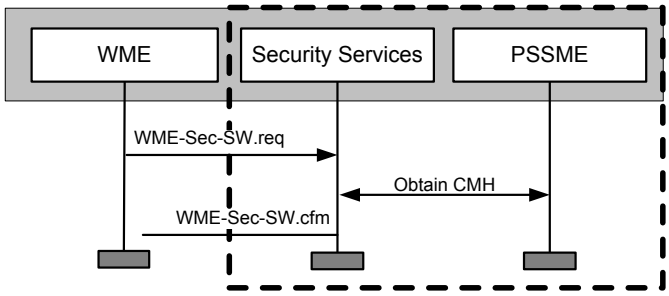


Figure 20—Process flow to sign WSA

5.4.2 Verify signed WSA

The security processing services are requested to verify a signed WSA via WME-Sec-SignedWsa-Verification.request[14]. The information elements used by the security processing services for this primitive are specified in 7.3.4.

The security processing services shall then invoke WME-Sec-SignedWsaVerification.confirm to return the result to the SDEE. This result shall be either a failure code or a notification of success along with information extracted from the signed WSA.

As described in the specification of WME-Sec-SignedWsaVerification.confirm, if the WSA is valid the security processing services shall store any new certificate information obtained while processing the data with the CME, for example via CME-AddCertificate.request.

WAVE security services shall reject a signed WSA as failing verification if the data and associated tests fail the validity tests specified in 5.5. WAVE security services may reject a signed WSA if the full certificate chain from the certificate to a trust anchor, as described in 5.5.2, is longer than 5 certificates. WAVE security services may reject a signed WSA if it contains more than 32 ServiceInfo entries (see IEEE Std 1609.3 for the definition of WSAs and ServiceInfo entries). WAVE security services may reject a signed WSA if it is not signed by an explicit certificate (see 5.5.2).

The signed WSA verification process is illustrated in Figure 21, where the heavy dashed lines indicate entities and processing defined in this standard. Primitive names in the figure are abbreviated for compactness.

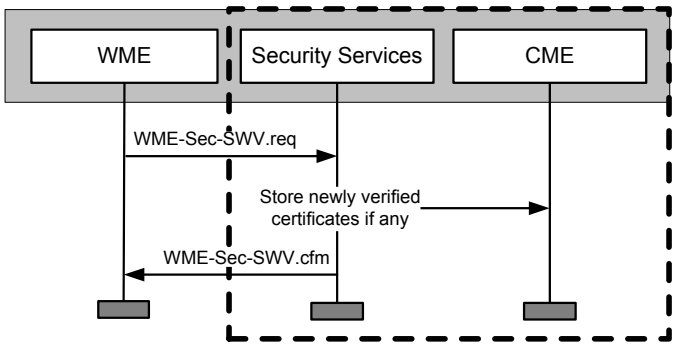


Figure 21—Process flow for validating a signed WSA

[14] This is referred to in IEEE Std 1609.3-2010 as WaveSecurityServices-SignedWsaValidation.request, and the corresponding confirm primitive is referred to as WaveSecurityServices-SignedWsaValidation.confirm.

5.5 Validity of signed communications

5.5.1 General

A signed communication for a secure communications entity is valid if all of the following hold:

- The communication is correctly formed using the data structures of Clause 6.
- There is a certificate chain that leads from the signing certificate to a known trust anchor, such that:
- All of the certificates in the chain are correctly formed using the data structures of Clause 6.
- The certificate chain is internally consistent.
- None of the certificates in the chain have been revoked.
- All certificates in the chain are cryptographically verified with the appropriate public keys.
- (Optional) The number of certificates in the chain, from end-entity certificate to root certificate inclusive, is less than some entity-specific limit[15].
- The communication is consistent with the signing certificate.
- The communication cryptographically verifies with the appropriate public keys.
- The communication meets other appropriate criteria (for example, to protect against data that are replays or are not relevant).

This clause specifies the criteria that determine whether a message is valid:

- Clause 5.5.2.1 specifies a validly constructed certificate chain.
- Clause 5.5.2.2 specifies cryptographic verification of certificates and communications.
- Clause 5.5.3 specifies consistency of data with a certificate or two certificates with each other.
- Clause 5.5.4 specifies expiry and revocation.
- Clause 5.5.5 specifies relevance checking.

Table 1 summarizes the steps to be used when validating a signed communication. The order of processing is fixed only where noted. If any of the steps fail, the signed communication shall be rejected. Clause 7 specifies in detail processing that may be carried out to check signed communications against these criteria.

[15] Whether there is such a limit, and if so what value it takes, is out of scope for this standard. If there is such a limit, it would be appropriate for it to be specified as part of the specification of the signed communications entity.

Table 1—Steps for validating a signed communication

Step	Description	Flow specification	Processing specification	Notes
a)	Construct the certificate chain	5.5.2	Sec-SignedData-Verification.request or WME-Sec-SignedWsa-Verification.request, CME-Function-ConstructCertificateChain	Process is illustrated in D.6.2. If the end-entity certificate is already known and no revocation information has been received since it was last used to verify a message, this step and steps b), c) and f) can be skipped.
b)	Check that the certificates in the chain were not revoked	5.5.4	CME-CertificateInfo.request	Cannot be completed before a) is completed.
c)	Check the certificate chain for consistency	5.5.3	Sec-SignedData-Verification.request or WME-Sec-SignedWsa-Verification.request, Sec-Function-CheckCertificateChain-Consistency	Cannot be completed before a) is completed.
d)	Check that the message is consistent with the certificate	5.5.3	Sec-SignedData-Verification.request or WME-Sec-SignedWsa-Verification.request	If the signing certificate used inherited permissions, depends on a). Otherwise, no dependencies.
e)	Check that the message is internally consistent and consistent with transport layers	5.5.3.2.3, 5.5.3.2.4	Sec-SignedData-Verification.request or WME-Sec-SignedWsa-Verification.request	If the signing certificate used inherited permissions, depends on a). Otherwise, no dependencies.
f)	Cryptographically verify the signature on explicit certificates	5.5.2.2	Sec-SignedData-Verification.request or WME-Sec-SignedWsa-Verification.request, Sec-Function-VerifyChainAndSignature	Depends on a).
g)	Cryptographically verify the signature on the communication	5.5.2.2	Sec-SignedData-Verification.request or WME-Sec-SignedWsa-Verification.request, Sec-Function-VerifyChainAndSignature	Includes cryptographic verification of implicit certificates. Depends on a) if there were implicit certificates.
h)	Check relevance, check for replay	5.5.5	Sec-SignedData-Verification.request, CME-Sec-ReplayDetection.request	Optional; only for signed data exchange.
i)	Check that the WSA is fresh	5.4.2	PSSME-OutOfOrder-Detection.request	Only for signed WSAs.

5.5.2 Certificate chains

5.5.2.1 General

Received signed communications are verified using the *end-entity certificate* associated with the signed data (an end-entity certificate is a certificate used for any purpose other than issuing certificates).

A certificate contains, explicitly or otherwise:

— At least one public key for a public key cryptosystem, belonging to the certificate holder.

— A list of the permissions associated with that public key.

— An identifier for the issuer (who may be the same as the certificate holder or may be different).

— A cryptographic demonstration that the issuer authorized the linkage between the public key and the permissions.

The entity that uses the private key corresponding to the public key is referred to as the *certificate holder*. A certificate used to validate application or management data is an *end-entity certificate*. A certificate used to validate another certificate is a *CA certificate*. If the public key is explicitly given in the certificate, the certificate is an *explicit certificate*. If the public key is not explicitly given in the certificate, but is obtained by performing additional processing, the certificate is an *implicit certificate*. The difference between implicit and explicit certificates is illustrated in Figure 22.

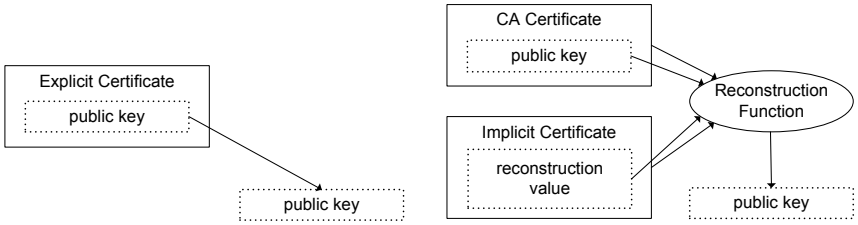


Figure 22—Implicit and explicit certificates

A necessary (but not sufficient) condition for a receiver to trust a received certificate is that the receiver can construct a *certificate chain* from the certificate to a *trust anchor*. A certificate chain is an ordered set of certificates such that each certificate in the chain is the *issuing certificate* for the subsequent one. One certificate is the issuing certificate for another if the certificate holder of the first certificate used the private key of the first certificate to create the final form of the second certificate, either by signing it (in the case of an explicit certificate) or by creating cryptomaterial to be used in recovering the public key (in the case of an implicit certificate). To enable construction of certificate chains, IEEE 1609.2 certificates contain an identifier for the signer, referred to as the `signer_id` (see 6.3.3). An algorithm for the construction of certificate chains is specified in 7.8.2 and illustrated in D.6.2.

The counterpart of an issuing certificate is a *subordinate certificate*; each certificate in a chain, other than the end-entity certificate and the trust anchor, is the issuer of the certificate after it and the subordinate of the certificate before it.[16]

A trust anchor is any certificate that is known to be trustworthy by itself; in other words, not by reference to any other certificate. The trust anchor may be a *root certificate*, which is a certificate that is verified with

[16] Technically, the certificate holder is the issuer of the certificate after it in the chain, rather than the certificate itself being the issuer. For conciseness, this standard refers to the certificate as the issuer.

the public key included directly in the certificate. Alternatively, the trust anchor may be any other CA certificate that is known to be trustworthy.

The data structures given in 6.2 allow the sender to indicate which certificate is to be used to verify the signed communication. The sender may include a hash of the signing certificate, the signing certificate itself, or a certificate chain including the end-entity certificate and a series of issuing certificates up to, but not including, the root.

In constructing the certificate chain, the receiver may use the certificates that were sent by the sender or locally cached copies of certificates. Local copies of certificates are managed by the CME. An algorithm for constructing a certificate chain using certificates from a signed communication and from local storage is given in 7.8.2 and illustrated in Figure D.19.

A receiver shall reject a communication signed by a certificate unless the chain from the certificate to its trust anchor meets the following conditions:

— The trust anchor is an explicit certificate.

— All explicit certificates are issued by explicit certificates.[17]

A specification of a secure communications entity may use the IEEE 1609.2 security profile (introduced in 4.3.2 and specified fully in Annex B) to specify a maximum length for the full certificate chain. Certificate chains greater than the specified maximum length are not considered valid in the context of that secure communications entity. The length of a full certificate chain is the number of certificates in the chain including the root certificate. Therefore, the minimum length of a full certificate chain is 2: the end-entity certificate and the root certificate.

The Protocol Implementation Conformance Statement (PICS) proforma given in Annex A allows the vendor of a general-purpose implementation of WAVE Security Services to state the maximum length of certificate chain that the implementation supports.

NOTE 1—An implementation only needs to construct a chain to a trust anchor, not to a root, to determine that it is valid; however, this does not allow an unambiguous definition of certificate chain length, as a particular non-root certificate may be considered a trust anchor by one instance of WAVE Security Services and not by another. The only unambiguous definition of length, therefore, is a length that includes the root.

NOTE 2—Secure communications entities may have a maximum certificate chain length, but may also give guidance to developers that an appropriate certificate chain length is less than this maximum. For example, since long certificate chains increase packet size and therefore channel congestion and error rates, it is appropriate for the specification of the secure communications entity to give guidance that short (relative to the maximum) certificate chains should be used. This is particularly important for secure communications entities that transmit frequently.

5.5.2.2 Cryptographic verification of certificate chains

The CME shall only accept an explicit certificate as a trust anchor.

Each subordinate explicit certificate in a chain is verified by ensuring that the public key in the issuing certificate can be used to verify the signature on the subordinate certificate. See Figure 23 for an illustration.

If the end-entity certificate is an explicit certificate, the signature on the communication is verified by ensuring that the public key in the end-entity certificate can be used to verify the signature on the communication. See Figure 23 for an illustration.

[17] This implies that if the certificate chain contains both implicit certificates and explicit certificates, all of the implicit certificates are at the end of the chain and all of the explicit certificates are at the start of the chain.

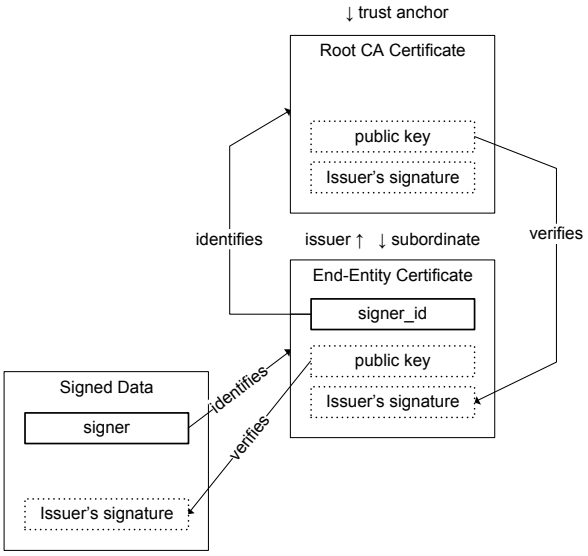


Figure 23—Cryptographic verification with explicit certificates

If the end-entity certificate is an implicit certificate, the signature on the communication is verified by ensuring that the public key that is obtained from the end-entity certificate using the methods of 5.8.6 can be used to verify the signature on the communication. Figure 24 illustrates that the verification process involves the end-entity's implicit certificate and the end-entity's issuing certificate's public key.

If any CA certificates are implicit certificates, they are (implicitly) verified by verifying the signature on the communication.

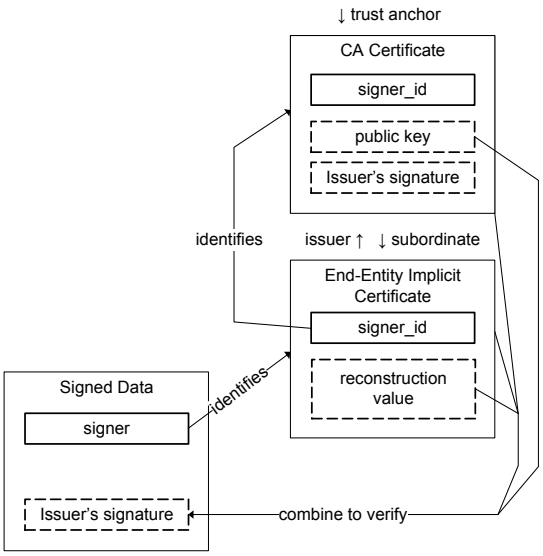


Figure 24—Length-2 certificate chain with implicit certificates and a non-root CA as trust anchor

For further illustrations, see D.6.1.

5.5.3 Permissions and consistency of permissions

5.5.3.1 General

A certificate communicates the permissions of its holder, namely:

- Geographic permissions: the region within which the certificate is valid, if relevant.
- Validity period: the time period within which the certificate is valid.
- Operational permissions: the activities that the holder is allowed to perform.
- For a CA certificate, the type(s) of certificate that it is permitted to issue.

Operational permissions are expressed using the following variables:

- The PSID data type defined in IEEE Std 1609.3.
- The SSPs. These permissions are defined outside this standard but subject to the constraints of 6.3.25.
- For WSA certificates, the maximum priority at which the WME shall permit a serivce to be advertised.

For a CA, the operational permissions contain PSIDs only, not SSPs (see 6.3.8, 6.3.20, 6.3.21). For a CA certificate, the operational permissions in the CA certificate specify the operational permissions that the CA is permitted to include in subordinate certificates issued by that CA certificate. For an enrolment certificate (see 5.6.1), the operational permissions are the list of permissions that the certificate holder is permitted to request be included in its operational certificates.

NOTE— For an end-entity certificate holder, the permitted activities corresponding to a given PSID and SSP are defined by the organization that reserves the PSID. IEEE Std 1609.12 specifies a list of reserved PSIDs and the organizations that have reserved them.

5.5.3.2 Consistency between signed communications and signing certificates

5.5.3.2.1 General

A signed communication (signed data or WSA) is consistent with the signing certificate if:

- The communication originated from within the geographic region indicated in the certificate.
- The communication was created within the validity period of the certificate.
- The communication’s expiry time is within the validity period of the certificate.
- The information within the communication is consistent with the operational permissions given in the certificate where consistency is determined using rules specified by the organization that reserved the PSID.
- The public key indicated by the certificate can be used to cryptographically verify the signature on the communication.

A signed communication that is inconsistent with its signing certificate is invalid.

The following clauses define consistency of permissions with signed data (5.5.3.2.2) and with WSAs (5.5.3.2.5).

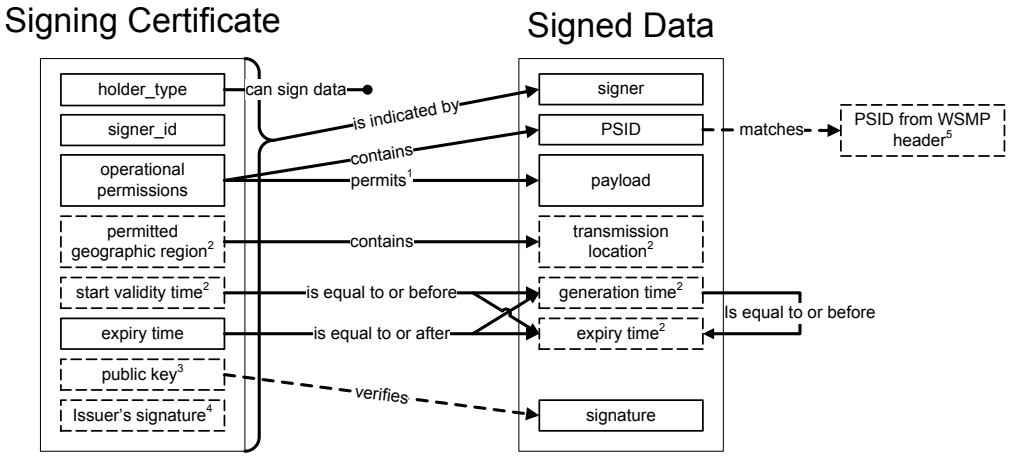
5.5.3.2.2 Signed data

Signed data is encoded using the data structures of Clause 6. The encoding contains a PSID. Signed data is consistent with the permissions in the signing certificate if the PSID information element in the signed data is one of the PSIDs in the operational permissions of the certificate.

A certificate also contains a SSP field for each PSID. The security processing services shall make the SSP available to an SDEE that verifies signed data.

NOTE—The SSP enables the SDEE to perform additional validity checks on the data payload. These checks are out of scope for this standard. See E.5.1 and E.5.2 for more discussion.

Figure 25 illustrates the process of checking that signed data is consistent with the associated signing certificate, and additionally captures the process of checking that the signed data is internally consistent as defined in Clause 6. The boxes within the data and certificate identify the information elements that are conveyed by those data structures. For clarity, only the relevant fields within the data structures are shown. The generation time, expiry time, and generation location may be included within the fields added by the security processing or may be a part of the payload. See Clause 6 for a specification of the exact fields in the certificate and the signed data that encode this information. See Clause 7 for specification of processing that correctly performs these checks.



NOTES:
1. Determined using the PSID and SSP. The process to determine whether the operational permissions permit the message payload is specified by the organization reserving the PSID and is out of scope for this standard.
2. Optional. Inclusion of this data is as determined by the organization reserving the PSID. This data may be contained in the payload or within the security header fields.
3. For implicit certificates, the public key is derived rather than explicitly stated within the certificate.
4. Not included in an implicit certificate.
5. For WSMP only.

Figure 25—Checking consistency of permissions between signed data and its signing certificate, and within signed data

5.5.3.2.3 Consistency between signed data and transport layers

If the signed data was received within the WAVE Short Message Protocol (WSMP) described in IEEE Std 1609.3, it shall be considered invalid unless the PSID in the signed data structure is the same as the WSMP PSID header field. This is also illustrated in Figure 25. The Sec-SignedDataVerification.request primitive specified in Clause 7 supports carrying out this check within the security services.

5.5.3.2.4 Consistency within signed data

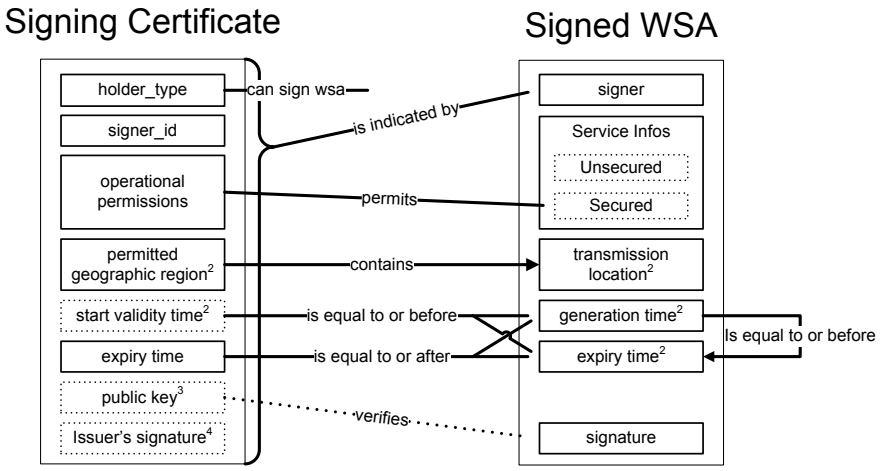
If the signed data contains both a generation time and an expiry time, it shall be considered invalid if the expiry time is after the generation time. This is also illustrated in Figure 25.

5.5.3.2.5 Signed WSA

The process of checking consistency of operational permissions within a signed WSA is specified in full in 7.3.4 and illustrated in D.4.

A signed WSA may contain both secured and unsecured ServiceInfo fields. The security services determine the validity of the secure provider services only.

Figure 26 illustrates the process overall of checking that a signed WSA is consistent with the associated signing certificate, and additionally captures the process of checking that a signed WSA is internally consistent as defined in Clause 6. The boxes within the data and certificate identify the information elements that are conveyed by those data structures. For clarity, only the relevant fields within the data structures are shown. See Clause 6 for a specification of the exact fields in the certificate and the signed WSA that encode this information. See Clause 7 for specification of the exact processing.



NOTES:
1. For implicit certificates, the public key is derived rather than explicitly stated within the certificate.
2. Not included in an implicit certificate.

Figure 26—Checking consistency of permissions between a signed WSA and its signing certificate, and within the signed WSA

A signed WSA is consistent with the operational permissions in its signing certificate if each of the ServiceInfo entries in the WSA is consistent with the corresponding permissions entry in the signing certificate. The ServiceInfo is defined in IEEE Std 1609.3. See later in this clause for the specification of how to determine which permissions entry in the certificate corresponds to which ServiceInfo, and how to determine which entries in the WSA are secured or unsecured.

A ServiceInfo entry consists of information elements including:

— A PSID.

— The priority at which the service is offered.

A permissions entry in a WSA certificate consists of the following information elements:

— A PSID.

— A maximum priority for the service.

— Optionally, SSPs, which are interpreted by the receiving user service rather than by the security services.

A ServiceInfo entry is consistent with a permissions entry if:

— The PSID entries are identical.

— The priority in the ServiceInfo is less than or equal to the maximum priority in the permissions.

Figure 27 illustrates the process of checking that a WSA certificate permissions entry is consistent with a particular ServiceInfo.

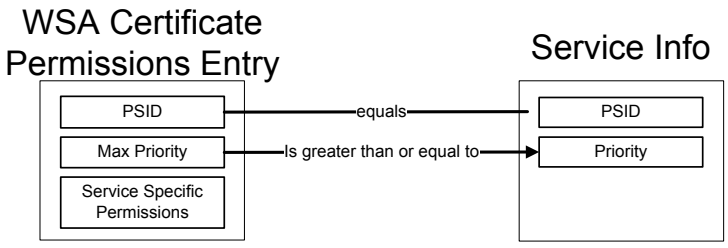


Figure 27—Checking consistency between a ServiceInfo in a signed WSA and the permissions in its signing certificate

The permissions in a WSA signing certificate are mapped to the ServiceInfos in the WSA by means of the `permission_indices` field in the signed WSA. See Clause 6 for a specification of the exact fields in the certificate and the signed data that encode this information. The `permission_indices` field is an array of integers the same length as the array of ServiceInfos in the WSA. For each ServiceInfo, the corresponding entry in the `permission_indices` field is either 0, indicating that the ServiceInfo describes an unsecured service, or a non-zero integer, indicating the relevant field in the permissions in the certificate.

Figure 28 illustrates the process of mapping from the ServiceInfos in the WSA to the permissions in the certificate. See Clause 7 for specification of the exact processing. An example is given in D.3.2. The use of the SSP by the user services is discussed in E.6.3.

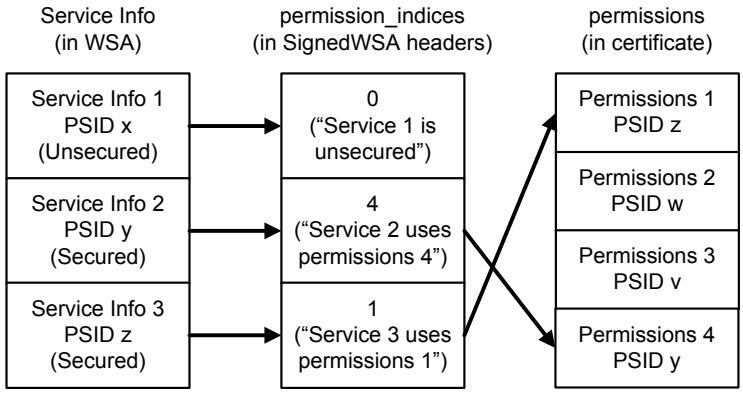


Figure 28—Mapping from the list of ServiceInfos in a WSA to the list of permissions in a certificate via permission_indices

5.5.3.3 Consistency between subordinate certificates and issuing certificates

A subordinate certificate is consistent with its issuing certificate if:

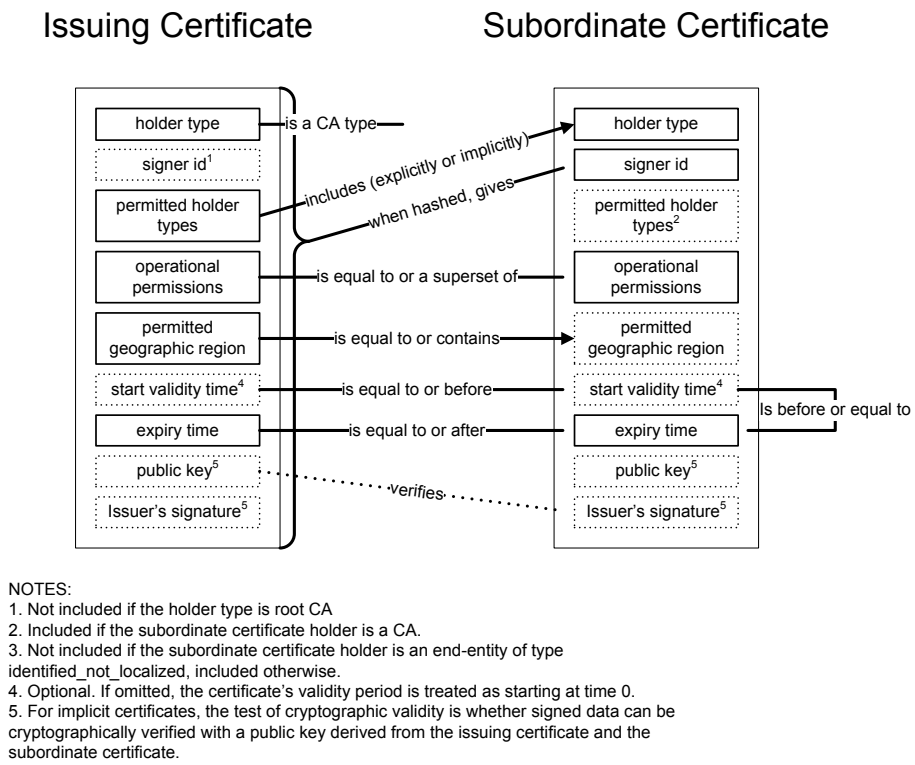
- The subordinate certificate's validity region is wholly contained in the issuing certificate's validity region.
- The subordinate certificate's validity period is within the issuing certificate's validity period.
- The subordinate certificate's operational permissions are a subset of the issuing certificate's operational permissions.
- The issuing certificate signed the subordinate certificate if it is an explicit certificate or the cryptographic material in the certificates can be used to verify a signature if it is an implicit certificate.
- The issuing certificate is allowed to issue a certificate of the type of the subordinate certificate, as determined by the `permitted_holder_types` field in the certificate (see 6.3.8, 6.3.20, 6.3.21).

A signed communication is invalid if any subordinate certificate in the chain is inconsistent with its issuing certificate.

If any subordinate certificate is inconsistent with its issuing certificate, the subordinate certificate in the pair, and any certificates that chain back to it, are invalid.

If any subordinate certificate is inconsistent with its issuing certificate but has a signature that verifies with the issuing certificate's public key, the security services shall treat the issuing certificate and all certificates issued by it as invalid.

Figure 29 illustrates the process of checking that a subordinate certificate is consistent with its issuing certificate, and additionally captures the process of checking that a certificate is internally consistent as defined in Clause 6. For clarity, only the relevant fields within the data structures are shown. See Clause 6 for a specification of the exact fields in the certificate and the signed data that encode this information. See Clause 7 for a specification of the exact processing.



NOTES:
1. Not included if the holder type is root CA
2. Included if the subordinate certificate holder is a CA.
3. Not included if the subordinate certificate holder is an end-entity of type identified_not_localized, included otherwise.
4. Optional. If omitted, the certificate's validity period is treated as starting at time 0.
5. For implicit certificates, the test of cryptographic validity is whether signed data can be cryptographically verified with a public key derived from the issuing certificate and the subordinate certificate.

Figure 29—Checking consistency of permissions between an issuing and a subordinate certificate, and within the subordinate certificate

5.5.3.4 Permission encoding: inherited permissions

The data structures of Clause 6 provide two methods for encoding a certificate's geographic or operational permissions. These permissions shall either be encoded explicitly within the certificate or omitted from the certificate and *inherited* by reference to the issuing certificate.

Figure 30 contains an example of the use of inherited permissions. The string "`from_issuer`" indicates that permissions are inherited. The end-entity certificate has the same operational permissions as CA Certificate 2 and the same geographical permissions as CA Certificate 1. CA Certificate 1 has the same operational permissions as CA Certificate 2.

A full specification of the encoding is given in 6.3. The CME provides the inherited permissions of a certificate to callers via CME-CertificateInfo.request.

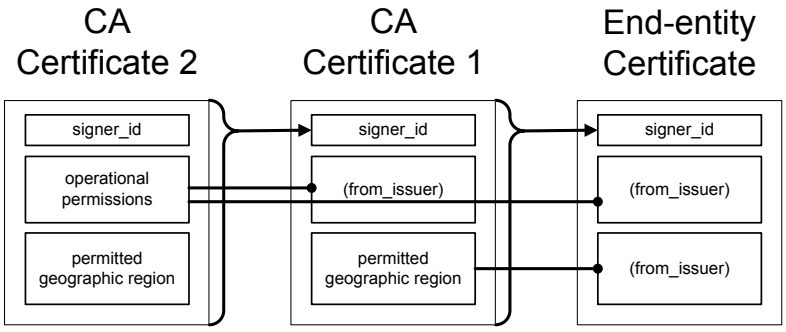


Figure 30—Inherited permissions

NOTE—The issuing certificate is assumed to be known to a verifier as the verifier needs the issuing certificate to verify the subordinate certificate. Therefore, the use of inherited permissions does not change the requirements for what information is available to the verifier.

5.5.4 Certificate validity

5.5.4.1 Certificate lifetime

A certificate is valid at a given time if it has not expired, is past its start validity time, and has not been revoked.

A certificate contains an expiry date. Communications are invalid if they have a generation time or expiry time after the expiry time of their signing certificate. If the data has no expiry time, it is considered to expire at the expiry time of the certificate.

A certificate may contain a start date. Communications are invalid if they have a generation time or expiry time before the start time of their signing certificate. If the data has no generation time, it is considered to have been generated at the time of the certificate. If a certificate has no start time, it is considered to have been generated at the 1609.2 epoch, 00:00:00 UTC, 1 January, 2004.

5.5.4.2 Certificate revocation

5.5.4.3 Revoked certificates

A certificate is said to be revoked if an authorized entity distributes an authenticated message stating that that certificate is known not to be trustworthy. Such an authenticated message is known as a Certificate Revocation List (CRL). If a certificate is revoked, all communications signed by that certificate and received after the issue date of the revocation list shall be considered invalid, even if their stated generation time is before the issue date of the revocation list.

If a CA certificate is revoked, the security services shall also consider all certificates issued by that CA and first received after the issue date of the CRL to be revoked, even if their stated generation time if before the issue date of the revocation list. This applies to any certificate that chains back to the revoked CA.

A validly signed CRL indicates that a certificate *c* was revoked if:

— The CRL refers to certificates issued by *c*'s issuing certificate. The CRL contains a `ca_id` field indicating which issuing certificate the CRL references. The `ca_id` is a hash of the issuing certificate.

— The CRL series value in the CRL also appears in the certificate *c*. This value allows a CA to segment its certificates into different populations that appear on different CRLs, to keep down the size of individual CRLs.

— The hash of the certificate appears in the entries on the CRL.

Figure 31 illustrates the process of checking whether a given certificate appears on a CRL. For clarity, only the relevant fields within the data structures are shown. See Clause 6 for a specification of the exact fields in the certificate and the signed data that encode this information. See Clause 7 for a specification of the exact processing. The process of determining that a CRL is validly signed is specified in 5.6.4.2.

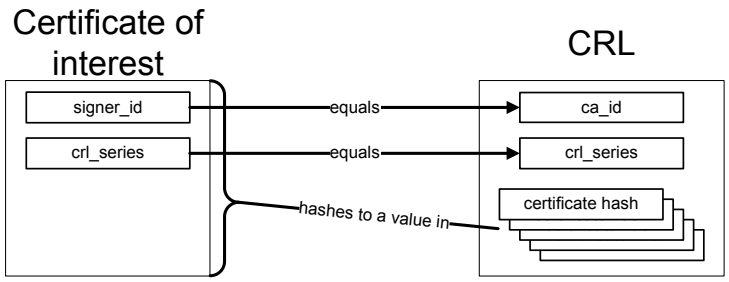


Figure 31—Checking whether a certificate appears on a CRL

5.5.4.4 Dubious certificates

A dubious certificate is a certificate for which an instance of the CME cannot determine whether or not it is revoked, because the CME has not been provided with an up-to-date CRL that would revoke that certificate if necessary.

NOTE—The CME can determine whether or not an up-to-date CRL has been received for any certificate, because the `signer_id` and `crl_series` fields in the certificate are equal to the `ca_id` and `crl_series` fields in the CRL. The revocation information for a certificate is not up to date if either the CME has no information about a CRL with that `ca_id` and `crl_series` value, or if the information about a CRL with that `ca_id` and `crl_series` value was provided via CME-AddCrlInfo.request and all values for that CRL series of *Next Crl* are in the past.

The standard provides the following mechanisms to handle the case where the security processing services determine that a communication signed with a dubious certificate would be valid if the certificate was known not to be revoked (i.e., it passes all checks except the revocation check and its revocation status is undetermined):

— If the security processing services are passed the parameter Overdue CRL Tolerance via Sec-SignedDataVerification.request, and if the CRL needed to check revocation was issued more than Overdue CRL Tolerance in the past and has not been received, the security processing services shall reject the signed communication.

— If the signed communication is signed data, the security processing services shall inform the invoking entity of how much the relevant CRL is overdue via Sec-SignedDataVerification.confirm.

— If the signed communication is a signed WSA, the security processing services shall inform the WME of how much the relevant CRL is overdue via WME-Sec-SignedWsaVerification.confirm.

Each certificate in a chain, except the root, belongs to a CRL series and has a next CRL time. For a chain with more than one certificate, the considerations above apply to each certificate in the chain.

5.5.5 Relevance and replay tests

A receiving entity may request that the security services carry out the following tests on received signed data to determine whether or not the data is relevant:

— **Correctness tests**: Is the signature generation time in the future?

— **Freshness tests**: Is the signature generation time sufficiently recent? Has the signed data expired?

— **Location tests**: Is the generation location sufficiently close to the receiver's location?

— **PSID tests**: Is the PSID in the signed data the one expected?

— **Replay tests**: Is the data a duplicate of a data already received?

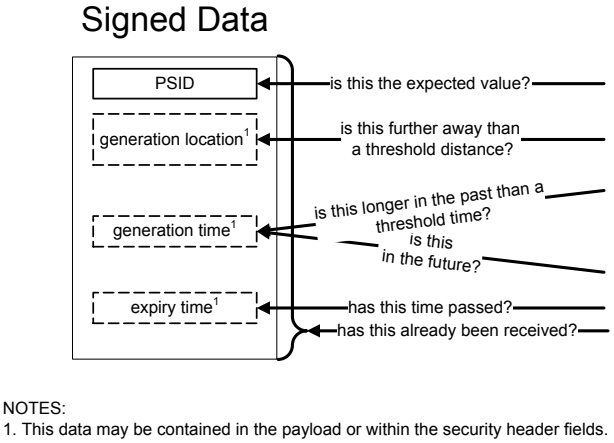
The security services shall only use information for relevance and replay tests if that information was authenticated and integrity-checked, which in the context of this standard means that information used for relevance and replay checks shall be obtained from a validly signed PDU. If the information necessary for relevance tests (generation time, generation location, and/or expiry time) is not included in the signed PDU, the security processing services shall reject the communication as failing the relevance tests. The data structures in Clause 6 allow the information necessary to be transported either in the payload of the signed data or in fields added by the sender-side security processing and included in the calculation of the signature.

The relevance tests to be carried out are application-specific. A valid signed PDU shall include the information needed to carry out the relevance tests applicable to that application, and may omit information that is not used for the applicable relevance tests.

Correctness, freshness, and location tests are all parameterized by a tolerance with the relevant value from the data tested to determine whether it differs from the locally observed time or location by more than the specified threshold.

The choice of relevance tests and the tolerance parameters for those tests depends on the receiver's requirements. The IEEE 1609.2 security profile described in Annex B allows a specification of an SDEE to identify which replay and relevance tests are carried out and how the relevant information is transported.

Figure 32 illustrates the process of carrying out replay and relevance tests. The relevance and replay tests to be carried out are specified as inputs to Sec-SignedDataVerification.request.



NOTES:
1. This data may be contained in the payload or within the security header fields.

Figure 32—Replay and relevance tests

5.5.6 Local estimates of time and location

WAVE Security Services have access to the estimate of time on the local device. This estimate is not simply the current best-guess value for current time, but the probability distribution function (PDF) specifying the distribution of time values. The use of this PDF is described in 7.2.19.4.1 step j)1) for signed data.

NOTE 1—This standard does not specify minimum accuracy requirements for the local time probability distribution function. If the local time service is used to obtain generation times to be included in the data structures of this standard, this standard does not specify minimum performance requirements, for example minimum time between request and confirm primitives or minimum time between the time indicated in the generation timestamp and the time the resulting data structure is available to the entity that invoked the security services.

WAVE Security Services have access to an estimate of current 3D location on the local device.

NOTE 2—This standard does not specify minimum accuracy requirements for the estimate of current 3D location.

For further discussion, see E.3.

5.6 Processing for security management

5.6.1 Certificate request

5.6.1.1 Processing

A certificate request is a request to a CA to associate a particular signing public key with a particular set of permissions within a certificate. The security processing services support the generation of certificate requests and processing of certificate responses via the primitives described in this section. A process flow illustrating the use of these primitives is given in D.3.This standard does not specify transport mechanism to be used for communications with the CA.

The security processing services are requested to create a signed, encrypted certificate request via Sec-CertificateRequest.request. The information elements used by the security processing services to generate a certificate request are specified in 7.2.23 and include the following:

— The permissions requested by the secure communications entity. This includes operational permissions, geographic permissions and requested duration.

— The public verification key to be included in the certificate request. This shall correspond to a CMH in the *Key Pair Only* state.

— The public encryption key to be included in the certificate request, if any. This shall correspond to a CMH in the *Key Pair Only* state.

— A CMH that references a private key suitable for use in signing the certificate request. A private key is suitable for signing a certificate request if it fulfils either of the following conditions:

— It is the private key corresponding to the public key in the certificate request.

— It corresponds to a certificate whose permissions include authorization to sign the certificate request.

— An indication of a public key encryption algorithm to be used to encrypt the response to the certificate request. The security services shall generate a fresh response encryption keypair for each certificate request, The public response encryption key shall be included in the certificate request.

The security services return the result of the request via Sec-CertificateRequest.confirm. The result is either the encoded request or an error code. The encoded request is a 1609Dot2Data of type `encrypted`, containing a ToBeEncrypted of type `certificate_request` which, itself, contains a signed CertificateRequest (see Clause 6 for a specification of these data structures). In the case of success, Sec-CertificateRequest.confirm shall also return other information elements including a ten-byte hash of the plaintext of the certificate request. The information elements returned by the security processing services are specified in 7.2.25.

The security services shall reject a request to sign a certificate request, and return an error code, if any of the certificates have expired or been revoked, or if the use of the CMHs provided to Sec-Certificate-Request.request would result in a certificate request that is invalid by the criteria of 5.6.1.2.

The process flow for requesting certificate request generation by the security processing services is illustrated in Figure 33, where the dashed lines indicate entities and processing that are defined in this standard. Primitive names in the figure are abbreviated for compactness.

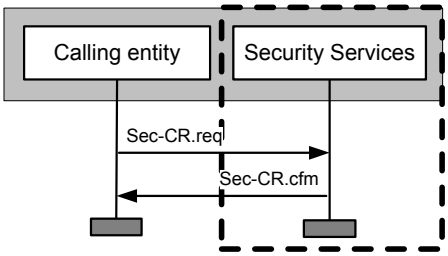


Figure 33—Process flow for requesting certificate request generation

5.6.1.2 Validity of certificate requests

A certificate request shall be considered invalid unless it is signed either by the public key within the certificate request, in which case the request is referred to as "self-signed," or by an enrolment certificate. See Clause 6 for a specification of Enrolment certificates.

NOTE—The term "enrolment certificate" is harmonized with the terminology introduced by ETSI [B6].

A self-signed certificate request is valid if:

— The signature on the certificate request can be verified with the verification public key in the certificate request.

A certificate request signed by a enrolment certificate is valid if:

— It it possible to construct a certificate chain that leads from the signing enrolment certificate to a known trust anchor by the methods of 5.5.2.1, such that:

— All of the certificates in the chain are correctly formed using the data structures of Clause 6.

— The certificate chain is internally consistent as specified in 5.5.3.

— None of the certificates in the chain have been revoked as specified in 5.5.4.

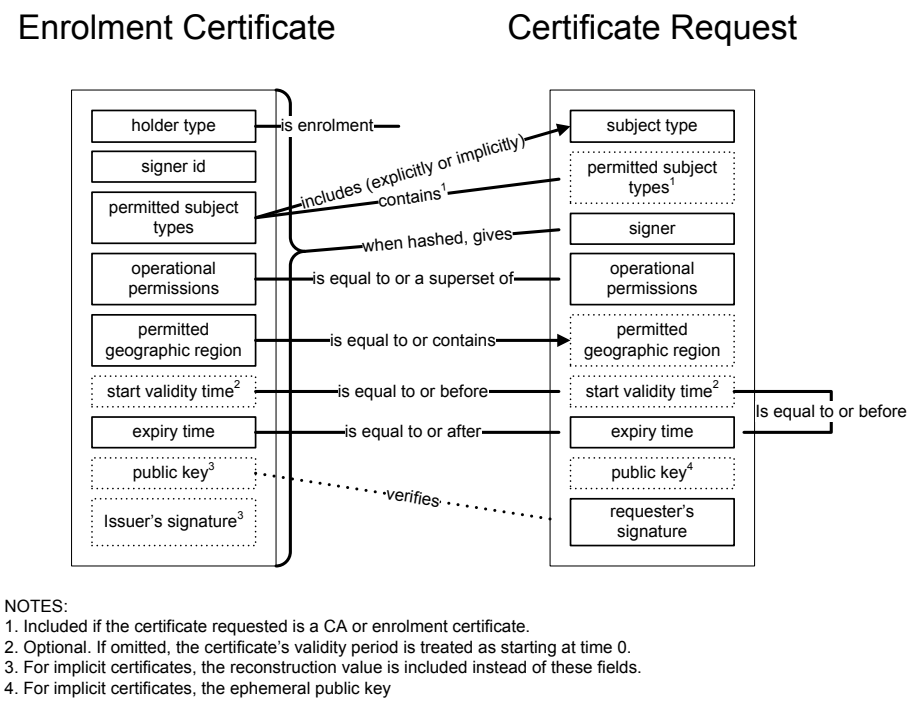
— All certificates in the chain can be cryptographically verified with the appropriate public keys as specified in 5.5.2.2.

— The certificate request is consistent with the signing certificate as specified in this clause.

— The certificate request can be cryptographically verified with the appropriate public keys as specified in 5.5.2.2.

Figure 34 illustrates the process of checking that a certificate request is consistent with the enrolment certificate that signed it, and additionally captures the process of checking that the certificate request is internally consistent as defined in Clause 6. For clarity, only the relevant fields within the data structures are shown. See Clause 6 for a detailed specification of the fields in the certificate and the signed data that encode this information. See Clause 7 for a detailed specification of the processing.

Consistency of an enrolment certificate with its issuing certificate is determined by the criteria of 5.5.3.3.



NOTES:
1. Included if the certificate requested is a CA or enrolment certificate.
2. Optional. If omitted, the certificate's validity period is treated as starting at time 0.
3. For implicit certificates, the reconstruction value is included instead of these fields.
4. For implicit certificates, the ephemeral public key

Figure 34 — Checking consistency of permissions between a certificate request and a enrolment certificate

NOTE—For a self-signed request, the signature provides proof of possession of the private key but not authentication or authorization. In this case, the CA obtains assurance that the request is authorized by out-of-band means, for example because the request is sent over a channel that is already authorized. See E.7.2 for further discussion.

5.6.2 Certificate response

5.6.2.1 Processing

The security services are requested to decrypt and verify a certificate response via Sec-CertificateResponse-Processing.request. The information elements used by the security processing services to process the certificate response are specified in 7.2.25. The certificate response submitted to the security services shall be in the form of a 1609Dot2Data of type `encrypted` containing a ToBeEncrypted of type `certificate_response` or `request_error` (see Clause 6 for a specification of these data structures). The information elements also include a CMH in *Key Pair Only* mode, which is used to perform the decryption. The `RecipientInfo` field in the received certificate response indicates which public key was used to encrypt the response, and therefore which private key should be used to decrypt it.

NOTE—A processing flow illustrating the use of these primitives is given in D.3.

On receipt of Sec-CertificateResponseProcessing.request, the security services shall attempt to decrypt the certificate response using the provided CMH. If the decryption is successful, the security services shall attempt to verify that the certificate response is valid by the validity criteria given in 5.6.1.2.

The security services return the result via Sec-CertificateResponseProcessing.confirm. The information elements transferred via this primitive are specified in 7.2.26. The result shall be one of the following:

— Message failed decryption.

— Message passed decryption, but failed verification.

— Message passed decryption and verification, request was rejected and reason is included.

— Message passed decryption and verification, request was granted and certificate is included.

The validity of a certificate response is determined according to the criteria of 5.6.2.2.

In the case in which a certificate was successfully issued, the security services shall update the certificate information stored by the CME using the mechanisms supported by CME-AddCertificateRevocation.request.

The process flow for requesting certificate response processing is illustrated in Figure 35, where the dashed lines indicate entities and processing that are defined in this standard. Primitive names in the figure are abbreviated for compactness.

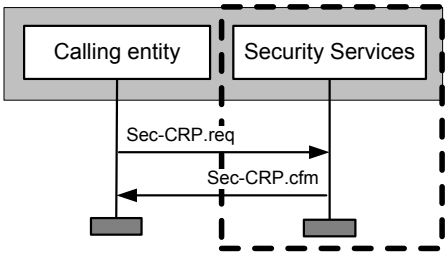


Figure 35—Process flow for requesting certificate request generation

5.6.2.2 Validity of certificate responses

If the certificate request was granted, the certificate response contains a certificate chain and a set of CRLs. The response is valid if the certificates and CRLs are valid as determined by the mechanisms specified in 5.5 and 5.6.4.2.

The end-entity certificate contained in a certificate response may contain permissions different from the permissions contained in the request.

If the certificate request was not granted, the certificate response contains a certificate request error, which is signed by the CA. A certificate request error is validly signed if it is signed with a CA certificate that would have been a valid signer of the issued certificate, and if that signature and the associated certificate chain are valid as determined by the mechanisms specified in 5.5.

NOTE—Although this standard does not require that the permissions in a certificate response match the permissions in the original request, profiles of this standard and system specifications may choose to impose such a requirement. If a certificate response contains different permissions from the request (for example, if it contains only PSIDs that the requester did not request), the certificate might not be usable by the requester.

5.6.3 Certificate response acknowledgement

As discussed in D.3, a CA may request that its response is acknowledged. This standard does not define a primitive dedicated to generating a certificate response acknowledgement. The certificate response consists of a payload, which is a ToBeEncryptedCertificateResponseAcknowledgment. This may be formed without invoking the security services. The security services may be requested to encrypt the payload via Sec-EncryptedData.request as specified in 5.3.3.

5.6.4 Certificate revocation information

5.6.4.1 Processing flow

A Certificate Revocation List (CRL) is a signed communication that is used to communicate revocation information about one or more certificates. This clause describes how the security processing services support CRL verification. See D.5 for a process flow that uses these primitives to obtain CRL information, verify the CRL, and update the revocation information managed by the CME.

The security processing services are requested to verify a CRL via Sec-CRLVerification.request. The information elements used by the security processing services are specified in 7.2.21.

The security processing services return the result of the request via CRLValidation.confirm. This indicates whether or not the CRL is valid according to the criteria of 5.6.4.2.

Figure 36 illustrates the use of the security services to verify a CRL. Primitive names in the figure are abbreviated for compactness.

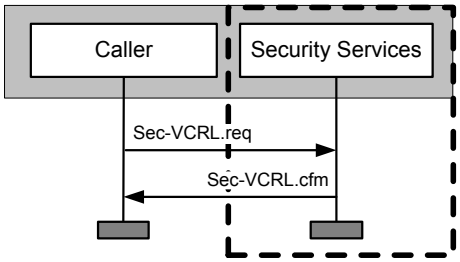


Figure 36—Process flow for processing a received CRL

NOTE—This standard does not specify primitives to create and sign CRLs.

5.6.4.2 Validity of CRLs

A CRL is signed by the issuer and contains an identifier for the CA whose issued certificates are to be revoked. A CRL is valid if:

- There is a certificate chain that leads from the signing certificate to a known trust anchor, following the methods of 5.5.2, such that:
  - All of the certificates in the chain are correctly formed using the data structures of Clause 6.
  - The certificate chain is internally consistent by the criteria of 5.5.3.
  - None of the certificates in the chain have been revoked as determined by the criteria of 5.5.4.

— All certificates in the chain can be cryptographically verified with the appropriate public keys as specified in 5.5.2.2.

— The CRL is consistent with the signing certificate as specified in this clause.

— The CRL can be cryptographically verified with the appropriate public keys as specified in 5.5.2.2.

A CRL is consistent with the signing certificate if:

— The CRL was created within the validity period of the certificate and expires within that validity period.

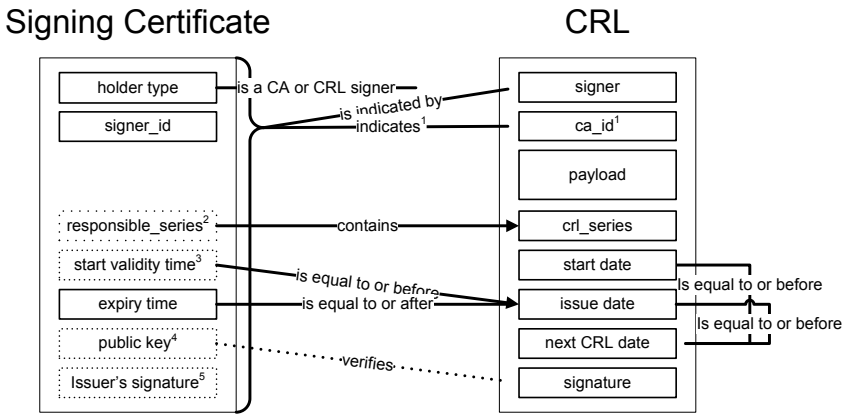
— The signing certificate is permitted to sign CRLs. CRLs shall be signed either by CA certificates or by specially designated CRL signing certificates, indicated by the `holder_type` field in the certificate.

— The signing certificate is permitted to sign CRLs for the indicated CA. A CA is always entitled to sign CRLs for its own certificates. A CRL signing certificate is entitled to issue CRLs for the specific CA that issued the CRL signing certificate.

— The signing certificate is permitted to sign the particular *CRL series* to which the CRL belongs. A CA may designate its certificates as belonging to one of multiple distinct series, identified by an integer, in order to manage the size of individual CRLs that a CA is entitled to sign.

— The public key indicated by the certificate can be used to cryptographically verify the signature on the communication.

Figure 37 illustrates the process of checking that a CRL is consistent with the signing certificate, and additionally captures the process of checking that the CRL is internally consistent as defined in Clause 6. For clarity, only the relevant fields within the data structures are shown. See Clause 6 for a detailed specification of the fields in the certificate and the signed data that encode this information. See Clause 7 for specification of the processing.



NOTES:
1. If holder type is a CA, the ca_id in the CRL identifies the signing certificate. If holder type is CRL signer, the ca_id in the CRL shall be is equal to the signer_id in the certificate.
2. Included only if holder type is CRL signer.
3. Optional. If omitted, start time is considered to be time 0.
4. For implicit certificates, the public key is derived rather than explicitly stated within the certificate.
5. Not included in an implicit certificate.

Figure 37—Checking consistency of permissions between a CRL and its signing certificate, and within the CRL

A CRL signing certificate is consistent with its issuing certificate if:

— The CRL signing certificate's validity region is wholly contained in the issuing certificate's validity region.

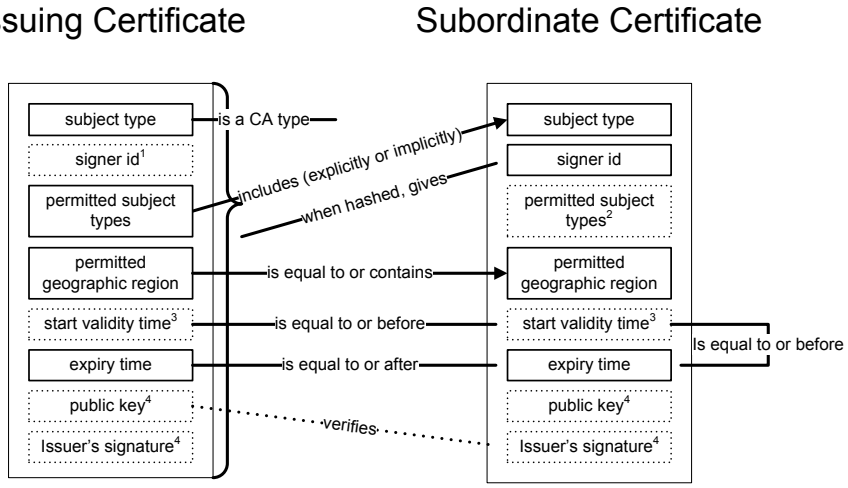
— The CRL signing certificate's validity period is within the issuing certificate's validity period.

— The issuing certificate is a CA certificate and is allowed to issue a certificate of the type of the CRL signing certificate.

— The issuing certificate signed the subordinate certificate for an explicit subordinate certificate or the cryptographic material in the certificate can be used to verify a signature for an implicit subordinate certificate.

A CRL is invalid if any subordinate certificate in the chain is inconsistent with its issuing certificate.

Figure 38 illustrates the process of checking that a subordinate certificate is consistent with its issuing certificate in a chain associated with a CRL, and additionally captures the process of checking that a certificate is internally consistent as defined in Clause 6. It illustrates the processing both for checking the consistency of a CRL signing certificate with its issuing certificate and for checking the consistency of an issuing and subordinate certificate pair further up the chain. For clarity, only the relevant fields within the data structures are shown in Figure 38. See Clause 6 for a detailed specification of the fields in the certificate and the signed data that encode this information. See Clause 7 for specification of the processing.



NOTES:
1. Not included if the holder type is root CA
2. Included if the subordinate certificate holder is a CA.
3. Optional. If omitted, the certificate's validity period is treated as starting at time 0.
4. For implicit certificates, the test of cryptographic validity is whether signed data can be cryptographically verified with a public key derived from the issuing certificate and the subordinate certificate.

Figure 38—Checking consistency of permissions between an issuing and a subordinate certificate, and within the certificate, in a chain associated with a CRL

5.6.4.3 Transport

A CRL may be distributed using the WAVE Short Message Protocol (WSMP) defined in IEEE Std 1609.3 with the PSID field in the WSMP header set to `0x23` and with the data field in the WAVE Short Message (WSM) containing a 1609Dot2Data with type set to `crl`. Implementations may support other distribution mechanisms.

An entity may request a CRL by generating a CRL request message, which is a 1609Dot2Data with type set to `crl_request` as specified in Clause 6, and sending it to a CRL distribution center using mechanisms outside the scope of this standard.

5.7 Certificate Management Entity

5.7.1 General

The Certificate Management Entity (CME) stores certificates and information about certificates. The CME shall store the following information relating to each certificate that it manages:

- The certificate data.
- The last received CRL time.
- The next expected CRL time.
- Whether or not the certificate has been verified by the security services.
- Whether or not the certificate is a trust anchor.
- Optionally, whether the certificate has been determined not to be trusted because, for example, it is part of an inconsistent or invalid certificate chain.

This standard provides CME-CertificateInfo.request and CME-CertificateInfo.confirm to allow calling entities to request information about certificates managed by the CME. The CME shall support requests for information about certificates that are identified using either the certificate itself, or an 8-byte or 10-byte hash of the certificate obtained as described in 6.2.7 and 6.3.44.

Information about certificates managed by the CME is updated via the primitives described in 5.7.2, 5.7.3, and 5.7.4.

Any certificate information added to an instance of a CME shall be available to all functional entities that have access to that CME.

5.7.2 Certificate revocation information

The CME is requested to update revocation information for a particular certificate via CME-AddCertificateRevocation.request. The information elements used by the CME are specified in 7.5.7. The CME shall respond with a CME-AddCertificateRevocation.confirm acknowledging receipt.

The CME is requested to update information about a revocation list series via CME-AddCrlInfo.request. The information elements used by the CME are specified in 7.5.9. The CME shall respond with CME-AddCrlInfo.confirm acknowledging receipt.

The process for updating revocation information is shown in Figure 39 where the dashed lines indicate functionality defined in this standard. Primitive names in the figure are abbreviated for compactness.

NOTE—The information provided to the CME by these primitives may be obtained from a valid CRL or from some other trusted source of revocation information. CRL validation is specified in 5.6.4.

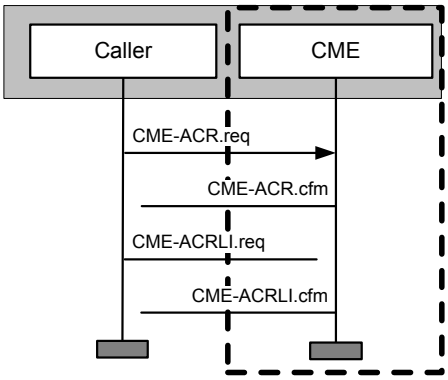


Figure 39—Process flow for updating certificate revocation information within the CME

5.7.3 Trust anchor

The CME is requested to add a certificate to its list of trust anchors via CME-AddTrustAnchor.request. The information elements used by the CME are specified in 7.5.3. The CME shall respond with a CME-AddTrustAnchor.confirm containing the result of the operation, which is success or an error code.

The CME shall reject a request to add a trust anchor if the certificate was not correctly formed or is determined to be revoked or expired.

The process for adding a trust anchor is shown in Figure 40 where the dashed lines indicate functionality defined in this standard. Primitive names in the figure are abbreviated for compactness.

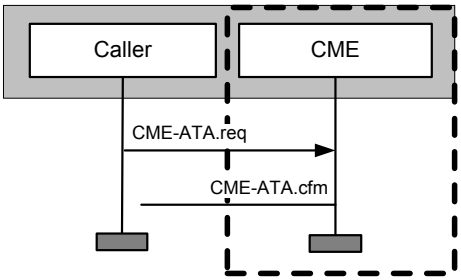


Figure 40—Process flow for updating trust anchor information within the CME

5.7.4 Other certificates

The CME is requested to add information about certificates other than trust anchors via CME-AddCertificate.request. The information elements used by the CME are specified in 7.5.5. The CME shall respond using CME-AddCertificate.confirm indicating success or failure. The CME shall reject a request to store certificate information if the certificate is poorly formed, has expired, or has been revoked.

The process for adding a certificate is shown in Figure 41, where the dashed lines indicate functionality defined in this standard. Primitive names in the figure are abbreviated for compactness.

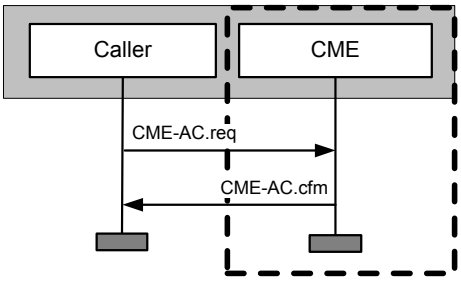


Figure 41—Process flow for adding information about certificates other than trust anchors to the CME

5.8 Cryptographic operations

5.8.1 Signature algorithms

This standard supports the Elliptic Curve Digital Signature Algorithm (ECDSA) specified in Federal Information Processing Standard (FIPS) 186-3, using either the P-224 or P-256 curves, optionally with the inclusion of additional information in the signature as specified in SEC 1 Version 2.[18] The signature is encoded in accordance with the EcdsaSignature data type and its associated encoding rules specified in Clause 6 of this standard.

5.8.2 Public key encryption algorithms: ECIES

The only asymmetric encryption algorithm supported in this standard is ECIES as specified in IEEE Std 1363a using the P-256 curve. In the data structures defined in this standard, keys for ECIES over P-256 are identified by the PKAlgorithm values `ecies_nistp256` respectively.

When encrypting with ECIES, the following constraints on the specification in IEEE Std 1363a shall be applied:

— The secret value derivation primitive shall be ECSVDP-DHC.

— The data encryption method shall be a stream cipher based on KDF2, which shall be parameterized by the choice.

— *Hash* = SHA-256.

[18] The additional information may allow faster verification operations as described in SEC1 Version 2 and [B2].

— The data authentication code shall be MAC1 which shall be parameterized by the choices:

— *Hash* = SHA-256

— *tBits* = 160.

— Encryption shall use non-DHAES mode.

— The elliptic curve points shall be converted to octet strings using least significant bit (LSB) compressed representation.

The output of this encryption is a triple (*V*, *C*, *tag*), where:

— *V* is an octet string representing the sender's ephemeral public key.

— *C* is the encrypted symmetric key.

— *tag* is the authentication tag.

5.8.3 Key pair generation

Key pairs for ECDSA or ECIES shall be generated according to the specification in B.4 of FIPS 196-3. Implementations should use a high-quality random number generator to generate the key pair. This is discussed in E.4.10.

5.8.4 Key pair validity

For ECDSA and ECIES, a key pair is judged to be valid or invalid relative to the criteria in 7.1.3 and A.16.10 of IEEE Std 1363-2000™.

5.8.5 Symmetric algorithms: AES-CCM

The only symmetric algorithm approved for use in this standard is the Advanced Encryption Standard (AES) in Counter Mode with Cipher Block Chaining Message Authentication Code (CCM) mode as described in National Institute for Standards and Technology (NIST) Special Publication (SP) 800-38C.

The ciphertext shall be calculated according to the specification of AES-CCM in NIST SP 800-38C.

The formatting mechanism used shall be the one described in A.2 of NIST SP 800-38C, with the following specific choices:

— **Control Information and Nonce (A.2.1)**: There is no associated data, so *Adata* = 0. The message authentication code length Tlen shall be 128 bits (16 octets). The octet length of the nonce N shall be 12, leaving three octets to encode the length of the data.

— **Formatting of the Associated Data (A.2.2):** There shall be no associated data.

The counter block generation mechanism used shall be the one described in A.3 of NIST SP 800-38C.

The input to AES-CCM encryption with no associated data is the nonce *N* and the payload *P* of length *Plen* bits. The output is the ciphertext *C* of length *Clen* = *Plen*+*Tlen* bits.

On decryption using the mechanisms of NIST SP 800-3C, the nonce *N* shall be set equal to the contents of the `nonce` field; the ciphertext C shall be set equal to the contents of the `ciphertext` field; and the ciphertext length *Clen* shall be set equal to eight times the encoded length of the `ciphertext` field.

5.8.6 Implicit certificates

Implicit certificates are processed as specified in Standards for Efficient Cryptography (SEC) 4.

When an implicit certificate is encoded for hashing as described in 3.2.1 of SEC 4:

— The encoded data input to the hash function shall be the PublicKeyReconstructionHashInput described in Clause 6.

— SHA-256 shall be used as the Hash algorithm H within the integer hash Hn specified in 2.3 of SEC 4.

The private key is judged as valid or invalid relative to an implicit certificate using the techniques of 3.6 of SEC 4.

When a CA issues an implicit certificate, the data structure containing the certificate (defined as ToBeEncryptedCertificateResponse in Clause 6) also includes the value `recon_priv`. This value is identical to the integer *r* specified in 3.6 of SEC 4, which is used to update the private key. Using the primitives in this standard, the private key is updated by invoking Sec-CryptomaterialHandle-StoreCertificate.request with:

— *Private Key Transformation.A* equal to the SHA-256 hash of the PublicKeyReconstructionHashInput of the end-entity certificate from the `certificate_chain` field from the ToBeEncryptedCertificateResponse.

— *Private Key Transformation.B* equal to the `recon_priv` value from the ToBeEncryptedCertificateResponse.

5.8.7 Hash algorithms: SHA-256

The only hash algorithm approved for use in this standard is SHA-256 as specified in the Federal Information Processing Standard (FIPS) 180-3. In this standard, the phrase "the SHA-256 hash of [an octet string]" is used to mean "the hash of [that octet string] obtained using SHA-256 as specified in FIPS 180-3."

6. Data structures for secure communication

6.1 Presentation language

6.1.1 General

This standard specifies data formats using a presentation language based on the presentation language used by Transport Layer Security (TLS) [Internet Engineering Task Force (IETF) Request for Comments (RFC) ([B18])]. This clause specifies that presentation language and its encoding. Annex F provides a copyright statement for this clause.

In this standard, *encoding* is used to denote the process of converting from an internal representation of a structure to a flat octet string containing the same information. *Decoding* is used to denote the process of converting a flat octet string into an internal representation of the structure encoded in that octet string. The process of decoding fails if the received octet string is not a valid encoding of the expected structure. In this case, the received octet string cannot be parsed.

6.1.2 Notation conventions

Elements of the presentation language are presented in Courier font. For example:

```
This is a sample of Courier font.
```

Presentation language statements contain variable names, data types, and functions. Variable names are all lowercase. Multiple words in a variable name are indicated by underscores, as in `variable_name`. Data types begin with an uppercase letter. Multiple words in a data type are indicated by uppercase letters, as in `DataType`. The exceptions to this rule are the built-in data types `uint8`, `uint16`, `uint32`, `uint64`, and `opaque`. Functions (such as `select` statements) follow the same conventions as variable names.

For compactness, a field within a structure may be referred to using the `structure.field` notation.

Comments may be included, and are preceded by two forward slash marks: `//`. For example:

```
// this is a comment
```

6.1.3 Basic block size

The basic data block size is one octet. Multiple octet data items are concatenations of octets in network byte order. All lengths are expressed in octets.

6.1.4 Numbers

The following numeric types are predefined:

- `uint8` is a single-octet encoding of an unsigned 8-bit integer.
- `uint16` is a two-octet big-endian encoding of an unsigned 16-bit integer.
- `uint32` is a four-octet big-endian encoding of an unsigned 32-bit integer.
- `uint64` is an eight-octet big-endian encoding of an unsigned 64-bit integer.
- `sint32` is a four-octet encoding of a 32-bit signed integer, in which the first bit is the sign bit and the remaining 31 bits encode the integer in big-endian form.

6.1.5 Fixed-length vectors

The syntax for specifying a vector `TNew` that is a fixed-length vector of type `TOld` is:

```
TOld  TNew[n];
```

where `n`, the length in octets, is a multiple of the size of `TOld`. The vector is encoded as `n` octets containing the data.

Example (1):

```
uint8  XY8[2];
```

This defines an XY8 as a concatenation of uint8s of length 2 octets (which should not be read as "a concatenation of two uint8s," even though in this case the two are equivalent.). An XY8 encoding the values (0x12, 0x34) is encoded as the octets 0x12 0x34.

Example (2):

```
uint32  XY32[8];
```

This defines an XY32 as a concatenation of uint32s of length 8 octets, in other words a concatenation of two uint32s. An XY32 encoding the values (0x12, 0x34) is encoded as the octets 0x00 0x00 0x00 0x12 0x00 0x00 0x00 0x34.

6.1.6 Variable-length vectors

6.1.6.1 Variable-length vectors with fixed-length length encoding

Subclause 6.1.6.1 defines a vector that has a variable number of elements, and thus has variable length. A vector of this type is encoded using a (length, value) form. In 6.1.6.1, the length indication itself has a fixed length, regardless of the number of elements in the vector. The syntax for specifying a new type TNew that is a variable-length vector of type TOld and maximum length n with fixed-length encoding of the length indication is:

```
TOld  TNew<n>;
```

The length in octets is encoded as a big-endian integer of the minimum size necessary to encode n. In definitions of types with variable length vectors, n shall take one of the values 2^8-1, 2^16-1, 2^32-1, or 2^64-1, denoting that the length is encoded in one, two, four, or eight octets respectively.

Example (1):

```
uint8       AsciiChar;
AsciiChar   Name<2^8-1>;
```

This defines a Name as a concatenation of AsciiChars of length no more than 2^8-1 octets. A Name encoding the value "abc" is encoded as the octets 0x03 0x61 0x62 0x63. The leading octet, 0x03, is the length and the remainder is the data.

Example (2):

```
uint8       AsciiChar;
AsciiChar   LongName<2^16-1>;
```

This defines a LongName as a concatenation of AsciiChars of length no more than 2^{16}-1 octets. A LongName encoding the value "abc" is encoded as the octets 0x00 0x03 0x61 0x62 0x63. The leading two octets (16 bits), 0x00 0x03, are the length and the remainder is the data.

Example (3):

```
uint8       AsciiChar;
AsciiChar   Name<2^8-1>;
Name        OfficerName<2^8-1>;
```

This defines the `OfficerName` type as a concatenation of `names` of length no more than 2^8-1 octets. An `OfficerName` encoding the two `names` "abc" and "def" is encoded as the octets 0x`08` 0x`03` 0x`61` 0x`62` 0x`63` 0x`03` 0x`64` 0x`65` 0x`66`.

NOTE—In this example, it is up to the implementer to check that the individual names are short enough for the concatenation of their encodings to fit in 28-1 or fewer octets.

6.1.6.2 Variable-length vectors with variable-length length encoding

Subclause 6.1.6.2 defines a vector that has a variable number of elements, and thus has variable length. A vector of this type is encoded using a (length, value) form. In 6.1.6.2, the length indication has a variable length that depends on the number of elements in the vector. The syntax for specifying a new type `TNew` that is a variable-length vector of type `TOld` and has maximum length 2^{56}-1 octets and variable-length encoding of the length indication is:

```
TOld  TNew<var>;
```

In specifications of variable-length vectors with variable-length length encoding, the string "<var>" literally appears in the definition; it is not replaced by a number.

The length in octets is encoded as follows. First, encode the length as positive integer using the minimum number of bits necessary. Then, obtain a bit string *ll* (for "length of length") as follows:

- if length < 2^7, *ll* is the bit "0"
- else if length < 2^{14}, *ll* is the bits "10"
- else if length < 2^{21}, *ll* is the bits "110"
- else if length < 2^{28}, *ll* is the bits "1110"
- else if length < 2^{35}, *ll* is the bits "11110"
- else if length < 2^{42}, *ll* is the bits "111110"
- else if length < 2^{49}, *ll* is the bits "1111110"
- else if length < 2^{56}, *ll* is the bits "11111110"

The full encoded length field is [*ll* || P || length] where P, the padding string, is an array of 0 bits chosen to be the minimum length such that the full encoded length field is an integer number of octets in length.

Example (1), length encoding:

- length 0 is encoded as 0x`00` (*ll* = `0`, P = `000 000`, length = `0`)
- length 1 is encoded as 0x`01` (*ll* = `0`, P = `000 000`, length = `1`)
- length 3 is encoded as 0x`03` (bits `0000 0011`; *ll* = 0, P = `000 00`, length = `11`)
- length 127 is encoded as 0x`7f` (bits `0111 1111`; *ll* = `0`, P =<omitted>, length = `111 1111`)
- length 128 is encoded as 0x`80` 0x`80` (bits `1000 0000 1000 0000`; *ll* = 1`0`, P = `000 0000`, length = `1000 0000`)

Example (2):

```
uint8       AsciiChar;
AsciiChar   VarName<var>;
```

This defines a `VarName` as a concatenation of `AsciiChar`s.

A `Name` encoding the value “abc” is encoded as the octets 0x`03` 0x`61` 0x`62` 0x`63`. The leading octet, 0x`03`, is the length and the remainder is the data.

A `Name` encoding the value “abcdefghabcdefghabcdefghabcdefghabcdefghabcdefghabcdefghabcdefgh-abcdefghabcdefghabcdefghabcdefghabcdefghabcdefghabcdefghabcdefgh” (128 characters) is encoded as the octets:

```
0x80 0x80
    0x61 0x62 0x63 0x64 0x65 0x66 0x67 0x68
    0x61 0x62 0x63 0x64 0x65 0x66 0x67 0x68
    0x61 0x62 0x63 0x64 0x65 0x66 0x67 0x68
    0x61 0x62 0x63 0x64 0x65 0x66 0x67 0x68
    0x61 0x62 0x63 0x64 0x65 0x66 0x67 0x68
    0x61 0x62 0x63 0x64 0x65 0x66 0x67 0x68
    0x61 0x62 0x63 0x64 0x65 0x66 0x67 0x68
    0x61 0x62 0x63 0x64 0x65 0x66 0x67 0x68
    0x61 0x62 0x63 0x64 0x65 0x66 0x67 0x68
    0x61 0x62 0x63 0x64 0x65 0x66 0x67 0x68
    0x61 0x62 0x63 0x64 0x65 0x66 0x67 0x68
    0x61 0x62 0x63 0x64 0x65 0x66 0x67 0x68
    0x61 0x62 0x63 0x64 0x65 0x66 0x67 0x68
    0x61 0x62 0x63 0x64 0x65 0x66 0x67 0x68
    0x61 0x62 0x63 0x64 0x65 0x66 0x67 0x68
    0x61 0x62 0x63 0x64 0x65 0x66 0x67 0x68
```

6.1.7 The opaque and opaqueExtLength type

The notation:

```
opaque fieldName[n],
opaque fieldName<n>,
opaque fieldName<var>,
```

is used to denote a block of data that may be fixed or variable-length, with length denoted as described in the preceding clauses. Data in a field of type `opaque` is not further parsed by the security services. In this standard, `opaque` types carry data that the security services protect but do not use.

The notation:

```
opaqueExtLength fieldLength
```

is used to denote a variable-length block of data that is not further parsed by the security services. The data in a field of type `opaqueExtLength` is *not* preceded by its length when encoded. Instead, the length of the field is deduced from external information.

NOTE—For a recipient to correctly process a structure that contains an `opaqueExtLength` field, the recipient needs to know the length. For each structure in this standard that contains an `opaqueExtLength`, the description of that structure identifies how that length is obtained. Within this standard, all encoded data output by or input to the primitives is encoded as a 1609Dot2Data (see 6.2.2). The 1609Dot2Data structure and its substructures are defined so that external lengths are always available at the time of decoding.

6.1.8 Enumerated type

A field of type `enum` is a list of labels, each with a unique value, and an optional unlabelled maximum value. Values not explicitly assigned in a definition of an enum are reserved for assignment in a future version of this standard and shall not be used. An enum is encoded as an integer. The size of the integer is the minimum number of octets capable of representing the maximum value in the enum definition. In this standard, the maximum value in the enum definition is always set as some power of 2^8, minus 1, to indicate that the enum is to be encoded in a particular number of octets.

Example (1):

```
enum {usa(0), canada(1), mexico(2), (2^8-1)}
     NaftaSignatories;
```

Because the maximum value given is 2^8-1, the enum is encoded as a single octet. The label `canada` as a member of `NaftaSignatories` is encoded as the value 0x`01`.

Example (2):

```
enum {usa(0), canada(1), mexico(2), (2^16-1)}
     NaftaSignatoriesWithRoomForExpansion;
```

Because the maximum value given is 2^{16}-1, the enum is encoded as two octets. The label `canada` as a member of `NaftaSignatoriesWithRoomForExpansion` is encoded as the value 0x`00` 0x`01`.

An `enum` may note that certain values are assigned for private use, for example for testing, using the `private_use` notation.

Example (3):

```
enum {usa(0), canada(1), mexico(2), private_use(240..255), (2^8-1)}
     NaftaSignatories;
```

6.1.9 The psid type

A field of type `psid` contains a Provider Service Identifier as defined in IEEE Std 1609.3. This is a variable-length type that encodes the length inside the data rather than in an external length field. See IEEE Std 1609.3 for further details.

6.1.10 Constructed types

Structured types may be constructed from primitive types for convenience using the `struct` syntax, as in the following example:

```
uint8       AsciiChar;
AsciiChar   Name<2^8-1>;
enum        {chair(0), treasurer(1), secretary(2)}
                OfficerPosition;
struct      {
                Name              name;
                OfficerPosition   position;
            } Officer;
officer     officers<2^16-1>;
```

An organization with Alice as chair, Bob as treasurer, and Carol as secretary might encode its `officers` structure as the following string of hex digits (line breaks included for clarity only):

```
00 19                // length of encoded data is 0x19 octets
                     // encoded as 2-octet integer
   05                // length of "Alice"
      41 6C 69 63 65 // "Alice", ASCII-encoded
   00                // chair(0)
   03                // length of "Bob"
      42 6F 62       // "Bob", ASCII-encoded
   01                // treasurer(1)
   05                // length of "Carol"
      43 61 72 6F 6C // "Carol", ASCII-encoded
   02                // secretary(2)
```

6.1.11 The select statement

Structures may contain conditional fields (i.e., fields that are present or absent depending on some condition). The selector is an enumerated type, as shown below, or a flag selector, discussed in 6.1.13. For example:

```
uint16      Pressure;
enum        {car(0), motorbike(1)}
                VehicleType;
struct      {
                VehicleType  type;
                select (type) {
                case car:
                    Pressure frontLeft;
                    Pressure frontRight;
                    Pressure rearLeft;
                    Pressure rearRight;
                case motorbike:
                    Pressure front;
                    Pressure rear;
                }
            } TirePressureInfo;
```

The syntax:

```
case sel_1:  ;
```

within a `select` statement denotes that one possible value of the selector (in this case, `sel_1`) has no conditional fields associated with it.

The syntax:

```
case sel_1:
case sel_2:
    Type1  t1;
case sel_3:
    Type2  t2;
```

within a `select` statement denotes that multiple values of the selector (in this case, `sel_1` and `sel_2`) have the same conditional fields associated with them (in this case, `t1`). A case statement consists of two parts, the selector and the content. The selector is terminated by a colon. The content is terminated by the next `case` statement if one exists, and by the closing brace (`}`) otherwise. If multiple selector values correspond to the same content, the selector may include all of those values, separated by commas.

To save space in the specification, the above structure may also be expressed as:

```
case sel_1, sel_2:
    Type1  t1;
case sel_3:
    Type2  t2;
```

The comma-separated list of selectors might occupy multiple lines; the final line will end with a colon and any other line will end with a comma.

The syntax:

```
case other_value:
    Type1  t1;
```

within a `select` statement specifies the syntax of the data structure for selector values not covered by an earlier case. It is only included if the syntax of the data structure is defined in this case and, if present, is the final case within each select statement.

6.1.12 The extern statement

A structure may depend on data external to it. The notation is given below:

```
uint16     Pressure;
struct     {
               extern  VehicleType type;
               select (type) {
               case car:
                   Pressure frontLeft;
                   Pressure frontRight;
                   Pressure rearLeft;
                   Pressure rearRight;
               case motorbike:
```

```
            Pressure front;
            Pressure rear;
        }
    } TirePressureInfo;
```

If a structure depends on external data, that data is not encoded in the encoding of the structure. For example, if the `TirePressureInfo` structure above was of type `motorbike` and had `front` = 0x1234 and `rear` = 0x5678, the full encoding of the structure would be:

```
12 34 56 78.
```

NOTE—For a recipient to correctly process a structure that depends on external data, the recipient needs to know the external data. For each structure in this standard that uses external data, the description of that structure identifies how that external data is obtained. Within this standard, all encoded data output by or input to the primitives is encoded as a 1609Dot2Data (see 6.2.2). The 1609Dot2Data structure and its substructures are defined so that external fields are always available at the time of decoding.

6.1.13 Flags

6.1.13.1 Use of flags field

The flags field provides an alternative mechanism for specifying structures with conditional fields. A `flags` type consists of a declaration of the flags that it may encode and the position of a given flag within the encoding. The number in parentheses following a flag indicates its bit position within the flags field encoding. Each flag may take on one of two states, "set" and "not set", and is encoded in a single bit, 1 and 0 respectively. A flags type definition may note that certain fields are assigned for private use, for example for testing, using the `private_use` notation. Values not explicitly assigned in a definition of a flags type are reserved for assignment in a future version of this standard and shall not be used.

The contents of a structure are made conditional on the contents of a flags type using the `if_set()`, `if_not_set`(), `if_any_set`, `if_none_set`, and `if_other_value_set` statements. These statements are used with opening and closing braces to denote the portions of the structure that are conditional on the given flag value, as demonstrated for the `if_set` statement in the following example.

In this standard, the values of the entries in the `flags` declaration are consecutive. The highest bit position of an entry in the `flags` declaration is 27.

The `if_set` and `if_not_set` statements each take two arguments and evaluate to "true" or "false". The first argument is the field to be checked, and the second argument is the bit position to check within the field. The `if_set` statement evaluates to true if the bit at the given bit position within the given field is 1, and false otherwise. The `if_not_set` statement evaluates to false if bit at the given bit position within the given field is 1, and true otherwise.

The `if_any_set` and `if_none_set` statements may take an arbitrary number of arguments and evaluate to "true" or "false". The first argument is the field to be checked and the remaining arguments are the values to check for. The `if_any_set` statement evaluates to true if any of the individual `if_any_set` statements are true, and the `if_none_set` statement evaluates to true if all of the individual `if_any_set` statements are false.[19]

[19] The `if_none_set` statement is not used in this standard but is defined here for completeness and for possible future use.

The `if_other_value_set` statement takes a single argument (the "flags"). If this field has any bits set other than the bits given in the definition of the flags type, the statement evaluates to true. See the example in the next section.[20]

An example of the use of a flags field is

```
flags       {start_included(0), end_included(1)}
                TimeInfoFlags;
uint64      Time;
struct      {
                TimeInfoFlags    time_flags;
                if_set(time_flags, start_included) {
                    Time       start_time;
                }
                if_set(time_flags, end_included) {
                    Time       end_time;
                }
                if_other_value_set(time_flags) {
                    opaque        extra_info<var>;
                }
             } TimeInfo;
```

6.1.13.2 Encoding of flags field

An encoded flags field is variable length and contains the length and the value in a single field.

The encoded flags value is a little-endian bitmask of the individual flags that are set within the field. For example, the value 0 is encoded by setting the rightmost bit of the value; the value 8 is encoded by setting the ninth-rightmost bit of the value.

The length is encoded as follows. Let MaxFlag be the highest flag value that is in the set state. Then:

- if MaxFlag ≤ 6, the length field is the bit "0"
- else if MaxFlag ≤ 13, the length field is the bits "10"
- else if MaxFlag ≤ 20, the length field is the bits "110"
- else if MaxFlag ≤ 27, the length field is the bits "1110"
- otherwise the length field is undefined

The full encoded flags field is [length || P || values] where P, the padding string, is an array of 0 bits chosen to be the minimum length such that the encoded flags field is an integer number of octets in length.

Example (1):

- "no flags set" is encoded as 0x`00`
- "flag 0 set" is encoded as 0x`01`
- "flags 0 and 1 set" is encoded as 0x`03`

[20] The `if_other_value_set` statement specifies a single extra_info field, no matter how many other values are set. Implementers who wish to extend this standard should be careful to specify behavior explicitly if their extension makes it possible to set two or more additional values in a structure that uses `if_other_value_set`.

— "flag 6 set" is encoded as 0x40 (bits 0100 0000)

— "flag 7 set" is encoded as 0x80 0x80 (bits 1000 0000 1000 0000)

— "flags 0 and 8 set" is encoded as 0x81 0x01 (bits 1000 0001 0000 0001)

Example (2):

Consider a TimeInfo structure as defined above that contains start_time with value (in hexadecimal) 0011223344556677. The encoding of this structure is:

```
01 // time_flags encodes start_included (0)
0011223344556677 // start_time
```

Example (3):

Consider an encoded TimeInfo structure whose encoding is the following:

```
06 00 11 22 33 44 55 66 77 02 01 02
```

Then the contents of the encoding are to be interpreted as follows:

— time_flags is 06, which means that bits 1 (end_included) and 2 (undefined, therefore private use) are set.

— There is no value for start_time.

— The value of end_time is (hexadecimal) 0011223344556677.

— extra_info is encoded as 02 01 02, meaning an octet string of length 2 consisting of the octets 0x01 0x02.

6.2 Structures for secure communications

6.2.1 General

This clause specifies the structures to be used for secure communications. Examples are given in D.1.

The order in which the structures are defined below is hierarchical, based on the first use in a prior structure. For example, in 6.2.2 1609Dot2Data is defined using several structures, of which the first three are ContentType, SignedData, and SignedWsa. Subsequently, ContentType is defined in 6.2.3, SignedData is defined in 6.2.4, and SignedWsa is defined in 6.2.21. The subclauses between 6.2.4 and 6.2.21 are used to define structures used within SignedData, and so on.

Additionally, in the electronic version of the standard, all uses of a structure name are hyperlinked to the title of the subclause that defines the structure.

6.2.2 1609Dot2Data

```
struct  {
            uint8        protocol_version;
            ContentType  type;
            select (type) {
            case unsecured :
                opaque           data<var>;
            case signed, signed_partial_payload,
                 signed_external_payload:
                SignedData    signed_data;
            case signed_wsa:
                SignedWsa        signed_wsa;
            case encrypted :
                EncryptedData encrypted_data;
            case crl_request :
                CrlRequest       crl_request;
            case crl :
                Crl          crl;
            case other_value:
                opaque           data<var>;
            }
        } 1609Dot2Data;
```

This data type is used to contain the other data types in this clause. The fields in the 1609Dot2Data have the following meanings:

— `protocol_version` contains the current version of the protocol. The version described in this document is version 2, represented by the integer 2. There are no major or minor version numbers.

— `type` contains the type of the data.

6.2.3 ContentType

```
enum    {   unsecured (0), signed(1), encrypted (2),
            certificate_request(3), certificate_response(4),
            anonymous_certificate_response(5),
            certificate_request_error(6), crl_request(7),
            crl(8),
            signed_partial_payload(9),
            signed_external_payload(10),
            signed_wsa(11),
            certificate_response_acknowledgment (12),
            private_use (240..255),
            (2^8-1)
        }   ContentType;
```

This type specifies the subfields that appear within a 1609Dot2Data or ToBeEncrypted. Values 240–255, inclusive, are assigned for private use.

6.2.4 SignedData

```
struct {
           extern ContentType   type;
           SignerIdentifier     signer;
           ToBeSignedData       unsigned_data;
           Signature            signature;
       } SignedData;
```

In this structure:

— type is obtained from the enclosing structure (1609Dot2Data or ToBeEncrypted) and is included in this structure as a pass-through to the enclosed ToBeSignedData structure.

— signer determines the keying material and hash algorithm used to sign the data.

— unsigned_data contains the data.

— signature contains the digital signature itself, calculated over the hash of the single byte type concatenated with the encoding of unsigned_data. The mechanism to be used to calculate this encoding is defined in the description of the ToBeSignedData structure. The Signature structure contains an extern field, algorithm, indicating the algorithm to be used to verify the signature. For Signature structures encapsulated in a SignedData, algorithm is determined as follows:

— If signer.type is certificate_digest_with_ecdsa_p224, then algorithm is ecdsa_nistp224_with_sha224.

— If signer.type is certificate_digest_with_ecdsa_p256, then algorithm is ecdsa_nistp256_with_sha256.

— If signer.type is certificate_digest_with_other_algorithm, then algorithm is signer.algorithm.

— If signer.type is certificate, then:

— If signer.certificate.version_and_type is 2, then algorithm is equal to signer.certificate.unsigned_certificate. verification_key.algorithm.

— If signer.certificate.version_and_type is 3, then algorithm is equal to signer.certificate.unsigned_certificate. signature_alg.

— If signer.type is certificate_chain, then:

— If signer.certificates[n].version_and_type is 2, then algorithm is equal to signer.certificates[n].unsigned_certificate.verification_-key.algorithm, where "signer.certificates[n]" denotes the last certificate in signer.certificates.

— If signer.certificates[n].version_and_type is 3, then algorithm is equal to signer.certificates[n]. unsigned_certificate. signature_alg, where "signer.certificates[n]" denotes the last certificate in signer.certificates.

6.2.5 SignerIdentifier

```
struct {
            SignerIdentifierType  type;
            select (type) {
            case self: ;
            case certificate_digest_with_ecdsap224 :
            case certificate_digest_with_ecdsap256 :
                HashedId8       digest;
            case certificate:
                Certificate   certificate;
            case certificate_chain:
                Certificate   certificates<var>;
            case certificate_digest_with_other_algorithm :
                PKAlgorithm   algorithm;
                HashedId8       digest;
            case other_value:
                 opaque       id<var>;
            }
        } SignerIdentifier;
```

This structure allows the recipient of data to determine which keying material to use to authenticate the data. It shall contain either a certificate, a digest of a certificate, or a certificate chain, as follows:

— If `type` is `self(0)` then the data is signed by a key contained within itself. This value shall only be used when signing a certificate request.

— If `type` is `certificate_digest_with_ecdsa_p224(1)` or `certificate_digest_with_ecdsa_p256(2)`, then the SignerIdentifier contains the HashedId8 of the relevant certificate, obtained as described in the description of the HashedId8 structure.

— If `type` is `certificate(3)` then the SignerIdentifier contains a `certificate`.

— If `type` is `certificate_chain(4)` then the SignerIdentifier contains two or more certificates. The last certificate in the chain shall be the certificate that signed the data, the next to last shall be the CA certificate that issued the end-entity certificate, and so on with each certificate in the chain issuing the next certificate.[21]

— If `type` is `certificate_digest_with_other_algorithm (5)` then the SignerIdentifier contains the HashedId8 of the relevant certificate and the field `algorithm` indicates the algorithm used to generate the associated signature.

6.2.6 SignerIdentifierType

```
enum { self (0) , certificate_digest_with_ecdsap224(1),
       certificate_digest_with_ecdsap256(2),
       certificate(3), certificate_chain (4),
       certificate_digest_with_other_algorithm(5),
       private_use (240..255),
    (2^8-1)}  SignerIdentifierType;
```

This enumerated type is used to indicate the data included in a SignerIdentifier. Values 240–255, inclusive, are assigned for private use.

[21] If certificate A issued certificate B, the algorithm used to sign certificate B is given by `algorithm` field of the verification public key in certificate A.

6.2.7 HashedId8

```
opaque HashedId8[8];
```

This data structure contains the hash of another data structure. The HashedId8 for a given data structure shall be calculated by calculating the SHA-256 hash of the encoded data structure and taking the low-order 8 bytes of the hash output. If the data structure is a Certificate, the encoded Certificate, which is input to the hash, shall use the compressed form for all elliptic curve points.

6.2.8 ToBeSignedData

```
struct {
            extern ContentType   type ;
            TbsDataFlags         tf;
            select(type) {
            case signed :
                Psid             psid;
                opaque           data<var>;
            case signed_partial_payload :
                Psid             psid;
                extern opaque    ext_data<var>;
                opaque           data<var>;
            case signed_external_payload :
                Psid             psid;
                extern opaque    ext_data<var>;
            case other_value:
                opaque           data<var>;
            }
            if_flag_set (tf, use_generation_time) {
                Time64WithStandardDeviation     generation_time;
            }
            if_flag_set (tf, expires) {
                Time64                   expiry_time;
            }
            if_flag_set (tf, use_location) {
                ThreeDLocation
                    generation_location;
            }
            if_flag_set (tf, extensions) {
                TbsDataExtension extensions<var>;
            }
            if_other_value_set (tf) {
                opaque other_data<var>;
            }
        } ToBeSignedData;
```

In this structure:

- `tf` may contain the flags:
  - `use_generation_time`, indicating that the structure contains the generation time.
  - `expires`, indicating that the structure contains an `expiry_time` field.
  - `use_location`, indicating that the `generation_location` field is included.

— `extensions`, indicating that the `extensions` field is included.

— `psid`, if present, is a Provider Service Identifier (PSID). See the definition of Psid in 6.2.9 for more details.

— `ext_data`, if present, contains data that is used to calculate the signature, but is not explicitly transmitted within the SignedData. The encoding of the `ext_data` shall be specified explicitly by any entities that use this field. For example, entities that use this field shall specify whether or not the length of the field is part of the encoding.

— `data`, if present, contains the data for the receiver. This is not interpreted by the security protocol.

— `generation_time` indicates the time at which the structure was generated. See 5.5.5 for discussion of the meaning of this field.

— `expiry_time`, if present, contains the time after which the data should no longer be considered relevant. If both `generation_time` and `expiry_time` are present, `generation_time` shall be strictly earlier than `expiry_time`.

— `generation_location`, if present, contains the location at which the signature was generated.

— `extensions`, if present, contains additional data that may be used by the receiver. This field is provided for future extensibility and no specific use is currently defined.

When the structure is encoded in order to be digested to generate or check a signature, the process to be followed is as follows:

a) Encode the ToBeSignedData according to the encoding rules.

b) If `type` is `signed_external_payload` or `signed_partial payload`, insert the encoding of `ext_data` at the location indicated. The encoding of `ext_data` consists of the length of the data in `ext_data`, encoded as a variable-length length `<var>`, followed by the `opaque` contents of `ext_data`. If there is no `ext_data`, then `ext_data` is encoded as a zero-length array (i.e., as the single octet `0x00`).

6.2.9 Psid

```
psid      Psid;
```

The `psid` type is defined in 6.1.9. It is used within the security services to encode permissions.

6.2.10 TbsDataFlags

```
flags  { use_generation_time(0), expires(1),
           use_location(2), extensions(3) } TbsDataFlags;
```

This structure is used by a ToBeSignedData to inform the recipient of the sending options that the sender chose.

6.2.11 Time64WithStandardDeviation

```
struct {
         Time64  time;
         uint8   log_std_dev;
       } Time64WithStandardDeviation;
```

This structure encodes a time and the standard deviation of that time. The fields have the following meaning:

— `time` is the time being encoded; see the definition of Time64 for more details.

— `log_std_dev` is a `uint8`. Values `0` – `0xfd` represent the rounded up value of the log to the base 1.134666 of *s*, where *s* is the implementation's estimate of the standard deviation in `time` in units of nanoseconds. The value `0xfe` represents any value greater than 1.134666^{244} nanoseconds (i.e., a day or longer). The value `0xff` indicates that the standard deviation is not known.

NOTE 1—Taking the log can be implemented as a lookup table. For example, a standard deviation of 1 μs = 55; of 1 ms = 110; of 1s = 165; of 1 minute = 197; and of one hour = 229.

NOTE 2—This standard does not specify minimum performance requirements for the accuracy of the estimate of the standard deviation. Secure communications entities that use this structure may impose minimum performance requirements by means out of scope of this standard.

6.2.12 Time64

```
uint64   Time64;
```

This data structure is a 64-bit integer, encoded in big-endian format, giving the number of (TAI) μs since 00:00:00 UTC, 1 January, 2004.

6.2.13 ThreeDLocation

```
struct {
            sint32        latitude;
            sint32        longitude;
            opaque        elevation[2]
        } ThreeDLocation;
```

The `latitude` and `longitude` fields contain the latitude and longitude as a `sint32` type, encoding the latitude and longitude with precision one-tenth (1/10) microdegree relative to the World Geodetic System (WGS)-84 datum as defined in NIMA Technical Report TR8350.2.

The encoded integer in the `latitude` field shall be no more than 900 000 000 and no less than –900 000 000, except that the value 900 000 001 shall be used to indicate the latitude was not available to the sender.

The encoded integer in the `longitude` field shall be no more than 1 800 000 000 and no less than –1 800 000 000, except that the value 1 800 000 001 shall be used to indicate that the longitude was not available to the sender.

The `elevation` field represents the geographic position above or below the WGS84 ellipsoid. The 16-bit value is interpreted as an integer number of decimeters and represents an asymmetric range of positive and negative values. The encoding is as follows: the range 0x0000 to 0xEFFF (0 to 61439 decimal) are positive numbers representing elevations from 0 to +6143.9 m (i.e., above the reference ellipsoid). The range 0xF001 to 0xFFFF are negative numbers representing elevations from –409.5 m to –0.1 m (i.e., below the reference ellipsoid). An elevation higher than +6143.9 m is represented 0xEFFF. An elevation lower than –409.5 m is represented 0xF001. An elevation data element of 0xF000 corresponds to unknown elevation.

Examples of this encoding are: the elevation 0 m is encoded as 0x0000. The elevation –0.1 m is encoded as 0xFFFF. The elevation +100.0 meters is encoded as 0x03E8.

NOTE—This data structure is consistent with the location encoding used in [B21].

6.2.14 TbsDataExtension

```
struct {
            TbsDataExtensionType   type;
            opaque                    value<var>;
        }   TbsDataExtension;
```

This structure may be used to encapsulate additional data, distinct from the payload data, within signed data.

The contents of this structure shall be consumed by the security services and shall not be passed to a secure communications entity.

6.2.15 TbsDataExtensionType

```
enum    {
            private_use (240..255), (2^8-1)
        }   TbsDataExtensionType;
```

This enumerated type specifies the type of the value in a TbsDataExtension structure. No types are currently supported. Values 240–255, inclusive, are assigned for private use.

6.2.16 Signature

```
struct {
            extern PKAlgorithm algorithm;
            select(algorithm) {
            case ecdsa_nistp224_with_sha224:
            case ecdsa_nistp256_with_sha256:
                EcdsaSignature ecdsa_signature;
            case other_value:
                opaque signature<var>
            }
        } Signature;
```

This structure encodes a signature for a supported public key algorithm. It may be contained within one of a number of data structures: SignedData, SignedWsa, Certificate, CertificateRequest, ToBeEncrypted-CertificateRequestError, Crl. See the definition of each of these structures for a specification of how to set the `algorithm` field for a Signature contained within that structure.

6.2.17 PKAlgorithm

```
enum    {   ecdsa_nistp224_with_sha224 (0),
            ecdsa_nistp256_with_sha256 (1),
            ecies_nistp256 (2),
            private_use (240..255), (2^8-1)
        }   PKAlgorithm;
```

This enumerated type specifies the algorithm of the key in a PublicKey structure. Values 240–255, inclusive, are assigned for private use.

6.2.18 EcdsaSignature

```
struct {
           extern PKAlgorithm  algorithm;
           extern uint8        field_size;
           EllipticCurvePoint        R;
           opaque              s[field_size];
       }   EcdsaSignature;
```

This structure encodes an ECDSA signature. The signature is generated as described in 5.8.1.

If the signature process followed the specification of FIPS 186-3 and output the integer *r*, *r* shall be encoded as an EllipticCurvePoint with type set to `x_coordinate_only`.

If the signature process followed the specification of SEC 1 and output the elliptic curve point *R* to allow for fast verification, *R* shall be encoded as an EllipticCurvePoint with type set to `compressed_lsb_y_0`, `compressed_lsb_y_1`, or `uncompressed` at the sender's discretion.[22]

The value `field_size` is constant for a given PKAlgorithm value. It takes the value given in Table 2. This structure is undefined when PKAlgorithm takes a value not shown in Table 2 because the `field_size` is indeterminate.

Table 2—field_size for different values of PKAlgorithm

PKAlgorithm value	field_size
ecdsa_nistp224_with_sha224	28
ecdsa_nistp256_with_sha256	32

6.2.19 EllipticCurvePoint

```
struct {
           extern PKAlgorithm  algorithm;
           extern uint8        field_size;
           EccPublicKeyType    type;
           opaque              x[field_size];
           select(type) {
           case uncompressed:
              opaque           y[field_size];
           }
       } EllipticCurvePoint;
```

This structure specifies an Elliptic Curve Point. It represents one of the following:

— A public key for either ECDSA or elliptic curve integrated encryption scheme (ECIES), jointly referred to as Elliptic Curve Cryptography (ECC).

— The temporary signature value R in an ECDSA signature.

[22] The compressed forms give some performance advantage on verification compared to the `x_coordinate_only` form, at the same packet size as the `x_coordinate_only` form; the uncompressed form gives a greater performance advantage at the cost of increased packet size.

ECC public keys are points on an elliptic curve. An Elliptic Curve Point shall be encoded with the elliptic curve point encoding and decoding methods defined in 5.5.6 in IEEE Std 1363-2000. Equivalently, the x-coordinate shall be encoded as an unsigned integer in network byte order and the encoding of the y-coordinate y shall depend on whether the point is compressed or uncompressed. If the point is compressed, the value of `type` depends on the LSB of y: if the LSB of y is 0, type shall take the value `compressed_lsb_y_0`, and if the LSB of y is 1, type shall take the value `compressed_lsb_y_1`. If the point is uncompressed, y shall be encoded explicitly as an unsigned integer in network byte order.

The value `field_size` is constant for a given PKAlgorithm value. It takes the value given in Table 2. This structure is undefined when PKAlgorithm takes a value not shown in Table 2 because the `field_size` is indeterminate.

6.2.20 EccPublicKeyType

```
enum  {  x_coordinate_only (0), compressed_lsb_y_0 (2),
         compressed_lsb_y_1 (3), uncompressed (4),
         (255)
      } EccPublicKeyType;
```

This enumerated type specifies the type of an EllipticCurvePoint.[23]

6.2.21 SignedWsa

```
struct {
           SignerIdentifier     signer;
           ToBeSignedWsa        unsigned_wsa;
           Signature            signature;
       } SignedWsa;
```

In this structure:

— `signer` determines the keying material and hash algorithm used to sign the WSA. For a SignedWsa, `signer.type` shall be equal to `certificate_chain`.

— `unsigned_wsa` contains the WSA data.

— `signature` contains the digital signature itself, calculated over the hash of `unsigned_wsa`. The `extern` value `algorithm` in `signature` is equal to `signer.certificates[n].unsigned_certificate.verification_key.algorithm`, where "`signer.certificates[n]`" denotes the last certificate in `signer.certificates`. For this version of this standard, `algorithm` is always equal to `ecdsa_nistp256_with_sha256`.

6.2.22 ToBeSignedWsa

```
struct {
           uint8                       permission_indices<var>;
           TbsDataFlags                tf;
           opaque                      data<var>;
           Time64WithStandardDeviation     generation_time;
```

[23] These values are non-consecutive so that when encoded they align with the encoding specified in IEEE Std 1363 for elliptic curve keys over prime fields.

```
            Time64                  expiry_time;
            ThreeDLocation          generation_location;
            if_flag_set (tf, extensions) {
                TbsDataExtension    extensions<var>;
            }
            if_other_value_set (tf) {
                opaque              other_data<var>;
            }
        } ToBeSignedWsa;
```

In this structure:

— `permission_indices` is used by a receiver to determine the Service Specific Permissions (which are encoded in the certificate) that are applicable to each offered service in the WSA. As detailed in IEEE Std 1609.3, a WSA may contain ServiceInfo fields, each of which announces the availability of a service. For each ServiceInfo in the WSA, there is a corresponding index in `permission_indices` that indicates which entry in the certificate corresponds to that ServiceInfo. The list of integers in `permission_indices` provides this mapping as follows. Denote the *i*th ServiceInfo field in the received WSA by `SI`[*i*], with *i* starting at 1. Denote the *j*th entry in the PsidPrioritySspArray in the received WSA certificate by `permissions`[*j*] , with *j* starting at 1. Then the PsidPrioritySsp that corresponds to `SI`[*i*] is `permissions`[`permission_indices` [*i*]]. If `permission_indices` [*i*] is 0, that indicates that the *i*th ServiceInfo has no corresponding entry in the `permissions` field of the certificate and should not be considered to be secured.

  An implementation of this structure shall support processing a `permission_indices` field that contains 32 or fewer entries.

  For examples of the mapping between a set of ServiceInfos and a set of permissions using `permission_indices`, see D.4.

— `tf` may contain the flags `extensions(4)` indicating that the `extensions` field is included. No other flag values are supported for `tf` in this version of this standard.

— `data` contains the WSA data.

— `generation_time` indicates the time at which the structure was generated. See 7.3.2 for discussion of the meaning of this field.

— `expiry_time` contains the time after which the data should no longer be considered relevant. It shall be strictly later than `generation_time`.

— `generation_location` contains the location at which the signature was generated.

— `extensions`, if present, contains additional data that may be used by the receiver. This field is provided for future extensibility and generation of ToBeSignedWsa structures containing extensions is not conformant with this version of this standard.

6.2.23 EncryptedData

```
struct {
            SymmAlgorithm           symm_algorithm;
            RecipientInfo           recipients<var>;
            select(symm_algorithm) {
            case aes_128_ccm:
                AesCcmCiphertext    ciphertext;
```

```
            case other_value:
                opaque              ciphertext<var>;
            }
        } EncryptedData;
```

This data structure supports encrypting data to one or more recipients using the recipients' public keys. The data is encrypted with a fresh symmetric key generated by the sender, and the symmetric key is then encrypted for each recipient separately using that recipient's public key.

The type contains the following fields:

— `symm_algorithm` contains an identifier for the symmetric algorithm that was used to encrypt the data. This field is provided primarily for future expansion; the only supported value in this version of the standard is `aes_128_ccm`, indicating that the data was encrypted with the Advanced Encryption Standard (AES) using a 128-bit key in Counter with cipher block chaining message authentication code (CCM) mode as described in 5.8.3. The SymmAlgorithm value here shall also appear in the `supported_symm_algorithms` field of all of the certificates identified in the `cert_id` fields of the `recipients`.

— `recipients` contains one or more `RecipientInfos`, defined below. A conformant implementation shall support issuing and receiving EncryptedDatas with 1-6 recipients, and may support more.

— The `ciphertext` field contains the encrypted data. This shall be the encryption of an encoded ToBeEncrypted structure.

6.2.24 SymmAlgorithm

```
enum   {   aes_128_ccm (0), private_use (240..255), (2^8-1) }
           SymmAlgorithm;
```

This enumerated type specifies the symmetric algorithm that shall be used with a public key for encryption. Values 240–255, inclusive, are assigned for private use.

6.2.25 RecipientInfo

```
struct {
            HashedId8                          cert_id;
            extern PKAlgorithm                pk_encryption;
            select (pk_encryption)  {
            case ecies_nistp256:
                EciesNistP256EncryptedKey     enc_key;
            case other_value:
                opaque                        enc_key<var>
            }
        } RecipientInfo;
```

This data structure is used to transfer the symmetric key to an individual recipient of an EncryptedData. It contains the following fields:

— `cert_id` contains the following:

— If the symmetric key was encrypted using a certificate, this field contains the HashedId8 of the certificate that contains the public key used to encrypt the symmetric key.

— If the symmetric key was encrypted using a public key, this field contains the low-order eight octets of the hash of the encoded public key in compressed form.

— `pk_encryption` is set equal to the `algorithm` field of the encryption key in the referenced certificate. See 6.3.34 for more information.

— The final field contains the encrypted key. The format of this field depends on `pk_encryption`. This standard only supports one `pk_algorithm`, `ecies_nistp256`, and only one format for the encrypted key, the EciesNistP256EncryptedKey type.

6.2.26 EciesNistP256EncryptedKey

```
struct {
            extern uint32     symm_key_len;
            EllipticCurvePoint      v;
            opaque            c[symm_key_len];
            opaque            t[20];
        }   EciesNistP256EncryptedKey;
```

This data structure is used to transfer a symmetric key encrypted using ECIES as specified in IEEE Std 1363a-2004. The type contains the following fields:

— `symm_key_len` is the length of the key for the symmetric algorithm identified in the `symm_algorithm` field of the containing EncryptedData. In this standard, the only `symm_algorithm` value supported is `aes_128_ccm` and the corresponding key length shall be 16 octets.

— `v` is the sender's ephemeral public key, which is the output *V* from encryption as specified in 5.8.2. The extern `field_size` value in the EllipticCurvePoint shall be set to `ecies_nistp256`.

— `c` is the encrypted symmetric key, which is the output *C* from encryption as specified in 5.8.2, of length `symm_key_length`.

— `t` is the authentication tag, which is the output *tag* from encryption as specified in 5.8.2. It shall be of length 20 for `ecies_nistp256`.

Encryption and decryption are carried out as specified in 5.8.2.

6.2.27 AesCcmCiphertext

```
struct {
            opaque          nonce[12];
            opaque          ccm_ciphertext<var>;
        }   AesCcmCiphertext;
```

This data structure encapsulates an encrypted ciphertext. It contains the following fields:

— `nonce` contains the nonce *N* as specified in 5.8.5.

— `ccm_ciphertext` contains the ciphertext *C* as specified in 5.8.5.

6.2.28 ToBeEncrypted

```
struct {
            ContentType                     type;
```

```
        select(type) {
        case unsecured:
            opaqueExtLength           plaintext;
        case signed, signed_external_payload,
             signed_partial_payload :
            SignedData              signed_data;
        case certificate_request :
            CertificateRequest        request;
        case certificate_response :
            ToBeEncryptedCertificateResponse
                                      response;
        case anonymous_certificate_response :
            ToBeEncryptedAnonymousCertResponse
                                      anon_response;
        case certificate_request_error:
            ToBeEncryptedCertificateRequestError
                                      request_error;
        case crl_request :
            CrlRequest                crl_request;
        case crl :
            Crl                       crl;
        case certificate_response_acknowledgment:
            ToBeEncryptedCertificateResponseAcknowledgment
                                      ack;
        case other_value:
            opaque                    data<var>;
        }
    }   ToBeEncrypted;
```

This data structure encodes data before it is encrypted. It contains the following fields:

— `type` contains a ContentType value indicating what action, if any, the security services shall take if the data decrypts successfully. A conformant implementation shall not create or parse a ToBeEncrypted structure that uses a ContentType value other than the ones listed in this clause. For example, a conformant implementation shall not create or parse a ToBeEncrypted structure that has a content type of `signed_wsa`. The `ToBeEncryptedAnonymousCertResponse` type is not defined in this version of this standard and is included for future extensibility.

— The remainder of the ToBeEncrypted contains the content.

— If type = unsecured, plaintext contains the plaintext. If the ToBeEncrypted was extracted from an AesCcmCiphertext, the plaintext length is the length of the ccm_ciphertext field minus 17 (16 octets for the authentication tag in the ciphertext, and 1 for the type field in the ToBeEncrypted).

— For all other values of `type`, the content is a structured 1609.2 data type as specified in this standard.

6.3 Certificates and other security management data structures

6.3.1 General

This clause specifies the structures to be used for certificates and security management. Examples are provided in D.1.

6.3.2 Certificate

```
struct {
            uint8                       version_and_type;
            ToBeSignedCertificate       unsigned_certificate;
            select (version_and_type) {
            case 2:
                extern PKAlgorithm      algorithm;
                Signature               signature;
            case 3 :
                EllipticCurvePoint            reconstruction_value;
            case other_value :
                opaque                  signature_material<var>;
            }
        } Certificate;
```

The fields in this structure have the following meaning:

— `version_and_type` contains the version of the certificate format and whether the certificate is implicit or explicit. In this version of the protocol, this field shall be set to 2 for explicit certificates and to 3 for implicit certificates.

— `unsigned_certificate` is the structure described below.

— `signature` is included if the certificate explicitly contains the holder's public signing key, in other words, if `version_and_type` is 2. It is the signature, calculated by the signer identified in the `signer_id` field of the `unsigned_certificate`, over the hash of the encoding of the `version_and_type` and `unsigned_certificate` fields. The `extern` value `algorithm` in `signature` is determined as follows:

— If `unsigned_certificate.holder_type` is `root_ca`, `algorithm` is `unsigned_certificate.verification_key.algorithm`.

— If `unsigned_certificate.holder_type` is any other value, `algorithm` is `unsigned_certificate.signature_alg`.

— `reconstruction_value` is included if the certificate is an *implicit certificate* that does not explicitly contain the holder's public signing key, in other words, if `version_and_type` is 3. It is the reconstruction value calculated by the CA and used by a verifier to recover the signer's public key. The `extern` value `algorithm` in `reconstruction_value` is `unsigned_certificate.signature_alg`.

When a signer or verifier calculates the hash of the certificate data, the hash shall be calculated using the compressed representation of all public keys and reconstruction values, even if the certificate includes the uncompressed form.

When an implicit certificate is encoded for hashing as described in 3.2.1 in SEC 4:

— The encoded data input to the hash function shall be the PublicKeyReconstructionHashInput described in this section.

— SHA-256 shall be used as the Hash algorithm H within the integer hash Hn specified in 2.3 in SEC 4.

6.3.3 ToBeSignedCertificate

```
struct {
            extern uint8            version_and_type;
            HolderType              holder_type;
            CertificateContentFlags cf;
            select (holder_type) {
            case root_ca:  ;
            case sde_ca, crl_signer, wsa, wsa_ca,
                 wsa_enrolment, sde_enrolment,
                 sde_identified_localized,
                 sde_identified_not_localized,
                 sde_anonymous, other_value:
                 HashedId8            signer_id;
                 PKAlgorithm         signature_alg;
            };
            CertSpecificData        scope;
            Time32                  expiration;
            if_set(cf, use_start_validity) {
                if_set(cf, lifetime_is_duration) {
                    CertificateDuration lifetime;
                }
                if_not_set(cf, lifetime_is_duration) {
                    Time32              start_validity;
                }
            }
            CrlSeries               crl_series;
            select (version_and_type) {
            case 2:
                PublicKey               verification_key;
            case 3:;
            case other_value:
                opaque                  other_key_material<var>;
            }
            if_set (cf, encryption_key) {
                PublicKey           encryption_key;
            };
            if_other_value_set (cf) {
                opaque              other_cert_content<var>;
            };
        } ToBeSignedCertificate;
```

The fields in the ToBeSignedCertificate structure have the following meaning:

— `version_and_type` is obtained from the encapsulating Certificate structure. It shall take the value 2 or 3.

— `holder_type` specifies the structure of the rest of the ToBeSignedCertificate. It is used to state whether the signer_id is present and how the scope field is to be interpreted. If `version_and_type` is 2, `holder_type` may take any value. If `version_and_type` is 3 (meaning implicit certificates), `holder_type` shall not take the value `root_ca`.

— `cf` specifies whether or not additional optional fields are present.

— `signer_id`, if present, contains the low order 8 octets of the SHA-256 hash of the issuing Certificate.

— `signature_alg`, if present, identifies the algorithm that was used to sign or create the reconstruction value in the Certificate. If `version_and_type` is 3, this shall be `ecdsa_nistp256_with_sha256`.

— `scope` contains data that is appropriate to the specific `holder_type`. The different `scope` types are described below.

— `expiration` gives the last Time32 on which the Certificate is valid. If the `expiration` is zero, the Certificate does not expire. A ToBeSignedCertificate shall have a non-zero value in at least one of the `expiration` field and the `crl_series` field.

— If `cf` contains the flag `use_start_validity`:

— If `cf` contains the flag `lifetime_is_duration`: the Certificate contains the field `lifetime`. The field gives the lifetime of the Certificate in the format described in the description of the CertificateDuration. The Certificate is valid starting at the time given by (`expiration` - `lifetime`).[24] `cf` shall not contain the flag `use_start_validity` if `expiration` is 0.

— Otherwise, the Certificate contains the field `start_validity`. The Certificate is valid starting at the time `start_validity`. If `start_validity` and `expiration` are both present and not both 0, `start_validity` shall be strictly earlier than `expiration`.

— `crl_series` contains an integer that allows a CA to partition its issued certificates into groups that appear on different CRLs, to allow the CA to manage CRL sizes as follows. A ToBeSignedCertificate contains a `crl_series` field. The Crl structure defined in 6.3.40 also contains a `crl_series` field (within the ToBeSignedCrl structure). If a certificate with `crl_series` value *c* is revoked, that information appears on a Crl whose `crl_series` value is *c*. A CA will therefore maintain c_{max} distinct series of CRLs, where c_{max} is the maximum value of `crl_series` in any certificate issued by that CA.

  The value 0 in this field indicates that the certificate shall not appear on a Crl. Such a certificate remains valid until it expires. A valid Certificate shall have a non-zero value in at least one of the `expiration` field and the `crl_series` field.

— `verification_key` contains the public key to be used to verify signatures generated by the holder of the Certificate. This key shall have its `algorithm` field set equal to either `ecdsa_nistp256_with_sha256` or `ecdsa_nistp224_with_sha224`. If the holder type of the certificate is any of the CA types (`root_ca`, `sde_ca`, `crl_signer`, `wsa_ca`), the key shall be for ECDSA-256 and shall have the algorithm field set equal to `ecdsa_nistp256_with_sha256`.

— If `cf` contains the value `encryption_key`, the `encryption_key` shall be present. This field shall contain a public key for encryption. This key shall have its `algorithm` field set equal to `ecies_nistp256`.

6.3.4 HolderType

```
enum   {  sde_anonymous(0),
          sde_identified_not_localized (1),
          sde_identified_localized (2),
          sde_enrolment (3)
          wsa (4),
          wsa_enrolment (5),
          sde_ca(6), wsa_ca (7), crl_signer(8),
```

[24] The use of `lifetime_is_duration` is recommended to reduce bandwidth.

```
        ...
        private_use (240..254),
        root_ca (255),
        (2^8-1)
    }   HolderType;
```

This enumerated field is used to determine which fields are included in Certificates and other security management data structures. Values 240–254, inclusive, are assigned for private use. HolderType values beginning with `sde_` are associated with secure data exchange entities. HolderType values beginning with `wsa_` are associated with secured WSAs. The values identify different types of scope used within the CertSpecificData structure; see the definition of that structure for more details.

6.3.5 CertificateContentFlags

```
flags  {  use_start_validity (0), lifetime_is_duration(1),
          encryption_key (2)
       } CertificateContentFlags;
```

This flags type is used to determine which fields are included in Certificates.

If `use_start_validity` is set, the ToBeSignedCertificate contains a `start_validity` time or a `lifetime`. If `use_start_validity` is not set, the ToBeSignedCertificate contains only an expiry time.

If `encryption_key` is set, the ToBeSignedCertificate contains a public key for encryption.

6.3.6 CertificateDuration

```
uint16  CertificateDuration;
```

This integer encodes the duration of validity of a certificate. The first three bits encode the units, as given in Table 3. The remaining 13 bits are to be treated as an unsigned integer encoded in network byte order and denote the lifetime of the certificate as a number of the units given by the first three bits.

Table 3—Interpretation of units for CertificateDuration

bits	interpretation
000	seconds
001	minutes (= 60 s)
010	hours (= 3600 s)
011	60-hour blocks (= 3600 min)
100	“year”s (= 31 556 925 s)
101, 110, 111	undefined

Examples:

— The octets 0x03 0xe8 (bits `000|0 0011 1110 1000`) correspond to 1000 s.

— The octets 0x5f 0xff (bits 010 | 1 1111 1111 1111) correspond to 8191 hours, about 341.3 days.

6.3.7 CertSpecificData

```
struct {
            extern HolderType holder_type;
            select(holder_type){
            case root_ca:
                RootCaScope             root_ca_scope;
            case sde_ca, sde_enrolment:
                SecDataExchCaScope      sde_ca_scope;
            case wsa_ca, wsa_enrolment:
                WsaCaScope              wsa_ca_scope;
            case crl_signer:
                CrlSeries               responsible_series<var>;
            case sde_identified_not_localized:
                IdentifiedNotLocalizedScope  id_non_loc_scope;
            case sde_identified_localized,
                IdentifiedScope         id_scope;
            case sde_anonymous:
                AnonymousScope          anonymous_scope;
            case wsa:
                WsaScope                wsa_scope;
            case other_value:
                opaque                  other_scope<var>;
            }
        } CertSpecificData;
```

This structure contains data that is unique to the relevant certificate holder_type. The different scope types are described below.

A certificate of type crl_signer indicates that the holder of this certificate is entitled to sign CRLs for this certificate authority. The responsible_series array contains all of the CRL series which this CRL signer is authorized to sign.

6.3.8 RootCaScope

```
struct {
            uint8               name<var>;
            HolderTypeFlags    permitted_holder_types;
            if_any_set (permitted_holder_types,
                sde_ca,
                sde_enrolment,
                sde_identified_localized,
                sde_identified_not_localized,
                sde_anonymous) {
                PsidArray          secure_data_permissions;
            }
            if_any_set (permitted_holder_types, wsa_ca,
                wsa_enrolment, wsa) {
                PsidPriorityArray wsa_permissions;
            }
            if_other_value_set (permitted_holder_types) {
                opaque             other_permissions<var>;
```

```
            }
            GeographicRegion  region;
        } RootCaScope;
```

This data structure defines a root CA's permissions.

— `name` identifies the CA. Its contents are a matter of policy and should consist of a human-readable string, encoded according to the Unicode Transformation Format (UTF)-8 encoding rules of IETF RFC 3629. This field may be empty (of length zero, encoded as `00`). Its maximum length shall be 32 octets.

— `permitted_holder_types` lists the holder types for which this CA is authorized to issue certificates. It shall contain at least one HolderTypeFlags value.

— If present, `secure_data_permissions` indicates the PSIDs for which this CA is authorized to issue certificates (the CA's "permitted PSIDs"). The `secure_data_permissions.type` field shall not be `from_issuer`. Secure data exchange certificates (certificates with holder type `sde_ca`, `sde_enrolment`, `sde_identified_localized`, `sde_identified_not-_localized`, `sde_anonymous`) issued, directly or indirectly, by this CA shall not include PSIDs that are not specified in this list.

— If present, `wsa_permissions` indicates the WSA PSIDs and priorities for which this CA is authorized to issue certificates (the CA's "WSA permissions"). The `wsa_permissions.type` field shall not be `from_issuer`. WSA certificates (certificates with holder type `wsa_ca`, `wsa_enrolment`, `wsa`) issued, directly or indirectly, by this CA shall not include PSIDs that are not specified in this list. In any WSA certificate issued, directly or indirectly by this CA, the max_priority value in each PsidPriority shall be no greater than the max_priority value in the PsidPriority field that contains the same PSID value.

— `region` indicates the area for which the CA is allowed to issue certificates. If this field has `region_type` = `none`, it indicates that the CA is valid worldwide. The `region` field shall not contain `from_issuer`.

6.3.9 HolderTypeFlags

```
flags  {  sde_anonymous(0),
          sde_identified_not_localized (1),
          sde_identified_localized (2),
          sde_enrolment (3)
          wsa (4),
          wsa_enrolment (5),
          sde_ca(6), wsa_ca (7), crl_signer(8)
       } HolderTypeFlags;
```

This flags type identifies the HolderType values that a CA is allowed to include in certificates it issues.

6.3.10 PsidArray

```
struct {
           ArrayType       type;
           select(type) {
           case specified:
               Psid        permissions_list<var>;
```

```
            case from_issuer: ;
            case other_value:
                opaque      other_permissions<var>;
            };
        } PsidArray;
```

This structure encodes a list of PSIDs for use in a CA or enrolment certificate.

If `type` is `from_issuer`, it indicates that the PSIDs permitted by a certificate are the same as the PSIDs permitted by the issuing certificate. In this case, the certificate is said to have "inherited [its] PSIDs." An algorithm for establishing the inherited PSIDs is given as part of the processing specification for CME-CertificateInfo.request.

If `type` is `specified`, the PSIDs permitted by the certificate are encoded in the `psids` field. If this field is empty (encoded as 0x00, indicating a zero-length data field), it indicates for a CA certificate that the CA may issue certificates for any PSID and for a enrolment certificate that the certificate holder may request certificates for any PSID. The array shall not contain duplicate PSIDs.

An implementation of this structure shall support processing a `permissions_list` field that contains 8 or fewer entries. An implementation of this structure may support processing a `permissions_list` field that contains more than 8 entries.

6.3.11 ArrayType

```
enum {from_issuer(0), specified(1), … (2^8-1)} ArrayType;
```

This type specifies whether an array is specified explicitly or obtained by reference.

6.3.12 PsidPriorityArray

```
struct {
            ArrayType             type;
            select(type) {
            case specified:
                PsidPriority      permissions_list<var>;
            case from_issuer: ;
            case other_value:
                opaque      other_permissions<var>;
            };
        } PsidPriorityArray;
```

This structure encodes a list of PSIDs and their associated priorities for use in a CA or enrolment certificate to authorize WSAs.

If `type` is `from_issuer`, it indicates that the WSA is permitted to be authorized by a certificate are the same as the WSAs permitted to be authorized by the issuing certificate. In this case, the certificate is said to have "inherited [its] permissions." An algorithm for establishing the inherited PSIDs is given as part of the processing specification for CME-CertificateInfo.request.

If `type` is `specified`, the PSIDs and priorities permitted by the certificate are encoded in the `permissions` field. If this field is empty (encoded as 0x00, indicating a zero-length data field), it

indicates for a CA certificate that the CA is authorized to issue certificates for any PSID at any priority and for a enrolment certificate that the holder is authorized to request certificates for any PSID at any priority. All of the PSIDs in the array shall be distinct.

An implementation of this structure shall support processing a `permissions_list` field that contains 8 or fewer entries. An implementation of this structure may support processing a `permissions_list` field that contains more than 8 entries.

6.3.13 PsidPriority

```
struct {
            Psid        psid;
            uint8       max_priority;
        } PsidPriority;
```

This data structure identifies a pair of Psid and maximum priority for use in a CA certificate within a chain that issues WSA signing certificates. The use of the priority is described in IEEE Std 1609.3.

6.3.14 GeographicRegion

```
struct {
            RegionType              region_type;
            select(region_type){
            case from_issuer: ;
            case circle:
                CircularRegion      circular_region;
            case rectangle:
                RectangularRegion   rectangular_region<var>;
            case polygon:
                PolygonalRegion     polygonal_region;
            case none: ;
            case other_value:
                opaque              other_region<var>;
            }
        } GeographicRegion;
```

This structure encodes a geographic region of a specified form.

If `type` is `from_issuer`, it indicates that the geographic region of validity of the current certificate is the same as the geographic region of validity of the issuing certificate. In this case, the certificate is said to have “inherited [its] permissions.” An algorithm for establishing the inherited geographic region is given as part of the processing specification for CME-CertificateInfo.request.

The `rectangular_region` field is an array of RectangularRegion structures containing at least one RectangularRegion. This field is interpreted as a series of rectangles, which may overlap or be disjointed. The permitted region is any point within any of the rectangles. An implementation of this structure shall support processing a `rectangular_region` field that contains 6 or fewer entries. An implementation of this structure may support processing a `rectangular_region` field that contains more than 6 entries.

The `circular_region` and `polygonal_region` fields contain a single instance of their respective structures.

If `region_type` is none, the certificate does not have any geographic restrictions.

A certificate is not valid if any part of the region encoded in its scope field lies outside the region encoded in the scope of its issuer.

An implementation that supports generating or verifying signed communications shall support at least one of CircularRegion, RectangularRegion, and PolygonalRegion.

6.3.15 RegionType

```
enum   {from_issuer(0), circle(1), rectangle(2), polygon(3),
           none (4), ..., private_use (240..255),
           (2^8-1)
       } RegionType;
```

This enumerated type indicates how a GeographicRegion type should be interpreted. It may take the values `from_issuer(0)`, `circle(1)`, `rectangle(2)`, `polygon(3)`, and `none(4)`. Values 240–255, inclusive, are assigned for private use.

6.3.16 CircularRegion

```
struct {
           TwoDLocation     center;
           uint16         radius;
       } CircularRegion;
```

This structure specifies a circle with its center at the center point, its radius value in meters, and located tangential to the reference ellipsoid. The indicated region shall be all the points on the surface of the reference ellipsoid whose distance to the center point over the reference ellipsoid is less than or equal to the radius. A point that contains an elevation component is considered to be within the circular region if its horizontal projection onto the reference ellipsoid lies within the region.

6.3.17 RectangularRegion

```
struct {
           TwoDLocation    north_west;
           TwoDLocation    south_east;
       } RectangularRegion;
```

This structure specifies a rectangle formed by connecting in sequence: (`north_west.latitude`, `north_west.longitude`), (`south_east.latitude`, `north_west.longitude`), (`south_east.latitude`, `south_east.longitude`), and (`north_west.latitude`, `south_east.longitude`). The points are connected by lines of constant latitude or longitude. A point that contains an elevation component is considered to be within the rectangular region if its horizontal projection onto the reference ellipsoid lies within the region. A RectangularRegion is valid only if the `north_west` value is north and west of the `south_east` value (i.e., the two points cannot have equal `latitude` or equal `longitude`).

6.3.18 PolygonalRegion

```
TwoDLocation PolygonalRegion<var>;
```

This data structure defines a region using a series of distinct geographic points, defined on the surface of the reference ellipsoid. The region is specified by connecting the points in the order they appear, with each pair of points connected by the geodesic on the reference ellipsoid. The polygon is completed by connecting the final point to the first point. The allowed region is the interior of the polygon and its boundary.

A point that contains an elevation component is considered to be within the polygonal region if its horizontal projection onto the reference ellipsoid lies within the region.

A valid PolygonalRegion contains at least three points. The implied lines that make up the sides of the polygon shall not intersect. An implementation of this standard that supports PolygonalRegions shall support processing polygonal regions containing 12 or fewer points. An implementation may support processing polygonal regions containing more than 12 points.

6.3.19 TwoDLocation

```
struct {
            sint32          latitude;
            sint32          longitude;
        } TwoDLocation;
```

This data structure is used to define validity regions for use in certificates. The `latitude` and `longitude` fields contain the latitude and longitude as a `sint32` type, encoding the latitude and longitude in microdegrees. The encoded integer in the `latitude` field shall be no more than 900 000 000 and no less than –900 000 000. The encoded integer in the `longitude` field shall be no more than 1 800 000 000 and no less than –1 800 000 000. All values are relative to the WGS84 datum.

NOTE—This data structure is consistent with the location encoding used in [B21], except that values 900 000 001 for latitude (used to indicate that the latitude was not available) and 1 800 000 001 for longitude (used to indicate that the longitude was not available) are not valid.

6.3.20 SecDataExchCaScope

```
struct {
            uint8               name<var>;
            HolderTypeFlags     permitted_holder_types;
            PsidArray           permissions;
            GeographicRegion    region;
        } SecDataExchCaScope;
```

This scope type is contained in a Secure Data Exchange CA certificate or an enrolment certificate for a secure data exchange entity.

In a Secure Data Exchange CA certificate, this scope indicates that certificates issued by that CA are only valid if they are consistent with this scope as defined in 5.5.3.3.

In a Secure Data Exchange enrolment certificate, this scope indicates that certificate requests issued by the holder are only valid if they are consistent with this scope as defined in 5.6.1.2.

— `name` identifies the certificate holder. Its contents are a matter of policy and should consist of a human-readable string, encoded according to the UTF-8 encoding rules of IETF RFC 3629. This field may be empty (of length zero, encoded as `00`). Its maximum length shall be 32 octets.

— `permitted_holder_types` lists the holder types for which the CA is authorized to issue certificates, or for which the CSR sender is authorized to request certificates. It shall contain at least one HolderTypeFlags value. It shall only include HolderTypeFlags values drawn from the following list: `sde_ca`, `sde_enrolment`, `sde_identified_localized`, `sde_identified_not_localized`, `sde_anonymous`, `crl_signer`.

— `permissions` indicates the PSIDs for which the CA is authorized to issue certificates, or for which the CSR sender is authorized to request certificates. The `permissions.type` field may be `from_issuer`.

— `region` indicates the area for which the CA is allowed to issue certificates, or for which the CSR sender is authorized to request certificates. If this field has `region_type = none`, it indicates that the certificate is valid worldwide. If `region.type` is `from_issuer`, the certificate inherits its permissions from the issuing certificate as described in 5.5.3.4. A certificate is not valid if any part of the region encoded in its scope field lies outside the region encoded in the scope of its issuer.

6.3.21 WsaCaScope

```
struct {
            opaque              name<var>;
            PsidPriorityArray   permissions;
            GeographicRegion    region;
        } WsaCaScope;
```

This scope type is contained in a WSA CA certificate or an enrolment certificate used to request WSA signing certificates.

In a WSA CA certificate, this scope indicates that certificates issued by that CA are only valid if they are consistent with this scope as defined in 5.5.3.3.

In a WSA signing enrolment certificate, this scope indicates that certificate requests issued by the holder are only valid if they are consistent with this scope as defined in 5.6.1.2.

— `name` identifies the certificate holder. Its contents are a matter of CA policy and should consist of a human-readable string, encoded according to the UTF-8 encoding rules of IETF RFC 3629. This field may be empty (of length zero, encoded as 0x00). Its maximum length shall be 32 octets.

— `permissions` indicates the WSA PSIDs and priorities for which this CA is authorized to issue certificates, or for which the CSR sender is authorized to request certificates (the "WSA permissions").

— `region` indicates the area for which the CA is allowed to issue certificates, or for which the CSR sender is authorized to request certificates. If this field has `region_type = none`, it indicates that the certificate is valid worldwide. If `region.type` is `from_issuer`, the certificate inherits its permissions from the issuing certificate as described in 5.5.3.4. A certificate is not valid if any part of the region encoded in its scope field lies outside the region encoded in the scope of its issuer.

6.3.22 CrlSeries

```
uint32 CrlSeries;
```

This type identifies the CRL series that a given certificate may appear on, or the CRL series that a CRL Signer may issue.

6.3.23 IdentifiedNotLocalizedScope

```
struct {
            opaque              name<var>;
            PsidSspArray        permissions;
        } IdentifiedNotLocalizedScope;
```

This scope is used to indicate that the certificate holder is allowed to take the actions associated with its PSIDs and SSPs, and has the same geographic restrictions as its issuer.

— `name` identifies the certificate holder. Its contents are a matter of CA policy and should consist of a human-readable string, encoded according to the UTF-8 encoding rules of IETF RFC 3629. This field may be empty (of length zero, encoded as 0x00). Its maximum length shall be 32 octets.

— `permissions` contains a description of the communications that the certificate holder is authorized to send. This takes the form of a list of the PSIDs and SSPs. These permissions are used in the consistency checks described in 5.5.3.

6.3.24 PsidSspArray

```
struct {
            ArrayType           type;
            select(type) {
            case specified:
                PsidSsp         permissions_list<var>;
            case from_issuer: ;
            case other_value:
                opaque          other_permissions<var>;
            };
        } PsidSspArray;
```

This structure encodes a list of PSIDs and the associated SSPs for use in a certificate.

If `type` is `from_issuer`, it indicates that the PSIDs permitted by a certificate are the same as the PSIDs permitted by the issuing certificate. In this case, the certificate is said to have "inherited [its] permissions." An algorithm for establishing the inherited PSIDs is given as part of the processing specification for CME-CertificateInfo.request. Since CA certificates do not contain SSPs, an inherited PSID shall be considered to have no SSP. The interpretation of no SSP is PSID-specific.

If `type` is `specified`, the certificate holder's permissions are encoded in the `permissions_list` field. A valid certificate shall contain at least one PsidSsp. In a valid certificate, all of the `psid` values in the PsidSsp fields in the array shall be distinct. CAs shall not issue an invalid certificate.

An implementation of this structure shall support processing a `permissions_list` field that contains 8 or fewer entries. An implementation of this structure may support processing a `permissions_list` field that contains more than 8 entries.

6.3.25 PsidSsp

```
struct {
            Psid        psid;
            opaque      service_specific_permissions<var>;
        } PsidSsp;
```

This structure encodes a Psid and the associated Service Specific Permissions (SSP) field. The service_specific_permissions field shall have a length no more than 32 octets. If a Psid does not have an associated SSP field the service_specific_permissions in this structure shall be encoded as a zero-length field (0x00).

6.3.26 IdentifiedScope

```
struct {
            opaque              name<var>;
            PsidSspArray        permissions;
            GeographicRegion    region;
        } IdentifiedScope;
```

This scope is used to indicate that the certificate holder is allowed to send the communications described by its PSIDs and SSPs; is authorized to operate within a certain area; and may have a unique identifying name.

— name identifies the certificate holder. Its contents are a matter of CA policy and should consist of a human-readable string, encoded according to the UTF-8 encoding rules of IETF RFC 3629. This field may be empty (of length zero, encoded as 0x00). Its maximum length shall be 32 octets.

— permissions contains a description of the communications that the certificate holder is authorized to send. This takes the form of a list of the PSIDs and SSPs. These permissions are used in the consistency checks described in 5.5.3.

— region indicates the geographic region within which the certificate holder is allowed to operate. If it contains the from_issuer region value, it indicates that the certificate holder is authorized to operate anywhere that its CA certificate is accepted. A certificate is not valid if any part of the region encoded in its scope field lies outside the region encoded in the scope of its issuer.

6.3.27 AnonymousScope

```
struct {
            opaque              additional_data<var>;
            PsidSspArray        permissions;
            GeographicRegion    region;
        } AnonymousScope;
```

This scope is used to indicate that the certificate holder is allowed to send the communications described by its PSIDs and SSPs; is authorized to operate within a certain area; and is not intended to be identifiable directly from its certificate.

— additional_data is a field that may contain additional request-specific or response-specific data. This field may be empty (of length zero, encoded as 0x00). Its maximum length shall be 32 octets.

— permissions contains a description of the communications that the certificate holder is authorized to send. This takes the form of a list of the PSIDs and SSPs. These permissions are used in the consistency checks described in 5.5.3.

— region indicates the geographic region within which the certificate holder is allowed to operate. If it contains the from_issuer region value, it indicates that the certificate holder is authorized to operate anywhere that its CA certificate is accepted. A certificate is not valid if any part of the region encoded in its scope field lies outside the region encoded in the scope of its issuer.

6.3.28 WsaScope

```
struct {
            opaque                  name<var>;
            PsidPrioritySspArray    permissions;
            GeographicRegion        region;
        } WsaScope;
```

This scope is used when the certificate holder may broadcast WSAs advertising a specific set of services and is authorized to operate within a certain area. It is used in certificates to support authenticated WSAs and in certificates to allow requests for WSA certificates.

— `name` identifies the certificate holder. Its contents are a matter of CA policy and should consist of a human-readable string, encoded according to the UTF-8 encoding rules of IETF RFC 3629. This field may be empty (of length zero, encoded as 0x`00`). Its maximum length shall be 32 octets.

— `permissions` contains a list of the services that the certificate authorizes the WSA Signer to offer. These permissions are used in the consistency checks described in 5.5.3.

— `region` indicates the geographic region within which the certificate holder is allowed to operate. If it contains the `from_issuer` region value, it indicates that the certificate holder may operate anywhere that its CA certificate is accepted. A certificate is not valid if any part of the region encoded in its scope field lies outside the region encoded in the scope of its issuer.

6.3.29 PsidPrioritySspArray

```
struct {
            ArrayType               type;
            select(type) {
            case specified:
                PsidPrioritySsp     permissions_list<var>;
            case from_issuer: ;
            case other_value:
                opaque              other_permissions<var>;
            };
        } PsidPrioritySspArray;
```

This structure encodes a list of PSIDs and their associated SSPs and priorities for use in a WSA certificate.

If `type` is `from_issuer`, it indicates that the PSIDs and priorities permitted by a certificate are the same as the PSIDs and priorities permitted by the issuing certificate. In this case, the certificate is said to have "inherited [its] PSIDs and priorities." An algorithm for establishing the inherited PSIDs is given as part of the processing specification for CME-CertificateInfo.request. Since CA certificates do not contain SSPs, an inherited PSID shall be considered to have a NULL SSP.

If `type` is `specified`, the PSIDs and priorities permitted by the certificate are encoded in the `permissions_list` field.

An implementation of this structure shall support processing a `permissions_list` field that contains 8 or fewer entries. An implementation of this structure may support processing a `permissions_list` field that contains more than 8 entries.

6.3.30 PsidPrioritySsp

```
struct {
           Psid        psid;
           uint8       max_priority;
           opaque      service_specific_permissions<var>;
       } PsidPrioritySsp;
```

This data structure encodes a Psid and the associated priority and SSP fields for use in a certificate used to sign a WSA. The use of the priority is described in IEEE Std 1609.3. The `service_specific_permissions` field shall have a length no more than 32 octets. If a Psid does not have an associated SSP field the service_specific_permissions in this structure shall be encoded as a zero-length field (0x00).

6.3.31 Time32

```
uint32   Time32;
```

The Time32 type is a 32-bit integer, encoded in big-endian format, giving the number of (TAI) seconds since 00:00:00 UTC, 1 January, 2004.

6.3.32 PublicKey

```
struct {
           PKAlgorithm              algorithm;
           select(algorithm){
           case ecdsa_nistp224_with_sha224:
           case ecdsa_nistp256_with_sha256:
               EllipticCurvePoint         public_key;
           case ecies_nistp256:
               SymmAlgorithm        supported_symm_alg;
               EllipticCurvePoint         public_key;
           case other_value:
               opaque               other_key<var>;
           }
       } PublicKey;
```

This structure encodes a public key and states with what algorithm the public key shall be used. Cryptographic mechanisms are defined in 5.8.

— `public_key` contains the encoded public key.

— `supported_symm_alg` indicates the symmetric algorithm that shall be used with the ECIES public key.

ECDSA-224 and ECDSA-256 are identified by the PKAlgorithm values `ecdsa_nistp224_with_sha224` and `ecdsa_nistp256_with_sha256` respectively.

ECIES-256 is specified with the PKAlgorithm value `ecies_nistp256`. In this case, the `supported_symm_alg` field shall be `aes_128_ccm(0)`.

6.3.33 PublicKeyReconstructionHashInput

```
struct {
           opaque       issuer_hash[32];
           Certificate  holder_certificate;
       } PublicKeyReconstructionHashInput;
```

This structure is used to encode the certificate data of the holder of an implicit certificate, for inputting to the hashing process as described in 5.8.6.

— `issuer_hash` is the SHA-256 hash of the issuing Certificate (i.e., the Certificate identified by the signer_id field in the holder's ToBeSignedCertificate).

— `holder_certificate` is the holder's Certificate. This shall be an implicit certificate (with `version_and_type` = 3).

6.3.34 CertificateRequest

```
struct {
           SignerIdentifier             signer;
           ToBeSignedCertificateRequest unsigned_csr;
           Signature                    signature;
       } CertificateRequest;
```

This data structure is used to request a certificate.

— `signer` determines the keying material and hash algorithm used to sign the data. If `signer.type` is not `self`, then `holder_type` of the end-entity certificate indicated by `signer` shall be either `sde_enrolment` or `wsa_enrolment`.

— `unsigned_data` contains the data.

— `signature` contains a signature over the hash of the `unsigned_csr` value. The key to be used to verify the signature, and the `extern` value `algorithm` in `signature`, are determined as follows:

  — If `signer.type` is equal to `self`:

    — The signature is verified using `unsigned_csr.verification_key`.

    — `algorithm` is equal to `unsigned_csr.verification_key.algorithm`.

  — If `signer.type` is equal to `certificate`:

    — The signature is verified with the public key indicated by `signer.certificate`.

    — If `signer.certificate.version_and_type` is `2`, then `algorithm` is equal to `signer.certificate.unsigned_certificate.verification_key.algorithm`.

    — If `signer.certificate.version_and_type` is `3`, then `algorithm` is equal to `signer.certificate.unsigned_certificate.signature_alg`.

6.3.35 ToBeSignedCertificateRequest

```
struct {
           uint8            version_and_type;
```

```
        Time32           request_time;
        HolderType       holder_type;
        CertificateContentFlags cf;
        CertSpecificData        type_specific_data;
        Time32                  expiration;
        if_set(cf, use_start_validity) {
            if_set(cf, lifetime_is_duration) {
                CertificateDuration lifetime;
            }
            if_not_set(cf, lifetime_is_duration) {
                Time32              start_validity;
            }
        }
        PublicKey               verification_key;
        if_set (cf, encryption_key) {
            PublicKey           encryption_key;
        }
        if_other_value_set (cf) {
            opaque             other_cert_content<var>;
        };
        PublicKey        response_encryption_key;
    } ToBeSignedCertificateRequest;
```

This data structure encodes the details of a certificate signing request.

— `version_and_type` indicates the version of certificate that the requester is requesting and whether the requester is requesting an implicit or explicit certificate. In this version of the protocol, this field shall be set to 2 for explicit certificates and to 3 for implicit certificates.

— `request_time` is the time that the request was formed, measured in (TAI) seconds since the epoch of 00:00:00 UTC, 1 January, 2004.

— `holder_type` specifies the `holder_type` that the requester is requesting.

— `cf` contains the requested CertificateContentFlags and states which optional fields are included in this certificate request.

— `type_specific_data` specifies the desired scope. None of the structures in the `type_specific_data` field shall take the value `from_issuer`.

— `expiration` is the requested expiration time for the certificate. If this field is all zeroes, it indicates that the requester is not requesting a specific expiry time. If this field is not all zeroes, it shall be strictly later than `request_time`.

— If `cf` contains the flag `use_start_validity`:

  — If `cf` contains the flag `lifetime_is_duration`, the field `lifetime` specifies the requested validity lifetime for the certificate. If this field is 0, it indicates that the requester is not requesting a specific lifetime.

  — Otherwise, the certificate contains the field `start_validity`, which specifies the requested start validity time for the certificate. If this field is 0, it indicates that the requester is not requesting a specific start validity time. If `start_validity` and `expiration` are both present and not both 0, `start_validity` shall be strictly earlier than `expiration`.

— `verification_key` contains a public key for a verification algorithm. In a request for an explicit certificate, the expectation is that this key will be given as the `verification_key` in the certificate issued in response to this request. If the certificate request is for an implicit

certificate, the expectation is that this key is used as the ephemeral public key from which the final public key is derived. This key shall have its PKAlgorithm field set equal to either `ecdsa_nistp256_with_sha256` or `ecdsa_nistp224_with_sha224`.

— If `cf` contains the value `encryption_key`, the `encryption_key` field shall be present. This field shall contain a public key for encryption. This key shall have its PKAlgorithm field set equal to an `ecies_nistp256`.

— `response_encryption_key` contains the key to be used to encrypt the response. Each distinct instance of a ToBeSignedCertificateRequest should include a different `response_encryption_key`. See 5.6.1.1 for further discussion.

See 5.6.2.2 for a discussion of consistency between certificate requests and certificate responses.

6.3.36 ToBeEncryptedCertificateResponse

```
struct {
            extern uint8     version_and_type;
            flags            f;
            Certificate      certificate_chain<var>;
            select (version_and_type) {
            case 2: ;
            case 3 :
                extern uint8 field_size;
                opaque       recon_priv[field_size];
            case other_value :
                opaque       other_material<var>;
            }
            Crl              crl_path<var>;
        } ToBeEncryptedCertificateResponse;
```

This data structure is used to form a certificate response.

— The value `version_and_type` is obtained from the end-entity certificate in the response, which is the last certificate in the `certificate_chain` field.

— The flags `f` may encode the single value 0. If the value 0 is encoded, the CA is requesting an acknowledgement in the form of a ToBeEncryptedCertificateResponseAcknowledgment.

— `certificate_chain` contains the certificate path of the new certificate. This path is in order, with the newly issued certificate being last and each preceding certificate signing the one before it. The path should be complete with the first certificate being a trust anchor. If the chain includes implicit certificates, the first certificate in the chain shall either be a trust anchor, or directly signed by a trust anchor, or an explicit certificate (i.e., the first certificate in the chain shall not be an implicit certificate unless it was signed by a trust anchor).

— If `version_and_type` is 3, then the end-entity certificate is an implicit certificate. In this case the field `recon_priv` is present and contains the reconstruction private value used by the requester to derive the private key corresponding to the public key associated with the implicit certificate. This is identical to the integer *r* specified in 3.6 of SEC 4 version 0.97. The use of this field is described in 5.8.6. The value `field_size` denoting the length of this value is obtained using Table 2 with input equal to the `algorithm` field of the `reconstruction_value` field in the last certificate in `certificate_chain`.

— `crl_path` contains the CRLs necessary to validate the certificate. At minimum, it shall contain the most recent version of the CRL series on which the issued certificate would appear if it were revoked. In addition, CAs should include CRLs corresponding to other CAs in the chain. These CRLs are not ordered.

Once the ToBeEncryptedCertificateResponse has been created, it shall be encapsulated in a ToBeEncrypted of type `certificate_response`, which is then encrypted and encapsulated in an EncryptedData, which in turn is then encapsulated in a 1609Dot2Data of type `encrypted`.

6.3.37 ToBeEncryptedCertificateRequestError

```
struct {
           SignerIdentifier             signer;
           opaque                       request_hash[10];
           CertificateRequestErrorCode  reason;
           Signature                    signature;
       } ToBeEncryptedCertificateRequestError;
```

If a certificate request cannot be honored, the CA informs the requestor using this structure.

— `signer` identifies the signing CA. `signer.type` shall be `certificate`.

— `request_hash` is the first 10 bytes of the SHA-256 hash of the CertificateRequest. The hash is calculated over the plaintext CertificateRequest before the request is encrypted.

— reason gives the reason why the request was rejected.

— `signature` is the responding CA's signature over the concatenation of the `request_hash` and `reason` fields. The `extern` value `algorithm` in `signature` is determined as follows:

— If `signer.certificate.version_and_type` is `2`, then `algorithm` is equal to `signer.certificate.unsigned_certificate.verification_key.algorithm`.

— If `signer.certificate.version_and_type` is `3`, then `algorithm` is equal to `signer.certificate.unsigned_certificate.signature_alg`.

A sender should encrypt this structure before sending it, for example by encapsulating it in a ToBeEncrypted with type equal to `certificate_request_error`, encrypting the ToBeEncrypted, and encapsulating it in an EncryptedData within a 1609Dot2Data of type `encrypted`.

6.3.38 CertificateRequestErrorCode

```
enum  {  verification_failure(0),
         future_request(1),
         request_too_old(2),
         future_enrolment_cert(3),
         enrolment_cert_expired(4),
         enrolment_cert_revoked(5),
         enrolment_cert_unauthorized_type(6),
         enrolment_cert_unauthorized_region(6),
         enrolment_cert_unauthorized_psids(6),
         future_ca_cert(7),
         ca_cert_expired(8),
         ca_cert_revoked(9),
```

```
    ca_cert_unauthorized_type(10),
    ca_cert_unauthorized_region(11),
    ca_cert_unauthorized_psids(12),
    canonical_identity_unknown (13),
    request_denied (14),
    private_use (240..255),
    (2^8-1)
} CertificateRequestErrorCode;
```

The values in this enumerated type have the following meaning:

— `verification_failure`: The signature on the ToBeSignedCertificateRequest could not be cryptographically verified.

— `future_request`: The request_time field in the ToBeSignedCertificateRequest was in the future when received by the CA.

— `request_too_old`: The request_time field in the ToBeSignedCertificateRequest was too far in the past when received by the CA.

— `future_enrolment_cert`: The enrolment certificate in the request was not valid at the generation time indicated in the certificate signing request.

— `enrolment_cert_expired`: The enrolment certificate in the request had expired at the time the CA processed the request (only applicable if an enrolment certificate was used).

— `enrolment_cert_revoked`: The enrolment certificate in the request had been revoked at the time the CA processed the request (only applicable if a enrolment certificate was used).

— `enrolment_cert_unauthorized_type`: The enrolment certificate that signed the request was not authorized to request certificates of the type indicated in the request.

— `enrolment_cert_unauthorized_region`: The enrolment certificate that signed the request was not authorized to request certificates with the region indicated in the request.

— `enrolment_cert_unauthorized_psids`: The enrolment certificate that signed the request was not authorized to request certificates with the PSIDs indicated in the request.

— `future_ca_cert`: The CA certificate that the request was encrypted for was not yet valid at the generation time indicated in the certificate signing request.

— `ca_cert_expired`: The CA certificate that the request was encrypted for had expired at the generation time indicated in the certificate signing request.

— `ca_cert_revoked`: The CA certificate that the request was encrypted for is revoked.

— `ca_cert_unauthorized_type`: The CA certificate that the request was encrypted for was not authorized to issue certificates of the type indicated in the request.

— `ca_cert_unauthorized_region`: The CA certificate that the request was encrypted for was not authorized to issue certificates of the type indicated in the request.

— `ca_cert_unauthorized_psids`: The CA certificate that the request was encrypted for was not authorized to issue certificates of the type indicated in the request.

— `canonical_identity_unknown`: The identity field in the CertificateRequest could not be identified as belonging to an authorized enrollee.

— `request_denied`: The request for a certificate is denied for some other reason (for example, credit card authorization failure).

Values 240–255, inclusive, are assigned for private use

6.3.39 ToBeEncryptedCertificateResponseAcknowledgment

```
struct {
            opaque                          response_hash[10];
        } ToBeEncryptedCertificateResponseAcknowledgment;
```

This acknowledgment consists of the first 10 bytes of the SHA-256 hash of the response being acknowledged. This "plaintext response" is defined as the encoded ToBeEncryptedCertificateResponse or ToBeEncryptedCertificateRequestError that was extracted from the encrypted certificate response.

6.3.40 Crl

```
struct {
            uint8               version;
            SignerIdentifier    signer;
            ToBeSignedCrl       unsigned_crl;
            Signature           signature;
        } Crl;
```

The fields in this structure have the following meaning:

— `version` contains the CRL version. In this version of the standard, this field shall be set to 1.

— `signer` identifies the signing key. `signer.type` shall not take the value `self`.

— `unsigned_crl` contains the unsigned CRL.

— `signature` contains the signature of the signer identified in the `signer` field. The signature is calculated over the contents of the `unsigned_crl` field. The `extern` value `algorithm` in `signature` is determined as follows:

— If `signer.type` is `certificate_digest_with_ecdsa_p224`, then `algorithm` is `ecdsa_nistp224_with_sha224`.

— If `signer.type` is `certificate_digest_with_ecdsa_p256`, then `algorithm` is `ecdsa_nistp256_with_sha256`.

— If `signer.type` is `certificate_digest_with_other_algorithm`, then `algorithm` is `signer.algorithm`.

— If `signer.type` is `certificate`, then:

— If `signer.certificate.version_and_type` is `2`, then `algorithm` is equal to `signer.certificate.unsigned_certificate.verification_key.algorithm`.

— If `signer.certificate.version_and_type` is `3`, then `algorithm` is equal to `signer.certificate.unsigned_certificate.signature_alg`.

— If `signer.type` is `certificate_chain`, then:

— If `signer.certificates[n].version_and_type` is `2`, then `algorithm` is equal to `signer.certificates[n].unsigned_certificate.verification_key.algorithm`, where "`signer.certificates[n]`" denotes the last certificate in `signer.certificates`.

— If signer.certificates[n].version_and_type is 3, then algorithm is equal to signer.certificates[n]. unsigned_certificate. signature_alg, where "signer.certificates[n]" denotes the last certificate in signer.certificates.

In this version of this standard, algorithm shall always be ecdsa_nistp256_with_sha256.

6.3.41 ToBeSignedCrl

```
struct {
            CrlType     type;
            CrlSeries   crl_series;
            HashedId8     ca_id;
            uint32      crl_serial;
            Time32      start_period;
            Time32      issue_date;
            Time32      next_crl;
            select (type) {
            case (id_only):
                CertId10    entries<var>;
            case (id_and_expiry):
                IdAndDate   expiring_entries<var>;
            case other_value:
                opaque      other_entries<var>;
            }
        } ToBeSignedCrl;
```

The fields in this structure have the following meaning:

— type indicates the type of the entries in the CRL. If the type id_only (0) is used, that indicates that the entries on the CRL expire no later than the expiry time of the certificate identified by ca_id.

— crl_series represents the CRL series to which this CRL belongs. This structure contains a crl_series field, and a ToBeSignedCertificate also contains a crl_series field. If a certificate with crl_series value c is revoked, that information appears on a Crl whose crl_series value is c. A CA will therefore maintain c_{max} distinct series of CRLs, where c_{max} is the maximum value of crl_series in any certificate issued by that CA.

— ca_id contains the low-order eight octets of the hash of the certificate of the CA for which this CRL is being issued.

— crl_serial is a counter that should increment by 1 for every CRL within the given CRL series for the given issuer.

— If the high-order bit of crl_serial is set, the CRL indicates all the certificates within that series that have been revoked prior to issue_date and have not expired. In this case the CRL is referred to as a full CRL.

— If the high-order bit of crl_serial is not set, the CRL indicates all the certificates within the series that have been revoked between start_period and issue_date. In this case the CRL is referred to as a delta CRL.

— start_period and issue_date specify the time period that this CRL covers. The CRL shall include all certificates belonging to that crl_series that were revoked between start_period and issue_date. CRLs from the same issuer for the same crl_series may

have overlapping time periods. If this is the case, any certificate revoked during the overlap period shall appear on multiple CRLs. `start_period` shall be before or equal to `issue_date`.

— `next_crl` contains the time when the next CRL is expected to be issued. `next_crl` shall be equal to or after `issue_date`.

— `entries` contains identifiers for each revoked certificate.

— If type is `id_only`, the entry lists only the ID for the certificate, calculated as described in 6.2.17.

— If type is `id_and_expiry`, the entry lists the ID for the certificate and the expiry date for that certificate. This allows the recipient to maintain the certificate store more efficiently.

In either case, the entries shall be sorted lexicographically by their CertId10 field.

6.3.42 CrlType

```
enum { id_only(0), id_and_expiry(1), ..., private_use(240..255),
       (2^8-1)
     } CrlType;
```

This enumerated type indicates the type of the entries in the CRL. Values 240–255, inclusive, are assigned for private use.

6.3.43 IdAndDate

```
struct {
           CertId10    id;
           Time32      expiry;
       } IdAndDate;
```

This data structure encodes information about a revoked certificate:

— `id` is the CertId10 for the revoked certificate.

— `expiry` is the value from the `expiry` field in that certificate.

NOTE—After the expiration time has passed, revocation information need no longer be maintained for that certificate. This allows the CME to maintain the revocation information store more efficiently.

6.3.44 CertId10

```
opaque CertId10[10];
```

This data structure is used to identify a certificate. The CertId10 for a given certificate shall be calculated by calculating the SHA-256 hash of the Certificate and taking the low-order 10 bytes of the hash output. The encoded Certificate which is input to the hash shall use the compressed form for all elliptic curve points.

6.3.45 CrlRequest

```
struct {
           HashedId8        issuer;
           CrlSeries      crl_series;
```

```
        Time32          issue_date;
    } CrlRequest;
```

This structure contains a request for a CRL.

The issuer field identifies the CA on whose behalf the CRL is being issued (in other words, the issuer field in the CrlRequest is the same as the ca_id that appears in the CRL itself).

The issue_date is the issue_date of the last CRL in the series for which no subsequent CRLs are missing. If 0, it indicates that the requester wants only the most recent CRL.

The request shall not be signed or encrypted.

7. Service primitives and functions

7.1 General comments and conventions

This clause specifies mechanisms for applying 1609.2 security processing to datagrams using primitives defined at Service Access Points (SAP). The primitives defined at each SAP are summarized in Table 4 and described subsequently. The details of the implementation of the primitives and their exchange protocols are not otherwise specified but are left as design decisions. Any processing that produces correct output on receipt of a given set of inputs is conformant to the standard.

Table 4—Summary of primitives

SAP	Primitive	Specified in
Sec	Sec-LocalServiceIndexForSecurity.request	7.2.1
	Sec-LocalServiceIndexForSecurity.confirm	7.2.2
	Sec-CryptomaterialHandle.request	7.2.3
	Sec-CryptomaterialHandle.confirm	7.2.4
	Sec-CryptomaterialHandle-GenerateKeyPair.request	7.2.5
	Sec-CryptomaterialHandle-GenerateKeyPair.confirm	7.2.6
	Sec-CryptomaterialHandle-StoreKeyPair.request	7.2.7
	Sec-CryptomaterialHandle-StoreKeyPair.confirm	7.2.8
	Sec-CryptomaterialHandle-StoreCertificate.request	7.2.9
	Sec-CryptomaterialHandle-StoreCertificate.confirm	7.2.10
	Sec-CryptomaterialHandle-StoreCertificateAndKey.request	7.2.11
	Sec-CryptomaterialHandle-StoreCertificateAndKey.confirm	7.2.12
	Sec-SignedData.request	7.2.13
	Sec-SignedData.confirm	7.2.14
	Sec-EncryptedData.request	7.2.15
	Sec-EncryptedData.confirm	7.2.16
	Sec-SecureDataContentExtraction.request	7.2.17
	Sec-SecureDataContentExtraction.confirm	7.2.18
	Sec-SignedDataVerification.request	7.2.19
	Sec-SignedDataVerification.confirm	7.2.20

SAP	Primitive	Specified in
	Sec-CRLVerification.request	7.2.21
	Sec-CRLVerification.confirm	7.2.22
	Sec-CertificateRequest.request	7.2.23
	Sec-CertificateRequest.confirm	7.2.24
	Sec-CertificateResponseProcessing.request	7.2.25
	Sec-CertificateResponseProcessing.confirm	7.2.26
WME-Sec	WME-Sec-SignedWsa.request	7.3.2
	WME-Sec-SignedWsa.confirm	7.3.3
	WME-Sec-SignedWsaVerification.request	7.3.4
	WME-Sec-SignedWsaVerification.confirm	7.3.5
PSSME	PSSME-LocalServiceIndexForSecurity.request	7.4.1
	PSSME-LocalServiceIndexForSecurity.confirm	7.4.2
	PSSME-SecuredProviderService.request	7.4.3
	PSSME-SecuredProviderService.confirm	7.4.4
	PSSME-SecureProviderServiceInfo.request	7.4.5
	PSSME-SecureProviderServiceInfo.confirm	7.4.6
	PSSME-CryptomaterialHandleStorage.request	7.4.7
	PSSME-CryptomaterialHandleStorage.confirm	7.4.8
	PSSME-OutOfOrderDetection.request	7.4.9
	PSSME-OutOfOrderDetection.confirm	7.4.10
CME	CME-CertificateInfo.request	7.5.1
	CME-CertificateInfo.confirm	7.5.2
	CME-AddTrustAnchor.request	7.5.3
	CME-AddTrustAnchor.confirm	7.5.4
	CME-AddCertificate.request	7.5.5
	CME-AddCertificate.confirm	7.5.6
	CME-AddCertificateRevocation.request	7.5.7
	CME-AddCertificateRevocation.confirm	7.5.8
	CME-AddCrlInfo.request	7.5.9
	CME-AddCrlInfo.confirm	7.5.10
	CME-CrlInfo.request	7.5.11
	CME-CrlInfo.confirm	7.5.12
PSSME-Sec	PSSME-Sec-CryptomaterialHandle.request	7.6.1
	PSSME-Sec-CryptomaterialHandle.confirm	7.6.2
CME-Sec	CME-Sec-ReplayDetection.request	7.7.1
	CME-Sec-ReplayDetection.confirm	7.7.2

The primitive specifications include processing specifications that may be followed to produce correct output. Some of this processing is specified in separate "functions," as summarized in Table 5:

Table 5—Interpretation of units for CertificateDuration

Functional Element	Function	Specified in
CME	CME-Function-ConstructCertificateChain	7.8.2
Security Processing Services	Sec-Function-CheckCertificateChainConsistency	7.8.3
	Sec-Function-CheckChainPsidsConsistency	7.8.4
	Sec-Function-CheckChainPsidPriorityConsistency	7.8.5
	Sec-Function-CheckChainGeographicConsistency	7.8.6
	Sec-Function-VerifyChainAndSignature	7.8.7
	Sec-Function-DecryptData	7.8.8
	Sec-Function-CertificateRequestErrrorVerification	7.8.9
	Sec-Function-CertificateResponseVerification	7.8.10

Parameters to primitives and functions are denoted in italics and have names beginning with upper-case letters. Variables used within primitives and functions are denoted in italics and have names beginning with lower-case letters. Fields within the data structures defined in Clause 6 are denoted in a `fixed-width` font.

Some primitives take parameters that are variable length strings. An implementation shall require to receive both the string contents and its length in a request primitive, and return both the contents and length in a confirm primitive. To save space in the descriptions, wherever a data type contains the word “string” (e.g., “Octet String”) this standard adopts the convention that the indicated data object provides both the length and the contents of the string. The length is denoted by *String Name.length*. The contents are denoted by *String Name.contents*.

Some primitives take parameters that are keys for cryptographic operations (or that contain keys: for example, a CMH in any state other than *Initialized*). An implementation shall require to receive both the key data and the algorithm of the key. This standard adopts the convention that the encoding of the key includes an identification of the algorithm for which it shall be used. The algorithm is denoted by *Key Name.algorithm*.

Some primitives take parameters that are arrays of variables. This standard adopts the convention that when an array is passed as a parameter, the array object makes available the number of entries in the array as well as the contents. The length is denoted by *Array Name.length*, and individual entries are denoted by *Array Name*[*i*]. The first element in an array is element 0.

A primitive may take as a parameter an array of elements where each element is structured. In this case the elements of entry i are denoted by *Array Name*[*i*].*Element Name*.

Where error indications or failure responses are considered critical to the operations, they are specified within the ".confirm" primitives. Other error conditions not defined here may be indicated in such primitives.

NOTE 1—Processing described within primitives is informative and intended to be illustrative of processing that should produce correct output. More efficient implementations than those described in this standard may be possible and may be implemented in such a way as to conform to the standard. In particular, implementations may choose to cache the results of status requests rather than recalculating them every time as a literal interpretation of this standard would suggest. An example of this is given in the notes on the specification of CME-AddCertificateRevocation.request.

NOTE 2—Sentences within processing descriptions that are enclosed in parentheses are intended to be read as comments, to make the intent of the processing clearer.

7.2 Sec SAP

7.2.1 Sec-LocalServiceIndexForSecurity.request

7.2.1.1 Function

The primitive is used by an SDEE to request a Local Service Index for Security.

7.2.1.2 Semantics of the service primitive

The primitive does not take parameters.

7.2.1.3 When generated

The primitive is generated as needed by SDEEs.

7.2.1.4 Effect of receipt

On receipt of this primitive the security processing services shall generate a unique LSI-S that has not previously been returned, and shall then return this LSI-S via the corresponding confirm primitive.

7.2.2 Sec-LocalServiceIndexForSecurity.confirm

7.2.2.1 Function

The primitive returns the values calculated in the processing described for the corresponding request primitive.

7.2.2.2 Semantics of the service primitive

The parameters of the primitive are as follows:

Sec-LocalServiceIndexForSecurity.confirm (
Result Code,
Local Service Index for Security
)

Name	Type	Valid range	Description
Result Code	Enumerated	Success, Failure	The result of the request
Local Service Index for Security	Integer	Any	The local service index for security to be used in subsequent interactions with the security processing services

7.2.2.3 When generated

The primitive is generated in response to Sec-LocalServiceIndexForSecurity.request.

7.2.2.4 Effect of receipt

No behavior is specified.

7.2.3 Sec-CryptomaterialHandle.request

7.2.3.1 Function

The primitive is used by an SDEE to request a CMH.

7.2.3.2 Semantics of the service primitive

The primitive does not take parameters.

7.2.3.3 When generated

The primitive is generated as needed by SDEEs.

7.2.3.4 Effect of receipt

On receipt of this primitive, the security processing services generate a CMH value that they have not previously returned. The security processing services return the new CMH via the corresponding confirm primitive.

7.2.4 Sec-CryptomaterialHandle.confirm

7.2.4.1 Function

The primitive returns the values calculated in the processing described for the corresponding request primitive.

7.2.4.2 Semantics of the service primitive

The parameters of the primitive are as follows:

Sec-CryptomaterialHandle.confirm (
Result Code
Cryptomaterial Handle,
)

Name	Type	Valid range	Description
Result Code	Enumerated	Success, Failure	The result of the operation.
Cryptomaterial Handle	Integer	Any	A CMH as specified in 5.2.3.

7.2.4.3 When generated

The primitive is generated in response to Sec-CryptomaterialHandle.request.

7.2.4.4 Effect of receipt

No behavior is specified.

7.2.5 Sec-CryptomaterialHandle-GenerateKeyPair.request

7.2.5.1 Function

The primitive is used by an SDEE to request a key pair from the security services for use with an associated CMH.

7.2.5.2 Semantics of the service primitive

The parameters of the primitive are as follows:

Sec-CryptomaterialHandle-GenerateKeyPair.request (
Cryptomaterial Handle,
Algorithm
)

Name	Type	Valid range	Description
Cryptomaterial Handle	Integer	Any	A CMH in *Initialized* state.
Algorithm	Enumerated type	ECDSA NIST P224, ECDSA NIST P256, ECIES NIST P256	The algorithm identifier for the key pair to be generated.

7.2.5.3 When generated

The primitive is generated as needed by SDEEs.

7.2.5.4 Effect of receipt

On receipt of this primitive, the security processing services generate a key pair for the given algorithm. The key pair is stored at the CMH provided and the CMH is transitioned to the *Key Pair Only* state. The public key is returned in the corresponding confirm primitive.

Implementations should use a high-quality random number generator to generate the key pair. This is discussed in E.4.10.

NOTE—This primitive does not check whether the handle already has a key pair associated with it. In this standard, it is the responsibility of the SDEE to keep track of the status of handles.

7.2.6 Sec-CryptomaterialHandle-GenerateKeyPair.confirm

7.2.6.1 Function

The primitive returns the values calculated in the processing described for the corresponding request primitive.

7.2.6.2 Semantics of the service primitive

The parameters of the primitive are as follows:

Sec-CryptomaterialHandle-GenerateKeyPair.confirm (
Result Code
Public Key,
)

Name	Type	Valid range	Description
Result Code	Enumerated	Success, Failure	The result of the request.
Public Key	A public key	Any public key that is valid for the algorithm provided to the corresponding request primitive	The public key from the key pair that was generated in response to the corresponding request primitive.

7.2.6.3 When generated

The primitive is generated in response to Sec-CryptomaterialHandle-GenerateKeyPair.request.

7.2.6.4 Effect of receipt

No behavior is specified.

7.2.7 Sec-CryptomaterialHandle-StoreKeyPair.request

7.2.7.1 Function

The primitive is used by an SDEE to request that the security services store a key pair generated elsewhere for use with an associated CMH.

7.2.7.2 Semantics of the service primitive

The parameters of this primitive are as follows:

Sec-CryptomaterialHandle-StoreKeyPair.request (
Cryptomaterial Handle,
Algorithm,
Public Key,
Private Key
)

Name	Type	Valid range	Description
Cryptomaterial Handle	Integer	Any	A CMH in *Initialized* state.
Algorithm	Enumerated type	ECDSA NIST P224, ECDSA NIST P256, ECIES NIST P256	The algorithm identifier for the key pair to be stored.
Public Key	Public key	Any public key valid for *Algorithm*	The public key to be stored.
Private Key	Private key	Any private key valid for *Algorithm*	The private key to be stored.

7.2.7.3 When generated

The primitive is generated as needed by SDEEs.

7.2.7.4 Effect of receipt

On receipt of this primitive, the security processing services verify that the public key and private key form a valid key pair as defined in 5.8.4. If the key pair is valid, it is stored at the CMH provided and the CMH is transitioned to the *Key Pair Only* state. The public key is returned in the corresponding confirm primitive.

NOTE 1—Implementations may choose to implement this primitive in such a way that the private key is encrypted with a key known to the security services.

NOTE 2—This primitive does not check whether the handle already has a key pair associated with it. In this standard, it is the responsibility of the SDEE to keep track of the status of handles.

7.2.8 Sec-CryptomaterialHandle-StoreKeyPair.confirm

7.2.8.1 Function

The primitive returns the result of the corresponding request primitive.

7.2.8.2 Semantics of the service primitive

The parameters of the primitive are as follows:

Sec-CryptomaterialHandle-StoreKeyPair.confirm (
Result Code
)

Name	Type	Valid range	Description
Result Code	Enumerated	Success, Invalid key pair	The result of the operation.

7.2.8.3 When generated

The primitive is generated in response to Sec-CryptomaterialHandle-StoreKeyPair.request. *Result Code* is "success" if the parameters *Public Key* and *Private Key* passed in the request primitive form a valid key pair, and "invalid key pair" if they do not.

7.2.8.4 Effect of receipt

No behavior is specified.

7.2.9 Sec-CryptomaterialHandle-StoreCertificate.request

7.2.9.1 Function

The primitive is used by an SDEE to request that the security services store a certificate at a specific CMH.

7.2.9.2 Semantics of the service primitive

The parameters of the primitive are as follows:

Sec-CryptomaterialHandle-StoreCertificate.request (
Cryptomaterial Handle,
Certificate,
Private Key Transformation
)

Name	Type	Valid range	Description
Cryptomaterial Handle	Integer	Any	A CMH in *Key Pair Only* state.
Certificate	1609.2 certificate	Any 1609.2 certificate containing a public verification key for the algorithm associated with *Cryptomaterial Handle*.	The certificate to be stored.
Private Key Transformation	A description of a linear transformation, $y = Ax+B$	Any	A transformation to be applied to the private key to ensure that it corresponds to the public key specified by the certificate. If the primitive is being invoked as a result of a ToBeEncryptedCertificate-Response with `version_and_type` equal to 3, *A* shall be equal to the SHA-256 hash of the PublicKeyReconstructionHashInput of *Certificate* and *B* shall be equal to the `recon_priv` value from the ToBeEncryptedCertificateResponse.

7.2.9.3 When generated

The primitive is generated as needed by SDEEs.

7.2.9.4 Effect of receipt

On receipt of this primitive, the security processing services verify that the following are true:

a) That the private key indicated by *Cryptomaterial Handle*, following the application of *Private Key Transformation*, and the public verification key indicated by *Certificate*, form a valid key pair as defined in 5.8.4 (for explicit certificates) or 5.8.6 (for implicit certificates).

b) That the certificate provided at *Certificate* is valid (for example by invoking CME-CertificateInfo.request).

If both statements are true, the security processing services store the updated private key and the certificate associated with the handle. If not, the certificate is not stored and the private key is unchanged.

7.2.10 Sec-CryptomaterialHandle-StoreCertificate.confirm

7.2.10.1 Function

The primitive returns the result of the corresponding request primitive.

7.2.10.2 Semantics of the service primitive

The parameters of the primitive are as follows:

Sec-CryptomaterialHandle-StoreCertificate.confirm (
Result Code
)

Name	Type	Valid range	Description
Result Code	Enumerated	Success, Keys do not match, Any *Result Code* value returned by CME-Certificate-Info.confirm	The result of the operation.

7.2.10.3 When generated

The primitive is generated in response to Sec-CryptomaterialHandle-StoreCertificate.request.

7.2.10.4 Effect of receipt

No behavior is specified.

7.2.11 Sec-CryptomaterialHandle-StoreCertificateAndKey.request

7.2.11.1 Function

The primitive is used by an SDEE to request that the security services store a certificate at a specific CMH.

7.2.11.2 Semantics of the service primitive

The parameters of the primitive are as follows:

Sec-CryptomaterialHandle-StoreCertificateAndKey.request (
Cryptomaterial Handle,
Certificate,
Private Key
)

Name	Type	Valid range	Description
Cryptomaterial Handle	Integer	Any	A CMH in *Initialized* state.
Certificate	1609.2 certificate	Any 1609.2 certificate containing a public verification key for the algorithm associated with *Cryptomaterial Handle*.	The certificate to be stored.
Private Key	A private key	Any suitable for the algorithm defined in the certificate.	The private key to be stored.

7.2.11.3 When generated

The primitive is generated as needed by SDEEs.

7.2.11.4 Effect of receipt

On receipt of this primitive, the security processing services verify that the following are true:

a) That *Private Key* and the public key indicated by *Certificate* form a valid key pair as defined in 5.8.4 (for explicit certificates) or 5.8.6 (for implicit certificates).

b) That the certificate provided at *Certificate* is valid (for example by invoking CME-CertificateInfo.request).

If both statements are true, the security processing services store the updated private key and the certificate associated with the handle. If not, the certificate is not stored and the private key is unchanged.

7.2.12 Sec-CryptomaterialHandle-StoreCertificateAndKey.confirm

7.2.12.1 Function

The primitive returns the result of the corresponding request primitive.

7.2.12.2 Semantics of the service primitive

The parameters of the primitive are as follows:

Sec-CryptomaterialHandle-StoreCertificate.confirm (
Result Code
)

Name	Type	Valid range	Description
Result Code	Enumerated	Success, Keys do not match, Any *Result Code* value returned by CME-Certificate-Info.confirm	The result of the operation.

7.2.12.3 When generated

The primitive is generated in response to Sec-CryptomaterialHandle-StoreCertificate.request.

7.2.12.4 Effect of receipt

No behavior is specified.

7.2.13 Sec-SignedData.request

7.2.13.1 Function

The primitive is used by an SDEE to request that the security services sign data.

7.2.13.2 Semantics of the service primitive

The parameters of the primitive are as follows:

Sec-SignedData.request (
Cryptomaterial Handle,
Signed Content Type,
Data,
External Data,
PSID,
Service Specific Permissions (optional),
Set Generation Time,
Generation Time,
Generation Time Standard Deviation (optional),
Set Generation Location,
Generation Location,
Set Expiry Time,
Expiry Time (optional),
Signer Identifier Type,
Signer Identifier Certificate Chain Length (optional),
Maximum Certificate Chain Length (optional),
Sign With Fast Verification,
EC Point Format,
)

Name	Type	Valid range	Description
Cryptomaterial Handle	Integer	Any integer	A CMH in the Key and Certificate state, where the permissions of the certificate referenced by the CMH allow that certificate to generate a signature on the provided data.
Signed Content Type	Enumerated	Signed, Signed partial payload, Signed external payload	The type of the signed data per 6.2.4.
Data	Octet string	Any	Used to fill in the `data` field of the ToBeSignedData.
External Data	Octet string	Any	Used to fill in the `ext_data` field of the ToBeSignedData.
PSID	Integer	0…(2^{32}-1)	Used to fill in the `psid` field of the ToBeSignedData.
Service Specific Permissions	Octet string	Length 0…32	Desired SSPs, to be checked against the certificate indicated by *Cryptomaterial Handle*.
Set Generation Time	Boolean	True, False	If True, the resulting ToBeSignedData contains the `generation_time` field.
Generation Time	Time	Any time	Used to set the `generation_time` in the ToBeSignedData.
Generation Time Standard Deviation	Integer or "unknown"	Any positive value or "unknown"	The estimated standard deviation of the *Generation Time* value, in nanoseconds.
Set Generation Location	Boolean	True, False	If True, the resulting ToBeSignedData contains the `generation_location` field.
Generation Location			Used to set the `generation_location` in the

Name	Type	Valid range	Description
			ToBeSignedData.
Set Expiry Time	Boolean	True, False	If True, the resulting ToBeSignedData contains the `expiry_time` field.
Expiry Time	Time	Any time. If *Generation Time* is included, this must be later than or equal to *GenerationTime*	If Set Expiry Time is True, this contains the expiry time.
Signer Identifier Type	Enumerated	Certificate, Certificate digest, Certificate chain	Sets the type of the SignerIdentifier within the SignedData.
Signer Identifier Certificate Chain Length	Integer or "Max"	1…256, -256…-1, or "Max"	If Signer Identifier Type is "certificate chain", sets the length of the certificate chain. If positive, includes that number of certificates from the chain. If negative with value –n, omits the top n certificates, starting with the root CA certificate, and includes the rest of the chain. If "Max", includes the entire certificate chain back to the root certificate.
Maximum Certificate Chain Length	Integer	Any positive integer ≥ 2	The maximum number of certificates that may be in the full certificate chain.
Sign With Fast Verification	Enumerated	Yes - uncompressed, Yes- compressed, or No	If this is "Yes - uncompressed" or "Yes - compressed", the output from the confirm primitive includes additional data to enable fast verification. If this is "No", the output does not contain this data, i.e. the type of R in the EcdsaSignature is set to `x_coordinate_only`.
EC Point Format	Enumerated	Uncompressed or Compressed	States whether elliptic curve points (public keys in explicit certificates and reconstruction values in implicit certificates) should be represented in compressed or uncompressed form.

7.2.13.3 When generated

The primitive is generated by an SDEE to request that the security services sign data.

7.2.13.4 Effect of receipt

On receipt of this primitive, the security services attempt to sign the indicated data using the indicated key and certificate. The security services shall only sign the data if this will result in a valid signature according to the criteria of 5.5.

The following processing ensures correct output from the corresponding confirm primitive.

a) Create variables to be used to return values to the caller via Sec-SignedData.confirm:

   1) An enumerated value, *Result Code*, initialized to "success".

   2) An octet string, *Signed Data.*

   3) An integer *Length of Certificate Chain*, corresponding to the number of certificates explicitly included in the signed data.

b) Create internal variables:

1) A certificate, *certificate*, set equal to the certificate indicated by *Cryptomaterial Handle*.
2) A certificate chain, *chain*.
3) A geographic region, *geographicScope.*
4) A permissions field, *certPermissions*.
5) Two ToBeSignedData structures, *tbsEncode* and *tbsSign*.
6) A SignedData structure, *signedData*.

c) (Check that the signing certificate is part of a consistent chain):

1) Invoke CME-Function-ConstructCertificateChain with inputs *Identifier Type* = Certificate Array, *Input Certificate Array* an array of length 1 containing *certificate*, *Terminate at Root* = True, *Maximum Chain Length = Maximum Certificate Chain Length*. Set *Result Code*, *chain*, *certPermissions*, *geographicScope* respectively to the values *Result Code*, *Certificate Chain*, *Permissions Array*[0], *Geographic Scopes*[0] returned by CME-Function-ConstructCertificateChain.
2) If *Result Code* is not "Found", return.

d) (Check that the requested fields in the data are consistent with the certificate):

1) If *Generation Time* is before the certificate's start time, set *Result Code* to "Certificate not yet valid" and return.
2) If *Generation Time* is after the certificate's expiry time, set *Result Code* to "Certificate expired" and return.
3) If *Expiry Time* was provided:
    i) If *Expiry Time* is before the certificate's start time, set *Result Code* to "expiry time before certificate validity period" and return.
    ii) If *Expiry Time* is after the certificate's expiry time, set *Result Code* to "expiry time after certificate validity period" and return.
4) If *Generation Location* is not within *geographicScope*, set *Result Code* to "outside certificate validity region" and return.
5) If *certPermissions* does not contain a (PSID, SSP) pair equal to (*PSID*, *Service Specific Permissions*), set *Result Code* to "inconsistent permissions in certificate" and return.
6) If *Signer Identifier Type* is "Certificate chain" and the absolute value of *Signer Identifier Certificate Chain Length* is greater than *chain.length*:
    i) Set *Result Code* to "incorrect requested certificate chain length".
    ii) Set *Length of Certificate Chain* to *chain.length*.
    iii) Return.

e) Fill in *tbsEncode*:

1) Set *tbsEncode*.`type` to the value of *Signed Content Type*.
2) Set *tbsEncode*.`psid` equal to *PSID*.
3) If *Set Generation Time* is True:
    i) Encode *Generation Time* and *Generation Time Standard Deviation* in *tbsEncode*.`generation_time` as described in the specification of the structure Time64WithStandardDeviation.
    ii) Set the `use_generation_time` flag in *tbsEncode*.`tf`.

4) If *Set Expiry Time* is True, encode *Expiry Time* in *tbsEncode*.`expiry_time` and set the `expiry` flag in *tbsEncode*.`tf`.

5) If *Set Generation Location* is True, encode *Generation Location* in *tbsEncode*.`generation_location` and set the `use_location` flag in *tbsEncode*.`tf`.

f) Encode *tbsEncode* including `ext_data` as necessary, as specified in 6.2.8. Hash it and digitally sign it using the mechanism associated with the signing key.

g) Fill in *signedData* as follows:

1) If *Signer Identifier Type* is "Certificate digest":

i) Set *signedData*.`signer_identifier.type` to `certificate_digest_with_ecdsap224` or `certificate_digest_with_ecdsap256`, depending on the algorithm of the verification public key indicated by *certificate*.

ii) Set *signedData*.`signer_identifier.digest` to he HashedId8 of *certificate*.

2) If *Signer Identifier Type* is "certificate":

i) Set *signedData*.`signer_identifier.type` to `certificate`.

ii) Set *signedData*.`signer_identifier.certificate` contains *certificate*.

3) If *Signer Identifier Type* is "certificate chain":

i) Set *signedData*.`signer_identifier.type` to `certificate_chain`.

ii) Set *signedData*.`signer_identifier.certificates` to the number of certificates from *chain* indicated by *Signer Identifier Certificate Chain Length*, starting with *chain*[0].

iii) If *Signer Identifier Certificate Chain Length* is negative, set *Length of certificate chain* to the number of certificates in *signedData*.`signer_identifier.certificates`.

4) Encode all elliptic curve points in the structure in compressed or uncompressed form according to *EC Point Format*.

5) Include or omit fast verification data in the signature according to *Sign With Fast Verification* as described in 6.2.18.

h) Encapsulate *signedData* in a 1609Dot2Data of type *Signed Content Type* and set *Signed Data* to the encoding of this 1609Dot2Data

i) Set *Result Code* to "Success".

j) Invoke the corresponding confirm primitive to return *Result Code*, *Signed Data*, and *Length of Certificate Chain*.

NOTE—The certificate chain construction in step c) ensures that no certificate in the chain has been revoked. It is not necessary to carry out other checks on the consistency of the certificate chain because of the requirement that a CMH is only transitioned to *Key and Certificate* state if the certificate is part of a valid and consistent chain.

7.2.14 Sec-SignedData.confirm

7.2.14.1 Function

The primitive returns the values calculated in the processing described for the corresponding request primitive.

7.2.14.2 Semantics of the service primitive

The parameters of the primitive are as follows:

Sec-SignedData.confirm (
Result Code,
Signed Data (optional),
Length of Certificate Chain (optional)
)

Name	Type	Valid range	Description
Result Code	Enumerated	Success, Certificate not yet valid, Certificate expired, Expiry time before certificate validity period, Expiry time after certificate validity period, Outside certificate validity region, Inconsistent permissions in certificate, Incorrect requested certificate chain length, or Any *Result Code* value returned from CME-Function-ConstructCertificateChain	The result of the signing operation.
Signed Data	Octet string	Any	The signed data, if it was created.
Length of Certificate Chain	Integer	Any integer ≥ 0	This is only returned if *Result Code* was "success". If returned, it has the following value: If the *Signer IdentifierType* parameter to the request primitive was not "certificate chain", undefined. If the *Signer Identifier Type* parameter was "certificate chain" but the *Signer Identifier Cert Chain Length* parameter was too big (positive) or too small (negative) to be implemented, the maximum length of the certificate chain. If the *Signer Identifier Type* parameter was "certificate chain" and if the data was successfully signed, the length of the certificate chain attached to the data.

7.2.14.3 When generated

The primitive is generated in response to Sec-SignedData.request. If the *Result Code* resulting from the Sec-SignedData.request is "success", the *Signed Data* parameter contains the encoded signed data. Otherwise, the *Result Code* is set equal to the *Result Code* from the Sec-SignedData.request and the contents of the other parameters are undefined.

7.2.14.4 Effect of receipt

No behavior is specified.

7.2.15 Sec-EncryptedData.request

7.2.15.1 Function

The primitive is used by an SDEE to request that the security services encrypt data.

7.2.15.2 Semantics of the service primitive

The parameters of the primitive are as follows:

Sec-EncryptedData.request (
Data Type,
Data,
Recipient Certificates,
EC Point Format
)

Name	Type	Valid range	Description
Data Type	Enumerated	Unsecured Signed Signed partial payload Signed external payload	The type of the data to be encrypted.
Data	Octet string	Any	The data to be encrypted.
Recipient Certificates	Array of certificates	Any array of valid certificates that contains encryption keys. An implementation may limit the maximum size of this array. If it does so, the maximum size shall be at least 6.	One certificate for each recipient.
EC Point Format	Enumerated	Compressed or Uncompressed	The format of the elliptic curve points included in the RecipientInfos.
Overdue CRL Tolerance	Time	Any positive value	The amount of time by which a CRL relevant to one of the recipient certificates may be overdue.

7.2.15.3 When generated

The primitive is generated by an SDEE to request that the security services encrypt data.

7.2.15.4 Effect of receipt

On receipt of this primitive, the security services attempt to encrypt the data for the specified recipient.

The following processing ensures correct output from the corresponding confirm primitive.

a) Create variables to be used to return values to the caller via Sec-EncryptedData.confirm:
   1) An enumerated value, *Result Code,* initialized to "success".
   2) An octet string, *Encrypted Data.*
   3) An array of certificates, *Failed Certificates*, initialized to an empty array.

b) Create internal variables:
   1) An array of certificates, *encryptionCerts*, initialized to an empty array. On successful completion this contains the certificates for which encryption is possible.
   2) An enumerated value *symm_alg* whose allowable values are any of the values of the SymmAlgorithm type, plus "not set". This variable is initialized to "not set".
   3) An octet string to be used to encode a symmetric key, *ok*.

c) For each *Recipient Certificates*[*i*]:
   1) Invoke CME-CertificateInfo.request with parameters *Identifier Type* = "Certificate", *Identifier = Recipient Certificates*[*i*].
   2) If *Result Code* returned by CME-CertificateInfo.confirm is not "found", set *Result Code* to "fail on some certificates" and add the certificate to *failedCerts.*
   3) If *Result Code* returned by CME-CertificateInfo.confirm is "found", and *Next Expected CRL Time* returned by CME-CertificateInfo.confirm is in the past by more than *Overdue CRL Tolerance*, set *Result Code* to "fail on some certificates" and add the certificate to *failedCerts.*
   4) Extract the encryption key from the certificate. If the PKAlgorithm for this key is not a known value, set *Result Code* to "fail on some certificates" and add the certificate to *failedCerts*.
   5) Set the value *current_symm_alg* equal to the symmetric encryption algorithm from the `supported_symm_alg` field of the PublicKey. If the symmetric encryption algorithm is not supported by this implementation, set *Result Code* to "fail on some certificates" and add the certificate to *failedCerts*.
   6) If *symm_alg* is not equal to "not set", and *symm_alg* is not equal to *current_symm_alg*:
      i) Set *Result Code* to "fail on some certificates" and add the certificate to *failedCerts*.
      ii) Otherwise, set *symm_alg* equal to *current_symm_alg* and add the certificate to *encryptionCerts*.

d) Check *encryptionCerts*:
   1) If there are no certificates in *encryptionCerts*, set *Result Code* equal to "fail on all certificates" and go to step l).

e) Generate a random key *k* for *symm_alg*. Convert this key to an octet string *ok*.

f) Create an array of RecipientInfo, *recipients*, initialized to an empty array.

g) Perform the following actions for every certificate in *encryptionCerts*:
   1) Extract the public encryption key from the certificate.
   2) Input *ok* and this public encryption key to the encryption mechanism. For ECIES as specified in 5.8.2, the inputs are:
      i) As the *message* parameter, the symmetric key
      ii) As the *key* parameter, the ECIES public key.

Format the output from the ECIES encryption as an EciesNistP256EncryptedKey, *enc_key*. Encode all elliptic curve points in compressed or uncompressed form according to the parameter *EC Point Format*.

3) Using *enc_key* and the current certificate, form the *RecipientInfo* for that recipient and add it to *recipients*.

h) Create a *ToBeEncrypted* structure. If *Data Type* is "Unsecured", set `type` to `unsecured` and set the `plaintext` field to *Data*. If *Data Type* is "Signed", "Signed partial payload", or "Signed external payload", set the `type` field to `signed`, `signed_partial_payload`, or `signed_external_payload` respectively, and set the `signed_data` field to the octet string beginning two octets into *data*. Encode the *ToBeEncrypted* to obtain the octet string, *plaintext*.

i) Encrypt the *plaintext* using *symm_alg* and the key *k*:

1) Generate a nonce *N* of length 12 octets.

2) Encrypt the data with AES-CCM with inputs nonce *N* and plaintext *plaintext* as described in 5.8.3.

3) Encode the output as an AesCcmCiphertext as described in 6.2.27.

j) Form an EncryptedData with the array of RecipientInfo and the AesCcmCiphertext.

k) Encapsulate the EncryptedData in a 1609Dot2Data.

l) Invoke the Sec-EncryptedData.confirm primitive to return the result to the caller.

NOTE—Step c)5) contains processing to handle a potential failure case if the symmetric encryption algorithm is not supported. This cannot happen in this version of the standard because only one symmetric algorithm is supported.

7.2.16 Sec-EncryptedData.confirm

7.2.16.1 Function

This primitive returns the values calculated in the processing described for the corresponding request primitive.

7.2.16.2 Semantics of the service primitive

The parameters of the primitive are as follows:

Sec-EncryptedData.confirm (
Result Code,
Encrypted Data (optional),
Failed Certificates (optional)
)

Name	Type	Valid range	Description
Result Code	Enumerated	Success, Incorrect inputs, Fail on some certificates, Fail on all certificates, or Any *Result Code* value returned from CME-CertificateInfo.request	The result of the encryption operation.
Encrypted Data	Octet string	Any	The encrypted data, if any was created.
Failed Certificates	Certificate array	Any	Any certificates on which encryption failed.

7.2.16.3 When generated

The primitive is generated in response to Sec-EncryptedData.request.

If the *Result Code* resulting from the Sec-EncryptedData.request is "success", the data was successfully encrypted for all recipients. In this case the *Encrypted Data* parameter contains the encoded encrypted data, and the contents of the *Failed Certificates* parameter are undefined.

If the *Result Code* from the Sec-EncryptedData.request is "fail on all certificates" or "incorrect inputs", the contents of the other parameters are undefined.

If the *Result Code* from the Sec-EncryptedData.request is "fail on some certificates", the *Encrypted Data* parameter contains the encoded encrypted data, and the *Failed Certificates* parameter contains the *failedCerts* variable that was calculated by the Sec-EncryptedData.request.

7.2.16.4 Effect of receipt

No behavior is specified.

7.2.17 Sec-SecureDataContentExtraction.request

7.2.17.1 Function

The primitive is used by an SDEE to request that the security services extract the contents from secure data.

7.2.17.2 Semantics of the service primitive

The parameters of the primitive are as follows:

Sec-SecureDataContentExtraction.request(
Data,
Cryptomaterial Handles,
)

Name	Type	Valid range	Description
Data	Octet string	An encoded 1609.2 data	The data to be processed.
Cryptomaterial Handle	Integer	Any integer	A CMH in either Key Pair Only or Key and Certificate state, referencing decryption keys.

7.2.17.3 When generated

The primitive is generated by an SDEE to request that the security processing services extract data from a 1609Dot2Data.

7.2.17.4 Effect of receipt

On receipt of this primitive, the security services attempt to extract data from a 1609Dot2Data, including decrypting the data if it is encrypted.

The following processing ensures correct output from the corresponding confirm primitive:

a) Create variables to be used to return values to the caller via Sec-SecureDataContentExtraction-.confirm.
   1) An enumerated value, *Result Code*, initialized to "success."
   2) An enumerated value, *Content Type*.
   3) An enumerated value, *Inner Content Type*.
   4) An octet string, *Contents*.
   5) An octet string, *Signed Data*.
   6) An integer, *Psid*.
   7) An octet string, *Service Specific Permissions*.
   8) A Boolean value, *Generation Time Included*.
   9) An integer, *Generation Time*.
   10) An type, *Generation Time Standard Deviation*, that can be either an integer or "Not known".
   11) A Boolean value, *Expiry Time Included*.
   12) An integer, *Expiry Time*.
   13) A Boolean value, *Generation Location Included*.
   14) An integer, *Generation Latitude*.
   15) An integer, *Generation Longitude*.
   16) An integer, *Generation Elevation*.
   17) A certificate, *Sender Certificate*.

b) Create an internal variable:
   1) An octet string, *dataToProcess*, initialized to an empty string.
   2) An enumerated value, *type*.

c) Consider *Data* as an encoded 1609Dot2Data. Set *Content Type* equal to *Data*.`type`. Set *type* equal to *Content Type*. Set *dataToProcess* equal to the octet string following the `type` field in *Data*.

d) If *Data*.protocol_version is not equal to 2, set *Result Code* to "invalid input" and go to step j).

e) If *type* is equal to encrypted:

1) Set the Boolean value *encrypted* to True.

2) Set the octet string *dataToProcess* equal to the encrypted_data field from the 1609Dot2Data.

3) Invoke Sec-Function-DecryptData with parameters *Data* = *dataToProcess*, *Cryptomaterial Handle* = *Cryptomaterial Handle*. Set *Result Code*, *Inner Content Type* and *dataToProcess* respectively to the values *Result Code*, *Inner Content Type* and *Data* returned by the function. Set *type* equal to *Inner Content Type*.

4) If *Result Code* is not "success", go to step j).

f) If *type* is not equal to unsecured, signed, signed_partial_payload, or signed_external_payload, set *Result Code* to "invalid input" and go to step j).

g) If *type* is equal to unsecured:

1) Parse dataToProcess as an opaque<var>. If *dataToProcess* is not a correctly formed opaque<var>, set *Result Code* to "invalid input" and go to step j).

2) Set *Contents* equal to the data within dataToProcess and go to step j).

h) If *type* is equal to signed, signed_partial_payload, or signed_external_payload:

1) Set *Signed Data* equal to *dataToProcess*.

2) If *dataToProcess* is not a valid SignedData, set *Result Code* to "invalid input" and go to step j).

3) Set the values *Psid*, *Generation Time Included*, *Generation Time*, *Generation Time Standard Deviation*, *Expiry Time Included*, *Expiry Time*, *Generation Location Included*, *Generation Latitude*, *Generation Longitude*, *Generation Elevation*, based on the fields psid, data, generation_time, expiry_time, and generation_location from *dataToProcess*.unsigned_data. Set *Contents* equal to *dataToProcess*.unsigned-_data.data.

4) (Obtain the certificate):

i) Denote the SignerIdentifier from the SignedData by *si*. Initialize a Certificate, *certificate*, to NULL.

ii) If *si*.type is equal to certificate or certificate_chain (see NOTE for why other possible values of *si*.type are not directly addressed).

— Set *certificate* equal to the certificate (if type is certificate) or the last certificate in the certificates field (if type is certificate_chain)

— Invoke CME-CertificateInfo.request with parameters *Identifier Type* = certificate, *Identifier* = *Certificate*. Set *Result Code*, *permissionsType*, *permissions* and *verified* respectively to *Result Code*, *Permissions Type*, *Permissions* and *Certificate Data* returned by CME-CertificateInfo.confirm.

iii) If *Result Code* is "certificate not found":

— Invoke CME-AddCertificate.request with parameters *Certificate* = *certificate*, *verified* = False to store the certificate data.

— Set *Result Code* to "found".

iv) If *Result Code* is not "found", set *Result Code* to "unknown certificate" and return.

v) Set *Sender Certificate* to *certificate*.

5) If *permissionsType* is not "PsidSsp" or "PsidPrioritySsp", set *Service Specific Permissions* to NULL.

6) Otherwise, find the entry in *permissions* that has its `psid` field equal to *Psid*. If such a PsidSsp is found, set *Service Specific Permissions* to the `service_specific_permissions` field from that PsidSsp. Otherwise, set *Result Code* to "inconsistent permissions" and return.

i) Set *Result Code* to "success".

j) Invoke the corresponding confirm primitive to return *Result Code*, and other fields if set: *Content Type, Inner Content Type, Contents, SignedData, Psid, Service Specific Permissions, Generation Time Included, Generation Time, Generation Time Standard Deviation, Expiry Time Included, Expiry Time, Generation Location Included, Generation Latitude, Generation Longitude*, *Generation Elevation,* and *Sender Certificate*.

NOTE—In step h)4)ii), the security processing services determine whether the message is signed with a previously unknown certificate. If this is the case, the security processing services store the certificate so that it may be used to verify future messages that are signed with a digest type. Any information about the validity of the certificate returned by CME-CertificateInfo.confirm is thrown away in step i). An implementation may therefore completely implement the processing for this primitive without completely implementing the processing for CME-CertificateInfo.request. See D.2 for an example of process flow for incoming secured data.

7.2.18 Sec-SecureDataContentExtraction.confirm

7.2.18.1 Function

The primitive returns the values calculated in the processing described for the corresponding request primitive.

7.2.18.2 Semantics of the service primitive

The parameters of the primitive are as follows:

Sec-SecureDataContentExtraction.confirm(
Result Code,
Content Type (optional),
Inner Content Type (optional),
Contents (optional),
SignedData (optional),
Psid (optional),
Service Specific Permissions (optional),
Generation Time Included (optional),
Generation Time (optional),
Generation Time Standard Deviation (optional),
Expiry Time Included (optional),
Expiry Time (optional),
Generation Location Included (optional),
Generation Latitude (optional),
Generation Longitude (optional),
Generation Elevation (optional),
Sender Certificate (optional)
)

Name	Type	Valid range	Description	When included
Result Code	Enumerated	Success, Invalid input, Unknown certificate, Inconsistent permissions, or Any *Result Code* value returned from Sec-Function-DecryptData, CME-CertificateInfo.request, CME-AddCertificate.request	The result of the data extraction operation.	
Content Type	Enumerated	Unsecured, Encrypted, Signed, Signed external payload, Signed partial payload	The type of the 1609Dot2Data passed in the request.	Included if *Result Code* is "success".
Inner Content Type	Enumerated	Unsecured, Signed, Signed external payload, Signed partial payload	The type contained in the ToBeEncrypted structure within the encrypted data.	Included if *Result Code* is "success" and Content Type is "encrypted".
Contents	Octet string	Any	The data payload provided by the sending SDEE; all data in the received 1609Dot2Data that was not added by the security services.	Included if *Result Code* is "success".
Signed Data	Octet string	Any	An encoded SignedData.	Included if *Result Code* is "success" and *Content Type* or *Inner Content Type* is "signed", "signed external payload", or "signed partial payload".
Psid	Integer	$0…(2^{32}-1)$	The *PSID* from the signed data.	Included if *Result Code* is "success" and *Content Type* or *Inner Content Type* is "signed", "signed external payload", or "signed partial payload".
Service Specific Permissions	Octet string	An octet string of length 0-32 octets, or NULL	The SSP from the certificate that validates the signed data.	Included if *Result Code* is "success" and *Content Type* or *Inner Content Type* is "signed", "signed external payload", or "signed partial payload".
Generation Time Included	Boolean	True, False	Whether or not the signed data included generation time.	Included if *Result Code* is "success" and *Content Type* or *Inner Content Type* is "signed", "signed external payload", or "signed partial payload".
Generation Time	Integer	$0…(2^{64}-1)$	The generation time, encoded following the rules for a Time64.	Included if *Result Code* is "success", *Content Type* or *Inner Content Type* is "signed", "signed external payload", or "signed

Name	Type	Valid range	Description	When included
				partial payload", and *Generation Time Included* is "True".
Generation Time Standard Deviation	Integer or "Not Known"	Any positive integer or "Not Known".	The estimated standard deviation of the *Generation Time* field, measured in nanoseconds, as described in Time64With-StandardDeviation.	Included if *Result Code* is "success", and *Content Type* or *Inner Content Type* is "signed", "signed external payload", or "signed partial payload", and *Generation Time Included* is True.
Expiry Time Included	Boolean	True, False	Whether or not the signed data included expiry time.	Included if *Result Code* is "success" and *Content Type* or *Inner Content Type* is "signed", "signed external payload", or "signed partial payload".
Expiry Time	Integer	$0…(2^{64}-1)$	The expiry time, encoded following the rules for a Time64.	Included if *Result Code* is "success", and *Content Type* or *Inner Content Type* is "signed", "signed external payload", or "signed partial payload", and *Expiry Time Included* is True.
Generation Location Included	Boolean	True, False	Whether or not the signed data included generation location.	Included if *Result Code* is "success" and *Content Type* or *Inner Content Type* is "signed", "signed external payload", or "signed partial payload".
Generation Latitude	Integer	-900 000 000 to 900 000 000	The latitude in one-tenth (1/10) microdegrees.	Included if *Result Code* is "success", and *Content Type* or *Inner Content Type* is "signed", "signed external payload", or "signed partial payload", and *Generation Location Included* is "True".
Generation Longitude	Integer	–1 800 000 000 to 1 800 000 000	The longitude in one-tenth (1/10) microdegrees.	Included if *Result Code* is "success", and *Content Type* or *Inner Content Type* is "signed", "signed external payload", or "signed partial payload", and *Generation Location Included* is "True".
Generation Elevation	Integer	-4095, … , 61439	The elevation, relative to the WSC-84 ellipsoid, expressed in units of 10 cm.	Included if *Result Code* is "success," and *Content Type* or *Inner Content Type* is "signed", "signed external payload", or "signed partial payload", and *Generation Location Included* is "True".
Sender Certificate	Certificate	Any certificate	The sender's certificate.	Included if *Result Code* is "success".

7.2.18.3 When generated

The primitive is generated in response to Sec-SecureDataContentExtraction.request.

7.2.18.4 Effect of receipt

If the *Content Type* parameter is signed, the SDEE may invoke the Sec-SignedDataVerification.request primitive with the *Signed Data* parameter.

7.2.19 Sec-SignedDataVerification.request

7.2.19.1 Function

The primitive is used by an SDEE to request that the security services verify signed data.

7.2.19.2 Semantics of the service primitive

The parameters of the primitive are as follows:

Sec-SignedDataVerification.request (
LSI-S,
PSID (optional),
Content Type,
Signed Data,
External Data (optional),
Maximum Certificate Chain Length (optional)
Detect Replay in Security Services,
Check Validity Based on Generation Time,
Validity Period (optional),
Generation Time (optional),
Generation Time Standard Deviation (optional),
Rejection Threshold for Too Old Data (optional),
Acceptable Future Data Period (optional),
Rejection Threshold for Future Data (optional),
Check Validity Based on Expiry Time,
Expiry Time (optional),
Rejection Threshold for Expired Data (optional),
Check Validity Based on Generation Location,
Current Location (optional),
Validity Distance (optional),
Generation Latitude (optional),
Generation Longitude (optional),
Generation Elevation (optional),
Overdue CRL Tolerance (optional)
)

Name	Type	Valid range	Description
LSI-S	Integer	Any	The Local Service Index for Security of the SDEE.
PSID	PSID	Any	The PSID that the SDEE requires to appear in the SignedData. For messages received over WSMP, this is the PSID field in the WSMP header.
Content Type	Enumerated	Signed, Signed partial payload, Signed external payload	The type of the signed data.
Signed Data	Octet string	Any	The signed data as obtained from Sec-SecureDataContentExtraction.confirm.
External Data	Octet string	Any	If *Content Type* is "signed external payload" or "signed partial payload", then the value is the external data. This value is used to fill in the `ext_data` field of the SignedData for verification per 6.2.8.
Maximum Certificate Chain Length	Integer	Any integer ≥ 2	The maximum length the certificate chain may have.
Detect Replay in Security Services	Boolean	True, False	If "True", the security services shall carry out replay detection. In this case, at least one of *Check Validity Based on Expiry Time* and *Check Validity Based on Generation Time* should be set to "True".
Check Validity Based on Generation Time	Boolean	True, False	If "True", the security services shall perform processing based on the generation time.
Validity Period	Integer	0…(2^{64}-1)	The period after the generation time for which the content is of interest to the recipient. Provided *if Check Validity Based on Generation Time* is "True".
Generation Time	Integer	0…(2^{64}-1)	The generation time to use in the security processing, encoded following the rules for a Time64. Otherwise, undefined. The security services require this field if *Check Validity Based on Generation Time* is "True" and the preceding invocation of Sec-SecureDataContentExtraction-.confirm did not return a generation time.
Generation Time Standard Deviation	Integer or "unknown"	Any positive value or "unknown"	The generation time standard deviation in nanoseconds as described in Time64With-StandardDeviation. The security services require this field if *Check Validity Based on Generation Time* is "True" and the preceding invocation of Sec-SecureDataContentExtraction.confirm did not return a generation time standard deviation.
Rejection Threshold for Too Old Data	Real-valued number	Between 0 and 1	A threshold such that if the probability that the data was generated too far in the past is greater than that threshold, the message will be rejected. Provided *if Check Validity Based on Generation Time* is "True".
Acceptable Future Data Period	Time	Any positive time value	Used in conjunction with *Rejection Threshold for Future Data* to determine if data should be rejected because its generation time is in the future. Provided *if Check Validity Based on*

Name	Type	Valid range	Description
			Generation Time is "True."
Rejection Threshold for Future Data	Real-valued number	Between 0 and 1	A threshold used in conjunction with *Acceptable Future Data Period*. If the probability that the data was generated more than *Acceptable Future Data Period* into the future is greater than this threshold, the message will be rejected. Provided *if Check Validity Based on Generation Time* is "True".
Check Validity Based on Expiry Time	Boolean	True, False	If "True", the security services will perform processing based on the expiry time.
Expiry Time	Integer	$0…(2^{64}–1)$	The expiry time to use in the security processing, encoded following the rules for a Time64. The security services require this field if *Check Validity Based on Expiry Time* is "True" and the preceding Sec-SecureDataContentExtraction-.confirm did not return an expiry time.
Rejection Threshold for Expired Data	Real-valued number	Between 0 and 1	A threshold such that if the probability that the data is past its expiry date is greater than that threshold, the message will be rejected. Provided *if Check Validity Based on Expiry Time* is "True".
Check Validity Based on Generation Location	Boolean	True, False	If True, the security services shall perform processing based on the generation location.
Current Location	2D Location	Any	The current location. Provided if *Check Validity Based on Generation Location* is "True".
Validity Distance	Integer	$1…(2^{32}–1)$	The maximum allowed distance in meters between the recipient and the generation location. This should be provided if *Check Validity Based on Generation Location* is "True".
Generation Latitude	Integer	–900 000 000 to 900 000 000	The latitude in one-tenth (1/10) microdegrees. This should be provided if *Check Validity Based on Generation Location* is true and the preceding Sec-SecureDataContentExtraction.confirm did not return a generation latitude.
Generation Longitude	Integer	–1 800 000 000 to 1 800 000 000	The longitude in one-tenth (1/10) microdegrees. This should be provided if *Check Validity Based on Generation Location* is "True" and the preceding Sec-SecureDataContentExtraction-.confirmdid not return a generation longitude.
Generation Elevation	Integer	–4095, … , 61439	The elevation, relative to the WGS-84 ellipsoid, expressed in units of 10 cm. This should be provided if *Check Validity Based on Generation Location* is "True" and the preceding Sec-SecureDataContentExtraction.confirm did not return a generation elevation.
Overdue CRL Tolerance	Time or "any"	Any time period (e.g. min, days, years), or "Any"	If a CRL relevant to a certificate in the sending chain was due to be issued more than *Overdue CRL Tolerance* time ago, and has not been received, the chain is rejected.

7.2.19.3 When generated

The primitive is generated by an SDEE to request that the security processing services verify signed data that was obtained from a previous Sec-SecureDataContentExtraction.request.

7.2.19.4 Effect of receipt

7.2.19.4.1 Overview

Upon receipt of this primitive, the security processing services attempt to verify the data using the process flow defined in 5.5.

The following processing ensures correct output from the corresponding confirm primitive:

a) Create variables to be used to return values to the caller via Sec-SignedDataVerification.confirm:
   4) An enumerated value, *Result Code*, initialized to "success".
   5) An array of times, *Last Received CRL Times*.
   6) An array of times, *Next Expected CRL Times*.
   7) A certificate, *Sender Certificate*.

b) Create internal variables:
   1) A certificate chain, *chain.*
   2) An array of permissions, *permissions.*
   3) An array of geographic scopes, *geoScopes.*
   4) An array of Booleans, *verified.*
   5) Time variables, *genTime* and *expiry*, initialized to "undefined".
   6) A time standard deviation, *genTimeStdDev*, initialized to "undefined".
   7) Location variables, *genLocLong*, and *genLocLat*, initialized to "undefined".

c) Use the provided *Content Type* to parse *Signed Data* as a SignedData. If the parsing fails, set *Result Code* to "invalid input" and go to step l).

d) Set *genTime*, *genTimeStdDev*, *expiry*, *genLocLong*, and *genLocLat* using the methods specified in 7.2.19.4.2. If *Result Code* is not "Success", go to step l).

e) If both *genTime* and *expiry* have a value (i.e., are not "unknown"), then if *expiry* is earlier than *genTime*, set *Result Code* to "expiry time before generation time" and go to step l).

f) Construct the certificate chain:
   1) Invoke CME-Function-ConstructCertificateChain with *Maximum Chain Length* set to the input parameter *Maximum Certificate Chain Length* and the other input parameters set as follows:
      i) If *Signed Data*.`signer_identifier.type` is `certificate_digest_with_ecdsap224` or `certificate_digest_with_ecdsap256`, then:
         — Identifier Type = CertId8,
         — Identifer = Signed Data.`signer_identifier.digest`.
      ii) If *Signed Data*.`signer_identifier.type` is `certificate`, then:

— *Identifier Type* = Certificate Array,

— *Input Certificate Array* = an array of length 1 containing *Signed Data*.`signer_identifier.certificate`.

iii) If *Signed Data*.`signer_identifier.type` is `certificate_chain`, then:

— *Identifier Type* = Certificate Array,

— *Input Certificate Array* = *Signed Data*.`signer_identifier-.certificates`.

2) Set *Result Code*, *chain*, *permissions*, *geoScopes*, *Last Received CRL Times*, *Next Expected CRL Times* and *verified* respectively to the output values *Result Code*, *Certificate Chain*, *Permissions Array*, *Geographic Scopes*, *Last Received CRL Times*, *Next Expected CRL Times*, and *Verified.*

3) If *Result Code* is not "success", go to step l).

g) Check the internal consistency of the certificate chain:

1) Invoke Sec-Function-CheckCertificateChainConsistency with parameters *Certificate Chain* = *chain*, *Permission Array* = *permissions*, *Geographic Scopes* = *geoScopes*. Set *Result Code* to the output value *Result Code*.

If *Result Code* is not "success", go to step l).

h) For each value in the array *Next Expected CRL Times*, if it is in the past by more than *Overdue CRL Tolerance*, set *Result Code* to "overdue CRL" and go to step l).

i) Check the consistency of the certificate chain with the data:

1) If *genTime* is defined, and is not later than or equal to the start validity time of *chain*[0], set *Result Code* to "future certificate at generation time".

2) If *genTime* is defined, and is not earlier than or equal to the expiry time of *chain*[0], set *Result Code* to "expired certificate at generation time".

3) If *expiry* is defined, and is not later than or equal to the start validity time of *chain*[0], set *Result Code* to "expiry date too early".

4) If *expiry* is defined, and is not earlier than or equal to the expiry time of *chain*[0], set *Result Code* to "expiry date too late".

5) If *genLocLat* and *genLocLong* are defined, and *geoScopes*[0] is not NULL, and *genLocLat* and *genLocLong* define a location outside *geoScopes*[0], set *Result Code* to "signature generated outside certificate validity region".

6) If *Signed Data*.`unsigned_data.psid` is not contained within *permissions*[0], set *Result Code* to "unauthorized PSID".

7) If *PSID* is provided, and *Signed Data*.`unsigned_data.psid` is not equal to *PSID*, set *Result Code* to "PSIDs don't match".

8) If the holder type of *chain*[0] is not anonymous, identified, or identified_not_localized, set *Result Code* to "unauthorized certificate type".

9) If *Result Code* is not "success", go to step l).

j) Perform the relevance tests requested by the SDEE:

1) If *Check Validity Based on Generation Time* is true:

i) Assuming a normal distribution of *Generation Time* with standard deviation *Generation Time Standard Deviation*, and given the PDF for the local estimate of time, calculate the

probability that the time interval between *Generation Time* and *Current Time* is more than *Validity Period*. If this probability is greater than *Rejection Threshold for Too Old Data*, set *Result Code* to "data expired based on generation time".

ii) Assuming a normal distribution of *Generation Time* with standard deviation *Generation Time Standard Deviation*, and given the PDF for the local estimate of time, calculate the probability that *Generation Time* is in the future by more than *Acceptable Future Data Period* relative to *Current Time*. If this probability is greater than *Rejection Threshold for Future Data*, set *Result Code* to "future data".

2) If *Check Validity Based on Expiry Time* is true:

i) Assuming a normal distribution of *Current Time* with standard deviation *Current Time Standard Deviation* respectively, calculate the probability that *Expiry Time* is in the past relative to *Current Time*. If this probability is greater than *Rejection Threshold for Expired Data*, set *Result Code* to "data expired based on expiry time".

3) If *Detect Replay in Security Services* is "True":

i) Encode *tbsEncode* including `ext_data` as necessary, as specified in 6.2.8.

ii) Invoke CME-Sec-ReplayDetection.request with parameters *LSI-S* = *LSI-S*, *Data* = *tbsEncode*. If the value of *Result Code* returned by CME-Sec-ReplayDetection.confirm is "replay", set *Result Code* to "replay".

4) If *Check Validity Based on Generation Location* is true:

i) Create the variables *localLat* and *localLong* and set them equal to the current latitude and longitude.

ii) Calculate *dist*, the distance from (*genLocLat*, *genLocLong*) to (*localLat*, *localLong*).

iii) If *dist* > (*Validity Distance*) + *localConf*, set *Result Code* to "out of range".

5) If *Result Code* is not "success", go to step l).

k) Verify the certificate chain and signature:

1) Encode *tbsEncode* including `ext_data` as necessary, as specified in 6.2.8. Set *digest* equal to the hash of this encoding, calculated with the mechanism associated with the verification key.

2) Invoke Sec-Function-VerifyChainAndSignature with inputs *Certificate Chain* = *chain*, *Verified* = *verified*, *Digest* = *digest*, *Signature* = *Signed Data*.`signature`. Set *Result Code* to the output value *Result Code*.

3) If *Result Code* is "success":

i) For each certificate *chain*[*i*]:

— Invoke CME-AddCertificate.request with parameters *Certificate* = *chain*[*i*], *verified* = True.

l) Invoke the corresponding confirm primitive to return *Result Code*, *Last Received CRL Times*, *Next Expected CRL Times*, and *Sender Certificate*.

7.2.19.4.2 Setting generation time, generation time standard deviation, generation location, expiry time

The following processing sets the correct values for the parameters *genTime*, *genTimeStdDev*, *genLocLong, genLocLat*, *expiry* used above:

Set *genTime* as follows:

a) If *Signed Data* contains a `generation_time` field, set *genTime* to the `time` field from the Time64WithStandardDeviation structure.

b) If the *Generation Time* parameter was provided:

   1) If *genTime* is already set and it is equal to the *Generation Time* parameter, return.

   2) If *genTime* is already set and it is different from the *Generation Time* parameter, set *Result Code* to "invalid input" and return.

   3) If *genTime* is not already set, set *genTime* equal to the *Generation Time* parameter.

Set *genTimeStdDev* as follows:

a) If *Signed Data* contains a `generation_time` field, set *genTimeStdDev* to the standard deviation value indicated by the `log_std_dev` field from the Time64WithStandardDeviation structure.

b) If the *Generation Time Standard Deviation* parameter was provided:

   1) If *genTimeStdDev* is already set and it is equal to the *Generation Time Standard Deviation* parameter, return.

   2) If *genTimeStdDev* is already set and it is different from the *Generation Time Standard Deviation* parameter, set *Result Code* to "invalid input" and return.

   3) If *genTimeStdDev* is not already set, set *genTimeStdDev* equal to the *Generation Time Standard Deviation* parameter.

Set *expiry* as follows:

a) If *Signed Data* contains a `expiry_time` field, set *expiry* to `expiry_time`.

b) If the *Expiry Time* parameter was provided:

   1) If *expiry* is already set and it is equal to the *Expiry Time* parameter, return.

   2) If *expiry* is already set and it is different from the *Expiry Time* parameter, set *Result Code* to "invalid input" and return.

   3) If *expiry* is not already set, set *expiry* equal to the *Expiry Time* parameter.

Set *genLocLat*, *genLocLong* as follows:

a) If *Signed Data* contains a `generation_location` field, set *genLocLat* and *genLocLong* to the values indicated by the `longitude` and `latitude` values in the ThreeDLocation structure.

b) If the *Generation Latitude* (resp. *Generation Longitude*) parameter was provided:

   1) If *genLocLat* (resp *genLocLong*) is already set and it is equal to the corresponding input parameter, return.

   2) If *genLocLat* (resp *genLocLong*) is already set and it is different from the corresponding input parameter, set *Result Code* to "invalid input" and return.

   3) If *genLocLat* (resp *genLocLong*) is not already set, set it equal to the corresponding input parameter.

c) If this has not resulted in setting *genLocLat* and *genLocLong*, set *Result Code* to "invalid input" and return.

d) If genLocLat and genLocLong have values representing unavailable sender location (900 000 001 for latitude and 1 800 000 001 for longitude), set *Result Code* to "sender location unavailable" and return.

NOTE—This processing does not use the elevation data.

7.2.20 Sec-SignedDataVerification.confirm

7.2.20.1 Function

The primitive returns the values calculated in the processing described for the corresponding request primitive.

7.2.20.2 Semantics of the service primitive

The parameters of the primitive are as follows:

Sec-SignedDataVerification.confirm (
Result Code,
Last Received CRL Times (optional),
Next Expected CRL Times (optional),
Sender Cerficate (optional),
)

Name	Type	Valid range	Description
Result Code	Enumerated	Success, Overdue CRL, Invalid input, Expiry time before generation time, Future certificate at generation time, Expired certificate at generation time, Expiry date too early, Expiry date too late, Signature generated outside certificate validity region, Unauthorized PSID, PSIDs don't match Unauthorized certificate type, Data expired based on generation time, Future data, Data expired based on expiry time, Replay, Out of range, Sender location unavailable, or Any *Result Code* value returned by CME-Function-ConstructCertificateChain, Sec-Function-	The result of the validation operation.

Name	Type	Valid range	Description
		CheckCertificateChainConsistency, CME-Sec-ReplayDetection.request, Sec-Function-VerifyChainAndSignature, or CME-Sec-ReplayDetection.request.	
Last Received CRL Times	Array of times	Any	If *Result Code* is "Success": for each certificate in the chain, the generation time of the most recently received relevant CRL in an unbroken series of relevant CRLs. . Otherwise, undefined.
Next Expected CRL Times	Array of times	Any	If *Result Code* is "Success": For each certificate in the chain, the time the next relevant CRL is expected to be generated. Otherwise, undefined.
Sender Certificate	Certificate	Any end-entity certificate	If Result Code is "*Success*", the sender's certificate. Otherwise, undefined.

7.2.20.3 When generated

The primitive is generated in response to Sec-SignedDataVerification.request.

7.2.20.4 Effect of receipt

No behavior is specified.

7.2.21 Sec-CRLVerification.request

7.2.21.1 Function

The primitive is used by a certificate management process to verify a received CRL.

7.2.21.2 Semantics of the service primitive

The parameters of the primitive are as follows:

Sec-CRLVerification.request (
CRL
Overdue CRL Tolerance
)

Name	Type	Valid range	Description
CRL	Octet string	An octet string containing a 1609Dot2Data of type `crl`.	The CRL to be verified.
Overdue CRL Tolerance	Time or "Any"	Any time period (e.g. min, days, years) or "Any".	If a CRL relevant to a certificate in the sending chain was due to be issued more than *Overdue CRL Tolerance* time ago, and has not been received, the chain is rejected.

7.2.21.3 When generated

The primitive is generated as needed by a certificate management process.

7.2.21.4 Effect of receipt

On receipt, the security processing services attempt to verify the CRL and return the result of the verification attempt to the caller. If the CRL is valid, the security processing services store the data contained in the CRL with the CME.

The following processing ensures correct output from the corresponding confirm primitive:

a) Create variables to be used to return to the caller via Sec-CRLVerification.confirm:
   1) An enumerated value, *Result Code*, initialized to "success".
   2) An array of times, *Last Received CRL Times*.
   3) An array of times, *Next Expected CRL Times*.
   4) A certificate, *Sender Certificate*.

b) Create internal variables:
   1) A certificate chain, *chain.*
   2) An array of permissions, *permissions.*
   3) An array of geographic scopes, *geoScopes.*
   4) An array of Booleans, *verified.*
   5) An octet string *digest*.

c) Check the internal correctness of the CRL:
   1) If `start_date` is not before `issue_date`, set Result Code to "Start date not before issue date" and go to step l).
   2) If `issue_date` is not before `next_crl`, set Result Code to "Issue date not before next CRL date" and go to step l).

d) If *CRL*.`signer_identifier.type` is not `certificate_digest_with_ecdsap224`, set *Result Code* to "invalid CA signature algorithm" and return.

e) Construct the certificate chain by invoking CME-Function-ConstructCertificateChain with the following inputs:
   1) If *CRL*.`signer_identifier.type` is `certificate_digest_with_ecdsap256`:
      i) *Identifier Type* = CertId8,

ii) *Identifier* = *Certificate Request Error*.`signer_identifier.digest`.

2) If *Certificate Request Error*.`signer_identifier.type` is `certificate`:

i) *Identifier Type* = Certificate Array,

ii) *Input Certificate Array* = an array of length 1 containing *Certificate Request Error*.`signer_identifier.certificate`.

3) If *Certificate Request Error*.`signer_identifier.type` is `certificate_chain`:

i) *Identifier Type* = Certificate Array,

ii) *Input Certificate Array* = *Certificate Request Error*.`signer_identifier.certificates`.

Set *Result Code*, *chain*, *permissions*, *geoScopes*, *Last Received CRL Times*, *Next Expected CRL Times* and *verified* respectively to the output values *Result Code*, *Certificate Chain*, *Permissions Array*, *Geographic Scopes*, *Last Received CRL Times*, *Next Expected CRL Times*, and *Verified.*

If *Result Code* is not "success", return *Result Code*.

f) Check the internal consistency of the certificate chain:

1) Invoke Sec-Function-CheckCertificateChainConsistency with parameters *Certificate Chain* = *chain*, *Permission Array* = *permissions*, *Geographic Scopes* = *geoScopes*. Set *Result Code* to the output value *Result Code*.

If *Result Code* is not "success", return *Result Code*.

g) For each value in the array *Next Expected CRL Times*, if it is in the past by more than *Overdue CRL Tolerance*, set *Result Code* to "overdue CRL" and go to step l).

h) Check the consistency of the certificate chain with the CRL:

1) If *Crl*.`issue_date` is not later than or equal to the start validity time of *chain*[0], set *Result Code* to "future certificate at generation time".

2) If *Crl*.`issue_date` is not earlier than or equal to the expiry time of *chain*[0], set *Result Code* to "expired certificate at generation time".

3) If *chain*[0].`holder_type` is not equal to `sde_ca`, `root_ca`, or `crl_signer`., set *Result Code* to "invalid certificate type".

4) If *chain*[0].`holder_type` is equal to `crl_signer`:

i) If *Crl*.`unsigned_crl.crl_series` is not within *chain*[0].`unsigned_certificate.scope.responsible_series`, set *Result Code* to "Certificate not authorized for specified CRL series".

ii) If *Crl*.`unsigned_crl.ca_id` is not equal to *chain*[0].`unsigned_certificate.signer_id`, set *Result Code* to "Certificate not authorized to issue CRL for specified CA".

5) If *chain*[0].`holder_type` is not equal to `crl_signer`:

i) If *Crl*.`unsigned_crl.ca_id` is not equal to the CertId8 of *chain*[0], set *Result Code* to "Wrong CA ID in CRL".

6) If *Result Code* is not "success", go to step l).

i) Verify the certificate chain and signature:

   1) Set *digest* equal to the hash of *Crl*.`unsigned_crl`, calculated with the mechanism associated with the verification key in *chain*[0].

   2) Invoke Sec-Function-VerifyChainAndSignature with inputs *Certificate Chain* = *chain*, *Verified* = *verified*, *Digest* = *digest*, *Signature* = *Signed Data*.`signature`. Set *Result Code* to the output value *Result Code*.

   3) If *Result Code* is not "success", go to step l).

j) Invoke CME-AddCrlInfo.request to store data about the CRL.

k) For every entry on the list, invoke CME-AddCertificateRevocation.request to add information about that revocation.

l) Invoke the corresponding confirm primitive to return *Result Code*.

7.2.22 Sec-CRLVerification.confirm

7.2.22.1 Function

The primitive returns the values calculated in the processing described for the corresponding request primitive.

7.2.22.2 Semantics of the service primitive

The parameters of the primitive are as follows:

Sec-CRLVerification.confirm (
Result Code,
Last Received CRL Times (optional),
Next Expected CRL Times (optional),
Sender Certificate (optional)
)

Name	Type	Valid range	Description
Result Code	Enumerated	Success, Start date not before issue date, Issue date not before next CRL date, Invalid CA signature algorithm, Overdue CRL, Future certificate at generation time, Expired certificate at generation time, Invalid certificate type, Certificate not authorized for specified CRL series, Certificate not authorized to issue CRL for specified CA, Wrong CA ID in CRL, or Any *Result Code* value returned by Sec-Function-CheckCertificateChainConsistency, Sec-Function-VerifyChainAndSignature, CME-AddCrlInfo.request, or CME-AddCertificateRevocation.request.	The result of the validation operation.
Last Received CRL Times	Array of times	Any	If *Result Code* is "success", for each certificate in the chain, the generation time of the most recently received relevant CRL in an unbroken series of relevant CRLs. Otherwise, undefined.
Next Expected CRL Times	Array of times	Any	If *Result Code* is "success", for each certificate in the chain, the time the next relevant CRL is expected to be generated. Otherwise, undefined.
Sender Certificate	Certificate	Any certificate	If *Result Code* is "success", the sender's certificate.

7.2.22.3 When generated

The primitive is generated in response to Sec-CRLVerification.request.

7.2.22.4 Effect of receipt

No behavior is specified.

7.2.23 Sec-CertificateRequest.request

7.2.23.1 Function

The primitive is used by a certificate management process to request the security processing services to construct and sign a certificate request of the form described in Clause 6.

7.2.23.2 Semantics of the service primitive

The parameters of the primitive are as follows:

Sec-CertificateRequest.request (
Signer Type,
CSR Signing Cryptomaterial Handle
Certificate Holder Type,
Public Key Transfer Type,
Permissions Type,
Permissions Array,
Identifier (optional),
Geographic Region (optional),
Use Start Validity,
Lifetime is Duration (optional),
Start Validity, (optional)
Expiration,
Verification Public Key,
Encryption Public Key (optional),
Response Encryption Key,
CA Certificate
)

Name	Type	Valid range	Description
Signer Type	Enumerated	Self, Certificate	The signer type for the certificate request.
CSR Signing Cryptomaterial Handle	Integer	If *Signer Type* is self, a Cryptomaterial Handle in *Key Pair Only* state. If *Signer Type* is certificate, a Cryptomaterial Handle in *Key and Certificate* state for which the certificate is of type Secure Data Exchange Enrolment or WSA Enrolment.	The Cryptomaterial Handle for the key to be used to sign the CSR.
Certificate Holder Type	Enumerated	Anonymous, Identified, Identified not localized, WSA signer, Secure Data Exchange Enrolment, WSA Enrolment	The type of certificate the certificate management process is requesting.
Public Key Transfer Type	Enumerated	Explicit, Implicit. If Certificate Holder Type is WSA signer, this shall be Explicit.	Type of certificate being requested, explicit or implicit.
Permissions Type	Enumerated	If *Certificate Holder Type* is anonymous, identified, or identified not localized: PsidSsp. If *Certificate Holder Type* is WSA signer: PsidPrioritySsp. If *Certificate Holder Type* is Secure Data Exchange Enrolment: Psid If *Certificate Holder Type* is WSA Enrolment: PsidPriority	The type of the permissions array to be included in the certificate request.

Name	Type	Valid range	Description
Permissions Array	Array of permissions of the type identified by *Permissions Type*	Any number of permissions of the type identified by *Permissions Type.*	The permissions to be included in the certificate.
Identifier	Octet string	If *Certificate Holder Type* is not "anonymous", an octet string of length 1-255. Otherwise, may be omitted.	An identifier to be included in the certificate.
Geographic Region	GeographicRegion	If *Certificate Holder Type* is not "identified not localized", then any valid geographic region. Otherwise, may be omitted.	The geographic region to be included in the certificate request.
Use Start Validity	Boolean	True, False	Whether to include a start validity time in the certificate request.
Lifetime is Duration	Boolean	True, False Omitted if *Use Start Validity* is False.	Indicates the format for the lifetime to be included in the certificate request.
Start Validity	Time	If *Use Start Validity* is true, this time shall be equal to or later than the start validity time of *CA Certificate*. Additionally, if *Signer Type* is certificate, this time shall be equal to or later than the start validity time of the certificate associated with *CSR Signing Cryptomaterial Handle*. If *Use Start Validity* is False, omitted.	The start validity time to appear in the certificate request.
Expiration	Time	This time shall be equal to or earlier than the expiration time of *CA Certificate.* If *Signer Type* is certificate, this time shall also be equal to or earlier than the expiration time of the certificate associated with *CSR Signing Cryptomaterial Handle.* If *Use Start Validity* is True, this time shall also be later than *Start Validity*.	The expiration time to appear in the certificate request.
Verification Public Key	A public key	A public key.	The key material to go in the `verification_key` field in the request.
Encryption Public Key	A public key, or "None"	Any public key encryption algorithm.	If provided, the key material to go in the `encryption_key` field in the request.
Response Encryption Key	Public key algorithm identifier	Any public key encryption algorithm.	The algorithm to be used to generate the response encryption key in the certificate request.

Name	Type	Valid range	Description
CA Certificate	Certificate	A correctly formed 1609.2 certificate of type `root_ca`, `sde_ca` or `wsa_ca`, containing an encryption key.	The CA certificate for which the request is to be encrypted.

7.2.23.3 When generated

The primitive is generated as needed by a certificate management process.

7.2.23.4 Effect of receipt

On receipt, the security processing services check that all certificates are valid and consistent with the request. If so, the security processing services generate a valid certificate request containing the requested fields, sign it with the indicated key, and encrypt it for the indicated CA.

The following processing ensures correct output from the corresponding confirm primitive.

The processing may make use of the following internal values:

— *Maximum Permissions Length*: the maximum number of entries in the `permissions_list` of the certificate (see 6.3.10, 6.3.12, 6.3.24, and 6.3.29).

— *Supported Region Types*: An array of enumerated values, containing “None”, “Rectangular”, “Circular”, or “Polygonal”.

— *Maximum Number of Rectangles*: the maximum number of entries in a GeographicRegion of type `rectangle` (see 6.3.14).

— *Maximum Number of Polygon Vertices*: the maximum number of entries in a GeographicRegion of type `polygon` (see 6.3.18).

a) Create variables to be used to return values to the caller via Sec-CertificateRequest.confirm:
   1) *Result Code*, an enumerated value.
   2) *Certificate Request*, an octet string.
   3) *Request Hash*, an octet string.

b) Create internal variables:
   1) Enumerated values, *caPermissionsType* and *csrCertPermissionsType.*
   2) Arrays of permissions, *caPermissions* and *csrCertPermissions.*
   3) Arrays of holder types, *caPermittedHolderTypes* and *csrCertPermittedHolderTypes*.
   4) Geographic scopes, *caGeographicScope* and *csrCertGeographicScope.*
   5) A certificate, *csrCert*, set to the certificate associated with *CSR Signing Cryptomaterial Handle* if *Signer Type* is “certificate”.

c) Check the request is consistent with the CA certificate.
   1) Invoke CME-CertificateInfo.request with parameters *Identifier Type* = certificate, *Identifier* = *CA Certificate*. Set *Result Code*, *caPermissionsType*, *caPermissions*, and *caGeographicScope* respectively to *Result Code*, *Permissions Type*, *Permissions* and *Geographic Scope* as

returned by the corresponding confirm primitive. Set *caPermittedHolderTypes* to the `permitted_holder_types` field from *CA Certificate*.

2) If *Result Code* is not "success", invoke the corresponding confirm primitive to return *Result Code*.

3) If *caPermittedHolderTypes* does not contain *Certificate Holder Type*, or *caPermissions* is not consistent with *Permissions Array*, or *caGeographicScope* does not entirely contain *Geographic Region*, set *Result Code* to "inconsistent CA permissions" and invoke the corresponding return primitive to return *Result Code*.

d) If *Signer Type* is "certificate", check the request is consistent with the enrolment certificate:

1) Invoke CME-CertificateInfo.request with parameters *Identifier Type* = certificate, *Identifier* = *csrCert*. Set *Result Code*, *csrPermissionsType*, *csrPermissions*, and *csrGeographicScope* respectively to *Result Code*, *Permissions Type*, *Permissions* and *Geographic Scope* as returned by the corresponding confirm primitive.

2) If *Result Code* is not "success", invoke the corresponding confirm primitive to return *Result Code*.

3) If *csrPermittedHolderTypes* does not contain *Certificate Holder Type*, or *csrPermissions* is not consistent with *Permissions Array*, or *csrGeographicScope* does not entirely contain *Geographic Region*, set *Result Code* to "inconsistent CSR permissions" and invoke the corresponding return primitive to return *Result Code*.

e) If *Signer Type* is "self", check that the public key in the request is equal to the public key associated with *CSR Cryptomaterial Handle*. If not, set *Result Code* to "inconsistent keys in request" and invoke the corresponding result primitive to return *Result Code*.

f) Check that the certificate conforms with local processing limits:

1) If *Permissions*.length is greater than *Maximum Permissions Length*, set *Result Code* to "Too many entries in permissions array" and invoke the corresponding return primitive to return *Result Code*.

2) If the type of *Geographic Scope* is not contained in *Supported Region Types*, set *Result Code* to "Unsupported region type in certificate" and go to step o).

3) If *Geographic Scope* is of type rectangle and contains more than *Maximum Number of Rectangles* entries, set *Result Code* to "Too many entries in rectangular geographic scope" and invoke the corresponding return primitive to return *Result Code*.

4) If *Geographic Scope* is of type rectangle and contains more than *Maximum Number of Polygon Vertices* entries, set *Result Code* to "Too many entries in polygonal geographic scope" and invoke the corresponding return primitive to return *Result Code*.

g) Generate a ToBeSignedCertificateRequest *tbs* with:

1) `version_and_type` derived from *Public Key Transfer Type*.

2) `cf` set according to *Use Start Validity*, *Lifetime is Duration*, and whether or not *Encryption Public Key* is provided.

3) `request_time` set to the current time.

4) `holder_type` set to the *Certificate Type.*

5) `type_specific_data.identifier` set to *Identifier*, if supplied.

6) `type_specific_data.permissions` set to *Permissions Array.*

7) `type_specific_data.region` set to *Geographic Region*, if supplied.

8) `start_validity` or `lifetime` set according to *Use Start Validity*, *Lifetime is Duration*, *Start Validity* and *Expiration.*

9) `verification_key` set to *Verification Public Key.*

10) `encryption_key` set to *Encryption Public Key*, if provided.

11) `response_encryption_key` set to *Response Encryption Key.*

h) Encode and sign *tbs* using the private key indicated by *CSR Signing Cryptomaterial Handle*.

i) Create a CertificateRequest *cr* with:

1) *cr*.`info.type` set according to *Signer Type.*

2) The other fields in *cr*.`info` set according to the certificate associated with *CSR Signing Cryptomaterial Handle*.

3) *cr*.`unsigned_request` set to *tbs*.

4) *cr*.`signature` set to the signature generated in the previous step.

j) Set *Request Hash* to the low-order ten bytes of the SHA_256 hash of *cr*.

k) Encrypt by invoking Sec-EncryptedData.request with parameters *Data Type* = "Certificate request", *Data* = *cr*, *Recipient Certificates* = *CA Certificate*, *EC Point Format* = "Compressed". Set *Result Code* and *Certificate Request* respectively to the values *Result Code* and *Encrypted Data* returned by Sec-EncryptedData.confirm.

l) Invoke the corresponding return primitive to return *Result Code*, *Certificate Request*, and *Request Hash* to the caller.

7.2.24 Sec-CertificateRequest.confirm

7.2.24.1 Function

The primitive returns the values calculated in the processing described for the corresponding request primitive.

7.2.24.2 Semantics of the service primitive

The parameters of the primitive are as follows:

Sec-CertificateRequest.confirm (
Result Code,
Certificate Request (optional),
Request Hash, (optional)
)

Name	Type	Valid range	Description
Result Code	Enumerated	Success, Inconsistent CA permissions, Inconsistent CSR permissions, Inconsistent keys in request, or Any *Result Code* value returned by CME-CertificateInfo.request or Sec-EncryptedData.request.	The result of the requesting operation.
Certificate Request	Octet string	A 1609Dot2Data containing an encrypted certificate request	If *Result Code* is success, the certificate request to be transmitted to the CA. Otherwise, undefined.
Request Hash	Octet string	A 10-byte octet string	The low-order ten bytes of the SHA-256 hash of the corresponding request.

7.2.24.3 When generated

The primitive is generated in response to Sec-CertificateRequest.request.

7.2.24.4 Effect of receipt

No behavior is specified.

7.2.25 Sec-CertificateResponseProcessing.request

7.2.25.1 Function

The primitive is used by a certificate management process to decrypt and verify a certificate response.

7.2.25.2 Semantics of the service primitive

The parameters of the primitive are as follows:

Sec-CertificateResponseProcessing.request (
Cryptomaterial Handle,
Data
)

Name	Type	Valid range	Description
Cryptomaterial Handle	Integer	A CMH in *Key Pair Only* state.	The handle for the key to be used to decrypt *Data.*
Data	Octet string	An octet string encoding a 1609Dot2Data encapsulating an EncryptedData	The certificate response to be verified.

7.2.25.3 When generated

The primitive is generated by a certificate management process following receipt of a certificate response.

7.2.25.4 Effect of receipt

Upon receipt of this primitive, the security processing services decrypt the certificate response, check it for validity, and return the result to the calling entity.

The following processing ensures correct output from the corresponding confirm primitive:

a) Create the following variables to be used to return to the calling entity:
   1) An enumerated value, *Result Code*.
   2) A enumerated content type, *Type*.
   3) An octet string, *Request Hash*.
   4) An enumerated value, *Error Reason*.
   5) A certificate, *Certificate*.
   6) An octet string, *Reconstruction Value*.
   7) A Boolean, *ACK Requested*.

b) Create the following internal variables:
   1) An enumerated type, *type*.
   2) An EncryptedData, *ed*.
   3) A ToBeEncryptedCertificateResponse, *tbscr*.

c) Decrypt the response:
   1) Set *ed* equal to the EncryptedData within *Data*.
   2) Invoke Sec-Function-DecryptData with parameters *Data* = *ed*, *Cryptomaterial Handle* = *Cryptomaterial Handle*. Set *Result Code*, *Type* and *data* respectively to the values *Result Code*, *Content Type* and *Data* returned by the function.
   3) If *Type* is neither “certificate response” nor “certificate request error”, set *Result Code* to “unexpected type”.
   4) If *Result Code* is not “success”, invoke the corresponding confirm primitive to return *Result Code*.

d) If type is “certificate request error”:
   1) Invoke Sec-Function-CertificateRequestErrrorVerification with input *Certificate Request Error* = *data*. Set *Result Code* equal to the value *Result Code* returned by the function.
   2) If *Result Code* is not “success”, invoke the corresponding confirm primitive to return *Result Code* to the caller.
   3) Consider *data* as a ToBeEncryptedCertificateRequestError. Set *Request Hash* to *data*.`request_hash` and *Error Reason* to *data*.`reason`.
   4) Invoke the corresponding confirm primitive to return *Result Code*, *Type*, *Request Hash* and *Error Reason* to the caller.

e) If type is “certificate response”:

1) Invoke Sec-Function-CertificateResponseVerification with *Certificate Response* = *data*. Set *Result Code* equal to the value *Result Code* returned by the function.

2) If *Result Code* is not "success", invoke the corresponding confirm primitive to return *Result Code* to the caller.

3) Consider *data* as a ToBeEncryptedCertificateResponse. Set *Certificate* and *Reconstruction Value* respectively to *data*.`certificate_chain`[0] and *data*.`recon_priv`. Set *ACK Requested* as indicated by *data*.`f`.

4) Invoke the corresponding confirm primitive to return *Result Code*, *Type, Certificate*, *Reconstruction Value*, and *ACK Requested* to the caller.

7.2.26 Sec-CertificateResponseProcessing.confirm

7.2.26.1 Function

The primitive returns the values calculated in the processing described for the corresponding request primitive.

7.2.26.2 Semantics of the service primitive

The parameters of the primitive are as follows:

Sec-CertificateResponseProcessing.confirm (
Result Code,
Type (optional),
Request Hash (optional),
Error Reason (optional),
Certificate (optional),
Reconstruction Value (optional),
ACK Request (optional)
)

Name	Type	Valid range	Description	When included
Result Code	Enumerated	Success, Unexpected type, or Any *Result Code* value returned by Sec-Function-DecryptData, Sec-Function-CertificateRequestErrrorVerification or Sec-Function-CertificateResponseVerification	The result of the verification.	
Type	Enumerated	Certificate response, Certificate request error	The type of the response.	Included if *Result Code* is "success".
Request Hash	Octet string	A 10-byte octet string	The low-order ten bytes of the SHA-256 hash of the corresponding request.	Included if *Result Code* is "success" and *Type* is "Certificate request error".
Error Reason	Enumerated	Verification failure, Enrolment cert expired, Enrolment cert revoked, Enrolment cert unauthorized, Request denied, Enrolment cert unknown, Canonical identity unknown.	The reason given by the CA for rejecting the certificate request.	Included if *Result Code* is "success" and *Type* is "Certificate request error".
Certificate	Certificate	A 1609.2 end-entity certificate	The end-entity certificate generated as a result of the request.	Included if *Result Code* is "success" and *Type* is "Certificate response".
Reconstruction Value	Octet string	An octet string of the same length as the private key for the certificate	The value to be supplied as the parameter *Private Key Transformation.B* of Sec-Cryptomaterial-Handle-Store-Certificate.request.	Included if *Result Code* is "success" and *Type* is "Certificate response".
ACK Requested	Boolean	True, False	Whether the CA has requested an ACK.	Included if *Result Code* is "success" and *Type* is "Certificate response".

7.2.26.3 When generated

The primitive is generated in response to Sec-CertificateResponseProcessing.request.

7.2.26.4 Effect of receipt

No behavior is specified.

7.3 WME-Sec SAP

7.3.1 General

This clause specifies the service access primitives and processing for signed WSAs.

From the security point of view, the difference between WSAs and signed data is in how permissions are expressed. The permissions claimed by a signed data are encoded in a single PSID. The permissions claimed by a WSA are expressed as a series of (PSID, priority) pairs, where the priority is an 8-bit field ranging in value from 0 to 63, with 0 indicating the lowest priority and 63 indicating the highest. The different form of these permissions requires a different form of certificate (with CertSpecificData of type WsaScope) and some differences in processing, specified below.

7.3.2 WME-Sec-SignedWsa.request

7.3.2.1 Function

The WME-Sec-SignedWsa.request primitive is used by an invoking WME to request that the security processing services sign a WSA.

7.3.2.2 Semantics of the service primitive

The parameters of the primitive are as follows:

WME-Sec-SignedWsa.request(
WSA Data,
Permissions,
Lifetime
)

Name	Type	Valid range	Description
WSA Data	Octet string	Any	Used to fill in the `data` field of the ToBeSignedWsa.
Permissions	Array of elements, each element being (*PSID, Priority, SSP, LSI-S*)	Any number of valid elements; an element is valid if all of its entries are valid. *PSID* is a PSID whose valid values are defined in IEEE Std 1609.3. *Priority* is an integer from 0 to 63. *LSI-S* is an arbitrary integer, and *SSP* is either null or an octet string of length no more than 32 octets. The entries in *Permissions* shall be in the same order as the corresponding ServiceInfo entries in the WSA. An implementation may set a maximum allowed length for the *Permissions* array and reject requests to sign WSAs with a *Permissions* array that is longer than this length. If this is the case, the maximum length shall be at least 32 entries.	Used to locate the signing certificate and generate the `permission_indices`. The entries map to entries in a Dot3ProviderServiceTableRequestEntry defined in IEEE Std 1609.3-2010 Annex B as follows: — *PSID* maps to dot3Provider-ServiceIdentifier — *Priority* maps to dot3Provider-ServiceRequestPriority — *SSP* maps to dot3ProviderService-SpecificPermissions — *LSI-S* maps to dot3Provider-SecurityServiceIdentifier
Lifetime	Time64	A Time64.	The time after which the signed WSA should be regarded as invalid. This is the value that will appear in the `expiry_time` field of the ToBeSignedWsa.

7.3.2.3 When generated

The primitive is generated by the WME to request that the security processing services sign a WSA.

7.3.2.4 Effect of receipt

Upon receipt of the WME-Sec-SignedWsa.request primitive, the security processing services attempt to sign the WSA data using an appropriate key and certificate. The result of the operation is a signed WSA suitable for transmission on success, or an error code on failure.

The processing creates the `permission_indices` field to map from the ServiceInfos in the WSA to the permissions in the sender's certificate. For examples of the mapping between a set of ServiceInfos and a set of permissions using `permission_indices`, and of the signing process as a whole, see D.4

The reader is referred to IEEE Std 1609.3 for the specification of the contents of the WSA, in particular the contents of the *ServiceInfo* fields within the WSA.

The following processing ensures correct output from the corresponding confirm primitive:

a) Create variables to be used to return values to the caller via WME-Sec-SignedWsa.confirm:
   1) An enumerated value, *Result Code*, initialized to "success".
   2) An octet string, *Signed WSA*.

b) Create internal variables:
   1) A certificate chain, *chain,* initialized to "uninitialized".
   2) An array of integers, *permission_indices,* whose length is equal to the number of entries in the *Permissions* parameter, and all entries initialized to 0.
   3) A cryptomaterial handle *cmh*.

c) Invoke PSSME-Sec-CryptomaterialHandle.request with parameter *Permissions* = *Permissions*, *Location* = the current location.

d) Set *Result Code*, *chain*, *cmh* and *permission_indices* respectively to the values *Result Code*, *Certificate Chain*, *CMH*, and *Permission Indices* returned by PSSME-Sec-Cryptomaterial-Handle.confirm. If Result Code is not "success", go to step i).

e) Fill in a ToBeSignedWsa structure as follows:
   1) Set the `permission_indices` field to *permission_indices*.
   2) Set the `use_location` and `use_generation_time` flags in the `tf` field.
   3) Set the `data` field to the *WSA Data* parameter.
   4) Encode the current time and estimated standard deviation in the `generation_time` field and the current location in the `generation_location` field.[25]
   5) Set the `expiry_time` field equal to the *Lifetime* parameter and set the `expiry` flag in the `tf` field.

f) Encode the ToBeSignedWsa as an octet string. Generate a cryptographic message digest of this octet string with the digest mechanism associated with the signing key. Set the time and location fields appropriately. Digitally sign the digest using the private key referenced by *CMH*. The signature algorithm shall be ECDSA using the method of FIPS 186 that does not output additional fast verification data.

g) Create and encode a SignedWsa such that:
   1) The `signer` field is of type `certificate_chain` and contains *chain*.
   2) The `unsigned_wsa` field contains the encoded ToBeSignedWsa.
   3) The `signature` field contains the signature, with the field R set not to include fast verification data.

h) Encapsulate the SignedWsa in a 1609Dot2Data with ContentType equal to `signed_wsa`. Denote this by *Signed WSA*.

i) Invoke the WME-Sec-SignedWsa.confirm primitive to return *Result Code* and *Signed WSA* to the caller.

[25] This standard does not specify performance requirements, and the order of operations within primitives is not normative (except where necessary to obtain correct output). A conformant implementation of this primitive may insert the time and location at any point between the invocation of the security services to sign a WSA and the start of the cryptographic operation of signing. If an implementation typically takes multiple μs between the invocation of the security services and the start of the cryptographic signature operation, it should choose the generation time confidence to be no less than the typical length of time that this processing takes.

7.3.3 WME-Sec-SignedWsa.confirm

7.3.3.1 Function

The primitive returns the values calculated in the processing described for the corresponding request primitive.

7.3.3.2 Semantics of the service primitive

The parameters of the primitive are as follows:

WME-Sec-SignedWsa.confirm (
Result Code,
Signed WSA (optional),
)

Name	Type	Valid range	Description
Result Code	Enumerated	Success, Geographic scope error, or Any Result Code value returned by PSSME-Sec-Cryptomaterial-Handle.confirm.	The outcome of the signing operation.
Signed WSA	Octet string	Any	If *Result Code* is "success", the signed WSA with the attached certificate chain.

7.3.3.3 When generated

The primitive is generated in response to WME-Sec-SignedWsa.request.

7.3.3.4 Effect of receipt

As described in IEEE Std 1609.3.

7.3.4 WME-Sec-SignedWsaVerification.request

7.3.4.1 Function

The primitive is used by a WME to request that the security processing services verify a signed WSA and extract the Service Specific Permissions (SSPs) relevant to each service advertised in the WSA.

NOTE—This primitive is referred to in IEEE Std 1609.3-2010 as WaveSecurityServices-SignedWsaValidation.request, and the corresponding confirm primitive is referred to as WaveSecurityServices-SignedWsaValidation.confirm. A future revision to IEEE Std 1609.3 will make the terminology consistent.

7.3.4.2 Semantics of the service primitive

The parameters of the primitive are as follows:

WME-Sec-SignedWsaVerification.request (
Signed WSA Data,
)

Name	Type	Valid range	Description
Signed WSA	Octet string	An encoding of a 1609Dot2Data containing a SignedWsa.	The encoded signed WSA.

7.3.4.3 When generated

The primitive is generated by a WME to request that the security processing services verify a signed WSA.

7.3.4.4 Effect of receipt

Upon receipt of the WME-Sec-SignedWsaVerification.request primitive, the security processing services attempt to verify the signature and extract the SSPs relevant to each secured service within the WSA. The result of the operation is an indication of whether the WSA was valid, and if it was not valid, the reason why the verification failed. The security processing services use the WME-Sec-SignedWsa-Verification.confirm primitive to return the result of the processing to the WME.

For an example of the mapping between a set of ServiceInfos and a set of permissions using `permission_indices`, see D.4.

The following processing ensures correct output from the corresponding confirm primitive:

a) Create variables to be used to return to the caller via WME-Sec-SignedWsaVerification.confirm:
   1) An array of enumerated values, *Result Codes*, each initialized to “unknown”.
   2) An octet string, *WSA Data.*
   3) An array of Service Specific Permissions, *Service Specific Permissions*, each initialized to null.
   4) An integer representing μs, *Generation Time*.
   5) An integer, *Generation Time Standard Deviation*.
   6) An integer representing μs, *Expiry Time.*
   7) Integers *Generation Latitude*, *Generation Longitude*, and *Generation Elevation.*
   8) A time variable, *Last Received CRL Time.*
   9) A time variable, *Most Overdue CRL Time.*
   10) An array of times, *Last Received CRL Times.*
   11) An array of times, *Next Expected CRL Times.*
   12) A certificate, *Sender Certificate*.
b) Create internal variables:
   1) A certificate chain, *chain.*
   2) An array of permissions, *permissions.*

3) An array of geographic scopes, *geoScopes.*

4) An array of Booleans, *verified.*

5) An enumerated value, *resultCode*, initialized to "success".

c) Ensure that the signed WSA parses correctly:

1) Consider the *Signed WSA* parameter as an encoded 1609Dot2Data. If any of the following conditions is true, set every entry in *Result Codes* to "invalid input" and go to step l):

i) `protocol_version` is not equal to 2

ii) `type` is not equal to `signed_wsa`

iii) *SignedWSA*`.signer.type` is not `certificate_chain`

iv) `generation_time` is later than `expiry_time`

2) Set *tbsWSA*, a ToBeSignedWsa, equal to *Signed WSA*`.data.unsigned_wsa`.

3) Set *WSA Data*, *Generation Time*, *Generation Time Standard Deviation*, *Expiry Time*, *Generation Latitude*, *Generation Longitude*, *Generation Elevation* to the corresponding values extracted from *tbsWSA*.

4) Extract the ServiceInfos from the received WSA into a variable *ServiceInfos*. Extract the `permission_indices` field from the SignedWsa. If the number of entries in *ServiceInfos* is different from the number of entries in `permission_indices`, set every entry in *Result Codes* to "invalid input" and go to step l).

5) Set *Result Codes*.length to the length of `permission_indices`. For *i* = 1 to *Result Codes*.length:

i) If `permission_indices`[*i*] is 0, set *Result Codes*[i] equal to "unsecured"; otherwise set *Result Codes*[i] to "undefined".

d) Construct the certificate chain by invoking CME-Function-ConstructCertificateChain with the following inputs:

1) *Identifier Type* = Certificate Array,

2) *Input Certificate Array* = *Signed WSA*`.signed_wsa.signer_identifier-.certificates`.

3) *Maximum Chain Length* = 8.

Set *resultCode*, *chain*, *permissions*, *geoScopes*, *Last Received CRL Times*, *Next Expected CRL Times* and *verified* respectively to the output values *Result Code*, *Certificate Chain*, *Permissions Array*, *Geographic Scopes*, *Last Received CRL Time*, *Next Expected CRL Time*, and *Verified.*

e) Invoke PSSME-OutOfOrderDetection.request with parameters *Generation Time* = *Generation Time* and *Certificate* = *chain*[0]. If PSSME-OutOfOrderDetection.confirm returns "not most recent WSA", set *resultCode* to "not most recent WSA" and go to step l).

f) Check the internal consistency of the certificate chain:

1) Invoke Sec-Function-CheckCertificateChainConsistency with parameters *Certificate Chain* = *chain*, *Permission Array* = *permissions*, *Geographic Scopes* = *geoScopes*. Set *resultCode* to the output value *Result Code*. If *resultCode* is not "success", go to step l).

g) Check the consistency of the certificate chain with the SignedWSA:

1) If *Generation Time* is not later than or equal to the start validity time of *chain*[0], set *Result Code* to "future certificate at generation time".

2) If *Generation Time* is not earlier than or equal to the expiry time of *chain*[0], set *Result Code* to "expired certificate at generation time".

3) If *Expiry Time* is not later than or equal to the start validity time of *chain*[0], set *Result Code* to "expiry date too early".

4) If *Expiry Time* is not earlier than or equal to the expiry time of *chain*[0], set *Result Code* to "expiry date too late".

5) If *Generation Latitude* and *Generation Latitude* are defined, and *geoScopes*[0] is not NULL, and *Generation Latitude* and *Generation Longitude* define a location outside *geoScopes*[0], set *Result Code* to "WSA generated outside certificate validity region".

6) If the `holder_type` field in chain[0] is not `wsa`, set *resultCode* to "unsupported signer type".

7) If *resultCode* is not "success", go to step l).

h) Check consistency of the ServiceInfos in the WSA with the permissions in the certificate:

1) Extract the ServiceInfos from the received WSA into a variable *ServiceInfos*. Extract the `permission_indices` field from the SignedWsa. If the number of entries in *ServiceInfos* is different from the number of entries in `permission_indices`, set *resultCode* to "invalid input" and go to step l).

2) For *i* = 1 to the length of `permission_indices`:

i) If `permission_indices`[*i*] is 0, set *Result Codes*[i] equal to "unsecured"; next *i*.

ii) Set *j* equal to `permission_indices`[*i*]; set *ppCert* equal to *permissions*[*j*]; set *pp* equal to the PSID and Priority values in *ServiceInfos*[*i*].

iii) If the PSID in *pp* is different from the PSID in *ppCert*, set *resultCode* to "inconsistent permissions" and go to step l).

iv) If the max_priority in *ppCert* is less than or equal to the priority in *pp*, set *resultCode* to "unauthorized psid and priority in WSA", and go to step l).

v) Set *Service Specific Permissions*[i] to the SSP in *ppCert*, or null if *ppCert* does not contain an SSP.

vi) Set i = i+1 and return to h)2)i).

3) If *resultCode* is not "success", go to step l).

i) Verify the certificate chain and signature:

1) Encode *tbsWSA*. Set *digest* equal to the hash of this encoding, calculated with the mechanism associated with the verification key.

2) Invoke Sec-Function-VerifyChainAndSignature with inputs *Certificate Chain* = *chain*, *Verified* = *verified*, *Digest* = *digest*, *Signature* = *Signed Data*.`signature`. Set *resultCode* to the output value *Result Code*.

3) If *resultCode* is not "success", go to step l).

j) Set *WSA Signing Certificate* equal to *chain*[0].

k) Set *resultCode* to "success".

l) For each entry in *Result Codes* that is not equal to "unsecured", set it equal to *resultCode*.

m) Invoke the corresponding confirm primitive to return *Result Codes*, *WSA Data*, *Service Specific Permissions*, *Generation Time*, *Generation Time Standard Deviation*, *Expiry Time*, *Generation Latitude*, *Generation Longitude*, *Generation Elevation*, *Most Overdue CRL*, *Last Received CRL Times*, *Next Expected CRL Times* and *Sender Certificate*.

7.3.5 WME-Sec-SignedWsaVerification.confirm

7.3.5.1 Function

The primitive returns the values calculated in the processing described for the corresponding request primitive.

7.3.5.2 Semantics of the service primitive

The parameters of the primitive are as follows:

WME-Sec-SignedWsaVerification.confirm (
Result Codes,
WSA Data (optional),
Service Specific Permissions (optional),
Generation Time (optional),
Generation Time Standard Deviation (optional),
Expiry Time (optional),
Generation Latitude (optional),
Generation Longitude (optional),
Generation Elevation (optional),
Last Received CRL Times (optional),
Next Expected CRL Times (optional),
Sender Certificate (optional)
)

Name	Type	Valid range	Description
Result Codes	Array of enumerated values	Success, Unsecured, Invalid input, Undefined, Not most recent WSA, Future certificate at generation time, Expired certificate at generation time, Expiry date too early, Expiry date too late, WSA generated outside certificate validity region, or Any *Result Code* value returned by PSSME-Out-OfOrderDetection.request, Sec-Function-CheckCertificateChain-Consistency, or Sec-Function-VerifyChainAndSignature.	For each ServiceInfo in the WSA, the result of the verification operation, corresponding to the *Result Codes* variable set within processing for the request primitive. The values “success” and “unsecured” indicate success cases. The other values indicate failure cases.
WSA Data	Octet string	Any	If any entry in *Result Codes* indicates a success case, the unsigned WSA. Otherwise, undefined.

Name	Type	Valid range	Description
Service Specific Permissions	Array of Service Specific Permissions	An array of Service Specific Permissions whose length (number of entries) is the same as the number of ServiceInfos in the WSA, such that each entry is either NULL or an octet string of length no more than 32 octets.	If any entry in *Result Codes* indicates a failure case, undefined. Otherwise, for each ServiceInfo in the WSA, the corresponding SSP.
Generation Time	Integer	0…(2^{64}-1)	If no entry in *Result Codes* is "invalid input", the generation time, encoded following the rules for a Time64. Otherwise, undefined.
Generation Time Standard Deviation	Integer or "unknown"	Positive integer or "unknown"	If no entry in *Result Codes* is "invalid input", the generation time standard deviation in nanoseconds as specified for Time64WithStandardDeviation. Otherwise, undefined.
Expiry Time	Integer	0…(2^{64}-1)	If no entry in *Result Codes* is "invalid input", the expiry time, encoded following the rules for a Time64. Otherwise, undefined.
Generation Latitude	Integer	-900 000 000 to 900 000 000	If no entry in *Result Codes* is "invalid input": The latitude in one-tenth (1/10) microdegrees. Otherwise, undefined.
Generation Longitude	Integer	-1 800 000 000 to 1 800 000 000	If no entry in *Result Codes* is "invalid input": The longitude in one-tenth (1/10) microdegrees. Otherwise, undefined.
Generation Elevation	Integer	-4095, … , 61439	If no entry in *Result Codes* is "invalid input": The elevation, relative to the WSC-84 ellipsoid, expressed in units of 10 cm. Otherwise, undefined.
Last Received CRL Times	Array of times	Any	For each certificate in the chain, the generation time of the most recently received relevant CRL in an unbroken series of relevant CRLs.
Next Expected CRL Times	Array of times	Any	For each certificate in the chain, the time the next relevant CRL is expected to be generated. See 7.2.19.4.1 step f) for a fuller explanation.
Sender Certificate	1609.2 certificate	Any certificate of type `wsa_signer`	If any entry in *Result Codes* indicates a failure case, undefined. Otherwise, the WSA signing certificate.

7.3.5.3 When generated

The primitive is generated in response to WME-Sec-SignedWsaVerification.request.

7.3.5.4 Effect of receipt

For each ServiceInfo in the WSA, if the corresponding entry in *Result Codes* is "success" or "unsecured", the WME updates the UserAvailableServiceInfo field for that service. If the corresponding entry in *Result Codes* takes any other value, the WME does not take action for that service.

The following parameters are not explicitly discussed in IEEE Std 1609.3-2010:

- Generation Time
- Generation Time Standard Deviation
- Expiry Time
- Last Received CRL Times
- Next Expected CRL Times
- Sender Certificate

An implementation of the WME may choose to make these parameters available, for example via an extension of the contents of UserAvailableServiceTable.

7.4 PSSME SAP

7.4.1 PSSME-LocalServiceIndexForSecurity.request

7.4.1.1 Function

The primitive is used by a secure provider service to request a Local Service Index for Security to enable interactions with the PSSME.

7.4.1.2 Semantics of the service primitive

The primitive does not take parameters.

7.4.1.3 When generated

The primitive is generated as needed by secure provider services.

7.4.1.4 Effect of receipt

On receipt of this primitive, the security processing services generate an LSI-S that they have not previously returned. The security services then return the new LSI-S via the corresponding confirm primitive.

7.4.2 PSSME-LocalServiceIndexForSecurity.confirm

7.4.2.1 Function

The primitive returns the values calculated in the processing described for the corresponding request primitive.

7.4.2.2 Semantics of the service primitive

The parameters of the primitive are as follows:

PSSME-LocalServiceIndexForSecurity.confirm (
Result Code,
Local Service Index for Security,
)

Name	Type	Valid range	Description
Result Code	Enumerated	Success, Failure	The outcome of the requesting operation.
Local Service Index for Security	Integer	Any	The local service index for security to be used in subsequent interactions with the security processing services.

7.4.2.3 When generated

The primitive is generated in response to PSSME-LocalServiceIndexForSecurity.request.

7.4.2.4 Effect of receipt

No behavior is specified.

7.4.3 PSSME-SecuredProviderService.request

7.4.3.1 Function

The primitive is used by a secure provider service to request inclusion in WSA signing certificates.

7.4.3.2 Semantics of the service primitive

The parameters of the primitive are as follows:

PSSME-SecuredProviderService.request (
Local Service Index for Security,
Action,
Provider Service Identifier,
Maximum Service Priority,
Service Specific Permissions,
)

Name	Type	Valid range	Description
Local Service Index for Security	Integer	0–65 535	Unique identifier of the secure provider service to the PSSME.
Action	Enumerated	Add, Delete	Indicates the requested action.
Provider Service Identifier	Octet string	See IEEE Std 1609.3	The PSID for the secure provider service, to be included in the certificate.
Maximum Service Priority	Integer	0–63	The maximum priority for the secure provider service, to be included in the certificate.
Service Specific Permissions	Octet string	Length 0-32 octets	The SSP for the secure provider service, to be included in the certificate.

7.4.3.3 When generated

The primitive is generated as needed by secure provider services.

7.4.3.4 Effect of receipt

On receipt, the PSSME generates a PSSME-SecuredProviderService.confirm indicating whether the request is accepted. On acceptance, the parameters are stored by the PSSME for retrieval by a certificate management process using the PSSME-SecuredProviderService.request primitive.

7.4.4 PSSME-SecuredProviderService.confirm

7.4.4.1 Function

This primitive confirms the acceptance of the corresponding request.

7.4.4.2 Semantics of the service primitive

The parameters of the primitive are as follows:

PSSME-SecuredProviderService.confirm (
Result Code
)

Name	Type	Valid range	Description
Result Code	Enumerated	Success, Failure	The result of the request

7.4.4.3 When generated

The primitive is generated in response to PSSME-SecuredProviderService.request.

7.4.4.4 Effect of receipt

No behavior is specified.

7.4.5 PSSME-SecureProviderServiceInfo.request

7.4.5.1 Function

The primitive is used by a certificate management process to obtain information about secure provider services that have registered with the PSSME.

7.4.5.2 Semantics of the service primitive

The parameters of the primitive are as follows:

PSSME-SecureProviderServiceInfo.request (
Local Service Index For Security,
)

Name	Type	Valid range	Description
Local Service Index For Security	Integer, or "All"	Any integer, or "All"	The LSI-S for the services about which information is requested.

7.4.5.3 When generated

The primitive is generated as needed by a certificate management process, typically to aggregate information to form one or more certificate requests. The invoking service shall previously have used the PSSME-LocalServiceIndexForSecurity primitive to obtain an LSI-S.

7.4.5.4 Effect of receipt

On receipt, the PSSME retrieves the information requested and invokes the corresponding confirm primitive to return a response to the requesting entity.

7.4.6 PSSME-SecureProviderServiceInfo.confirm

7.4.6.1 Function

The primitive returns the values obtained in the processing described for the corresponding request primitive.

7.4.6.2 Semantics of the service primitive

The parameters of the primitive are as follows:

PSSME-SecureProviderServiceInfo.confirm (
Result Code,
ServiceInfo Array (optional)
)

Name	Type	Valid range	Description
Result Code	Enumerated	Accepted, Rejected unknown LSI-S, Rejected unspecified	Indicates the result of the associated request.
ServiceInfo Array	Array of elements (*Local Service Index for Security, Provider Service Identifier, Maximum Priority, Service Specific Permissions*) where *Local Service Index for Security* is an integer, *Provider Service Identifier* is an octet string, *Maximum Priority* is an integer, and *Service Specific Permissions* is an octet string	*Local Service Index for Security* may have any value, *Provider Service Identifier* is a valid PSID as defined in IEEE Std 1609.3, *Maximum Priority* is an integer between 0 and 63, and *Service Specific Permissions* is an octet string of length 0-32 octets	If *Result Code* is not "Accepted", undefined. Otherwise, each entry contains a set of (*Local Service Index for Security, Provider Service Identifier, Maximum Priority, Service Specific Permissions*) that have been submitted to PSSME-SecuredProviderService.request with *Action* = Add and not subsequently deleted by invoking PSSME-SecuredProviderService.request with *Action* = Delete. If the *Local Service Index for Security* parameter to the corresponding request primitive was "All", this contains the information for all secure provider services. If the parameter was specified and corresponds to a known *LSI-S*, this contains those information elements containing the given *LSI-S*.

7.4.6.3 When generated

The primitive is generated in response to PSSME-SecureProviderServiceInfo.request.

7.4.6.4 Effect of receipt

No behavior is specified.

7.4.7 PSSME-CryptomaterialHandleStorage.request

7.4.7.1 Function

The primitive is used by a certificate management process to store a CMH in *Key and Certificate* state with the PSSME for later use when signing WSAs.

7.4.7.2 Semantics of the service primitive

The parameters of the PSSME-CryptomaterialHandleStorage.request primitive are as follows:

PSSME-CryptomaterialHandleStorage.request (
Cryptomaterial Handle,
LSI-S Array
)

Name	Type	Valid range	Description
Cryptomaterial Handle	Integer	A CMH in the *Key and Certificate* state, referencing a WSA certificate	The CMH to be stored.
LSI-S Array	Array of integers	The array shall be the same length as the `permissions` array in *Certificate*. Each integer may have any value	The LSI-S corresponding to the PsidPrioritySsp fields in the permissions_list field of Certificate.

7.4.7.3 When generated

The primitive is generated by a certificate management entity that has access to a new certificate and private key and wishes to store it within the PSSME for later access by the security services.

7.4.7.4 Effect of receipt

On receipt, the PSSME ensures that the entries in *LSI-S Array* are consistent with the `permissions_list` array in the certificate reference by *Cryptomaterial Handle*. The entries in *LSI-S Array* are consistent if each combination of (*LSI-S*[*i*], `permissions_list`[i].`psid`, `permissions_list`[i].`max_priority`, `permissions_list`[i].`service_specific_permissions`) matches a set of (*Local Service Index for Security*, *Provider Service Identifier*, *Maximum Priority*, *Service Specific Permissions*) that have been submitted to PSSME-SecuredProviderService.request with *Action* = Add and not subsequently deleted by invoking PSSME-SecuredProviderService.request with *Action* = Delete. If the entry is valid the PSSME stores the certificate and key internally. The PSSME then invokes the corresponding confirm primitive to return the result of the request to the caller.

7.4.8 PSSME-CryptomaterialHandleStorage.confirm

7.4.8.1 Function

The primitive confirms the acceptance of the corresponding request.

7.4.8.2 Semantics of the service primitive

The parameters of the primitive are as follows:

PSSME-CryptomaterialHandleStorage.confirm (
Result Code,
)

Name	Type	Valid range	Description
Result Code	Enumerated	Success, Unrecognized *LSI-S*, Invalid input, Permissions entry does not match registered secure provider service	The result of the request.

7.4.8.3 When generated

The primitive is generated in response to PSSME-SecuredProviderService.request.

7.4.8.4 Effect of receipt

No behavior is specified.

7.4.9 PSSME-OutOfOrderDetection.request

7.4.9.1 Function

The primitive is used by the security processing services to detect whether a WSA generation time for a given certificate is the most recent generation time received for that certificate, and to store that information for future reference.

7.4.9.2 Semantics of the service primitive

The parameters of the primitive are as follows:

PSSME-OutOfOrderDetection.request (
Generation Time,
Certificate
)

Name	Type	Valid range	Description
Generation Time	A time in μs	Any	The generation time of the WSA.
Certificate	A certificate	Any valid WSA signing certificate	The certificate used to sign the WSA.

7.4.9.3 When generated

The primitive is generated during processing of signed WSAs as specified under WME-Sec-SignedWsa-Verification.request.

7.4.9.4 Effect of receipt

On receipt, the PSSME determines whether the *Generation Time* parameter is the most recent generation time received for *Certificate*. If *Generation Time* is the most recent generation time, or no generation times have previously been stored for *Certificate*, the PSSME stores (*Certificate*, *Generation Time*) and invokes PSSME-OutOfOrderDetection.confirm to return “success”. If there is already a generation time value stored for the given certificate, and the generation time value is more recent than *Generation Time*, the PSSME invokes PSSME-OutOfOrderDetection.confirm to return “not most recent WSA”.

NOTE—Since this primitive stores (*Certificate*, *Generation Time*), it is making the assumption that the WSA has already been verified.

7.4.10 PSSME-OutOfOrderDetection.confirm

7.4.10.1 Function

The primitive returns the values calculated in the processing described for the corresponding request primitive.

7.4.10.2 Semantics of the service primitive

The parameters of the primitive are as follows:

PSSME-OutOfOrderDetection.confirm (
Result Code,
)

Name	Type	Valid range	Description
Result Code	Enumerated	Success, Not most recent WSA	The result of the out of order detection processing.

7.4.10.3 When generated

The primitive is generated in response to PSSME-OutOfOrderDetection.request.

7.4.10.4 Effect of receipt

As specified under WME-Sec-SignedWsaVerification.request.

7.5 CME SAP

7.5.1 CME-CertificateInfo.request

7.5.1.1 Function

This primitive is used by a certificate management process to query the CME for information about the contents, revocation status and inherited permissions of a certificate.

7.5.1.2 Semantics of the service primitive

The parameters of the primitive are as follows:

CME-CertificateInfo.request (
Identifier Type,
Identifier
)

Name	Type	Valid range	Description
Identifier Type	Enumerated	Certificate, HashedId8, CertId10	Indicates the type of input data used to identify the certificate.
Identifier	Octet string	Any	The encoded certificate, HashedId8, or CertId10 identifying the certificate in question.

7.5.1.3 When generated

The primitive is generated as needed within the WAVE Security Services or by other entities or processes.

7.5.1.4 Effect of receipt

On receipt, the CME determines:

— whether the certificate indicated is revoked.

— whether the certificate indicated is known.

— if the certificate indicated is known, what its permissions are.

— whether the certificate is a trust anchor.

— when the next CRL is due and when the last received CRL was received.

The CME then invokes the corresponding confirm primitive to return the result of the processing to the calling entity.

The calculation of permissions includes obtaining (or attempting to obtain) inherited permissions as illustrated in Figure 30.

The processing may make use of the following internal values:

— *Maximum Permissions Length*: the maximum number of entries in the `permissions_list` of the certificate (see 6.3.10, 6.3.12, 6.3.24, and 6.3.29).

— *Supported Region Types*: An array of enumerated values, containing “None”, “Rectangular”, “Circular”, or “Polygonal”.

— *Maximum Number of Rectangles*: the maximum number of entries in a GeographicRegion of type `rectangle` (see 6.3.14).

— *Maximum Number of Polygon Vertices*: the maximum number of entries in a GeographicRegion of type `polygon` (see 6.3.18).

The following processing ensures correct output from the corresponding confirm primitive:

NOTE—An implementation may cache the results of this processing for performance reasons.

a) Create variables to be used to return values to the caller via CME-CertificateInfo.confirm:

   1) An enumerated value, *Result Code*, initialized to “found”.

2) A certificate chain, *Certificate Data*.

3) An array, *Permissions*, with elements *Psid*, *PsidPriority*, *PsidSsp*, or *PsidPrioritySsp*.

4) An enumerated value *Permissions Type*, that may take the values *Psid*, *PsidPriority*, *PsidSsp*, *PsidPrioritySsp*, or "Not Found".

5) A geographic region, *Geographic Scope*.

6) Time variables *Last Received CRL Time* and *Next Expected CRL Time*.

7) A Boolean, *Trust Anchor*, initialized to False.

8) A Boolean, *Verified*.

b) If *Identifier Type* is Certificate or CertId10:

1) If the certificate has been revoked, in other words if its CertId10 corresponds to a CertId10 previously provided to CME-AddCertificateRevocation.request, set *Result Code* to "certificate revoked" and go to step o).

c) If there is no certificate that was provided using CME-AddCertificate.request and that has the corresponding Identifier:

1) Set *Result Code* to "certificate not found".

2) If *Identifier Type* is Certificate, use the `crl_series` and `signer_id` fields in the certificate to determine which CRL series that certificate would appear on, and set *Next CRL Date* to the next CRL date of that series.

3) go to step o).

d) (If processing reaches this step, the certificate has been found): If the certificate is marked "not trusted", set *Result Code* to "certificate not trusted" and go to step o).

e) Set *Result Code* to "found".

f) If the certificate has been stored but not verified (see 7.2.17.2), set *Verified* to False. Otherwise, set *Verified* to True.

g) Set *Certificate Data* equal to the certificate.

h) Set *Last Received CRL Time* and *Next Expected CRL Time* to the last received CRL time and next expected CRL time for the certificate, which have been set as specified in the description of CME-AddCrlInfo.request, and CME-AddCertificate.request.

i) If the certificate's operational permissions or geographic scope are explicit (not inherited), set *Permissions*, *Permissions Type* and *Geographic Scope* as given in the certificate.

j) If the certificate is a trust anchor, set *Trust Anchor* to True.

k) Check that the certificate conforms with local processing limits:

1) If *Permissions*.length is greater than *Maximum Permissions Length*, set *Result Code* to "Too many entries in permissions array" and go to step o).

2) If the type of *Geographic Scope* is not contained in *Supported Region Types*, set *Result Code* to "Unsupported region type in certificate" and go to step o).

3) If *Geographic Scope* is of type rectangle and contains more than *Maximum Number of Rectangles* entries, set *Result Code* to "Too many entries in rectangular geographic scope" and go to step o).

4) If *Geographic Scope* is of type rectangle and contains more than *Maximum Number of Polygon Vertices* entries, set *Result Code* to "Too many entries in polygonal geographic scope" and go to step o).

l) If all of the certificate's permissions are explicit, go to step o).

m) (If processing reaches this step, the certificate has been found but has inherited permissions and the processing needs to determine the permissions): If the certificate is a trust anchor, set *Result Code* to "badly formed certificate" and go to step o).

n) Set *c* equal to *Certificate Data*. Set the inherited permissions type indicator (*Permissions Type*, *Geographic Scope*, or both) to "inherited not found". Set the variable *done* to false. Do the following while *done* is false:

   1) Obtain the issuing certificate by invoking CME-CertificateInfo.request with parameters *Identifier Type* = CertId8, *Identifier* = *c*.`signer_id`.

   2) If the confirm primitive sets *Result Code* to any value other than "found", go to step o).

   3) If *Permissions Type* is "inherited not found", and the confirm primitive returned a value other than "inherited not found", set *Permissions Type* and *Permissions* to the value returned by the confirm primitive.

   4) If *Geographic Scope* is "inherited not found", and the confirm primitive returned a value other than "inherited not found", set *Geographic Scope* to the value returned by the confirm primitive.

   5) If neither *Permissions Type* nor *Geographic Scope* are "inherited not found", set *done* to true. Otherwise, set *c* equal to the value of *Certificate Data* returned by the confirm primitive.

o) Invoke the corresponding confirm primitive to return *Result Code*, *Certificate Data*, *Permissions Type*, *Permissions*, *Geographic Scope*, *Last Received CRL Time*, *Next Expected CRL Time*, *Trust Anchor* to the caller.

7.5.2 CME-CertificateInfo.confirm

7.5.2.1 Function

This primitive returns the certificate identified in the corresponding request, if found.

7.5.2.2 Semantics of the service primitive

The parameters of the primitive are as follows:

CME-CertificateInfo.confirm (
Result Code
Certificate Data,
Permissions Type,
Permissions,
Geographic Scope,
Last Received CRL Time,
Next Expected CRL Time,
Trust Anchor,
Verified
)

Name	Type	Valid range	Description
Result Code	Enumerated	Found, Certificate revoked, Certificate not found, Certificate not trusted,	Indicates the result of the associated request. All values other than "found" correspond to failure cases.

		Badly formed certificate, Too many entries in permissions array, Unsupported region type in certificate Too many entries in rectangular geographic scope, Too many entries in polygonal geographic scope	
Certificate Data	Octet string	Any	The encoded certificate, if Result Code is "found"; otherwise, undefined.
Permissions Type	Enumerated	Psid, PsidPriority, PsidSsp, PsidPrioritySsp, or "Inherited not found"	The type of the permissions parameter.
Permissions	List of *Psid, PsidPriority, PsidSsp, or PsidPrioritySsp*	Any	If Result Code is "found", the permissions associated with the certificate, either inherited or explicit. Otherwise, undefined.
Geographic Scope	Geographic Region or "Inherited not found"	Any	If Result Code is "found", the geographic scope associated with the certificate, either inherited or explicit. Otherwise, undefined.
Last Received CRL Time	Date or "none"	Any date in the past	If available, the generation time of the last CRL received that would have contained the certificate if it had been revoked. Otherwise, undefined.
Next Expected CRL Time	Date or "unknown"	Any date after Last Received CRL Time	If available, the next time a CRL will to be generated that could contain the certificate. This time may be in the past or the future. Otherwise, undefined.
Trust Anchor	Boolean	True, False	Whether or not the certificate is a trust anchor.
Verified	Boolean	True, False	Whether or not the certificate has been cryptographically verified.

7.5.2.3 When generated

The primitive is generated in response to the corresponding request.

7.5.2.4 Effect of receipt

Given in the specification of the invoking processes.

7.5.3 CME-AddTrustAnchor.request

7.5.3.1 Function

The primitive is used by a certificate management process to add a trust anchor to the CME's store of trust anchors.

7.5.3.2 Semantics of the service primitive

The parameters of the primitive are as follows:

CME-AddTrustAnchor.request (
Certificate
)

Name	Type	Valid range	Description
Certificate	1609.2 certificate	Any 1609.2 certificate	The certificate to be added as a trust anchor

7.5.3.3 When generated

The primitive is generated as needed by a certificate management process to add a trust anchor to the CME.

7.5.3.4 Effect of receipt

On receipt, the CME ensures that *Certificate* has the following properties:

— It is a correctly formed certificate.

— It is an explicit rather than implicit certificate.

— It has not been revoked.

— If it is of type `root_ca`, the public key in the `verification_key` field verifies the signature on the certificate.

If all of these conditions hold, the CME adds *Certificate* to its store of trust anchors. The CME then invokes the corresponding confirm primitive to return the result of the processing.

7.5.4 CME-AddTrustAnchor.confirm

7.5.4.1 Function

The primitive returns the values calculated in the processing described for the corresponding request primitive.

7.5.4.2 Semantics of the service primitive

The parameters of the primitive are as follows:

CME-AddTrustAnchor.confirm (
Result Code
)

Name	Type	Valid range	Description
Result Code	Enumerated	Success, Invalid input, Certificate revoked, Implicit certificate not acceptable as trust anchor	Indicates the result of the associated request.

7.5.4.3 When generated

The primitive is generated in response to the corresponding request.

7.5.4.4 Effect of receipt

No behavior is specified.

7.5.5 CME-AddCertificate.request

7.5.5.1 Function

This primitive allows a certificate management process calling entity to add a certificate to the CME's store of certificates.

7.5.5.2 Semantics of the service primitive

The parameters of the primitive are as follows:

CME-AddCertificate.request (
Certificate,
Verified
)

Name	Type	Valid range	Description
Certificate	1609.2 certificate	Any 1609.2 certificate	The certificate to be added.
Verified	Boolean		Whether or not the certificate has been cryptographically verified (see 7.2.17.2).

7.5.5.3 When generated

The primitive is generated as needed by any certificate management process or entity that wishes to add a trust anchor to the CME.

7.5.5.4 Effect of receipt

On receipt, the CME ensures that *Certificate* is a correctly formed certificate and has not been revoked. If these conditions hold, the CME adds *Certificate* and *Type* to its store of certificates. The CME sets the *Last Received CRL Time* and *Next Expected CRL Time* to the appropriate values as drawn from the most recent CRL that might apply to the certificate. The CME then invokes the corresponding confirm primitive to return the result of the processing.

7.5.5.5 Effect of receipt

No behavior is specified.

7.5.6 CME-AddCertificate.confirm

7.5.6.1 Function

The primitive returns the values calculated in the processing described for the corresponding request primitive.

7.5.6.2 Semantics of the service primitive

The parameters of the primitive are as follows:

CME-AddCertificate.confirm (
Result Code
)

Name	Type	Valid range	Description
Result Code	Enumerated	Success, Invalid input, Certificate revoked	Indicates the result of the associated request.

7.5.6.3 When generated

The primitive is generated in response to the corresponding request.

7.5.6.4 Effect of receipt

No behavior is specified.

7.5.7 CME-AddCertificateRevocation.request

7.5.7.1 Function

The primitive is used by a certificate management process to provide the CME with revocation information relating to a certificate.

7.5.7.2 Semantics of the service primitive

The parameters of the primitive are as follows:

CME-AddCertificateRevocation.request (
Identifier,
CA Identifier,
CRL Series,
Expiry
)

Name	Type	Valid range	Description
Identifier	Octet string	An octet string of length 10	The CertId10 identifying the revoked certificate.
CA Identifier	Octet string	An octet string of length 8	The CertID8 identifying the CA certificate that issued the revoked certificate.
CRL series	Integer	$1 \ldots 2^{32}-1$	The CRL series that includes the revoked certificate.
Expiry	Time	Any time in the future	The time at which the indicated certificate may be removed from the CME's store of revoked certificates. If the CRL is of type `id_and_expiry`, this should be the expiry field from the IdAndDate entry that contained *Identifier*. Otherwise, this shall be no later than the expiry time of the certificate identified by *CA Identifier*, and may be the expiry time of the revoked certificate if that information is known to the caller.

7.5.7.3 When generated

The primitive is generated by a process or entity that obtains certificate revocation information. This information may be obtained from a CRL or by other means out of the scope of this standard.

7.5.7.4 Effect of receipt

The CME stores the revocation information and returns a confirm primitive.

NOTE—If the revoked certificate is a CA certificate, the CME may choose to locate any certificates issued by that CA within the CME internal storage, and mark those certificates as also revoked. This may save time when processing subsequently received signed data, as it enables the CME to identify a signing certificate as revoked immediately, rather than having to use the full processing given in this standard.

7.5.8 CME-AddCertificateRevocation.confirm

7.5.8.1 Function

This primitive returns the result of the corresponding request.

7.5.8.2 Semantics of the service primitive

The parameters of the primitive are as follows:

CME-AddCertificateRevocation.confirm (
Result Code
)

Name	Type	Valid range	Description
Result Code	Enumerated	Success, Invalid input	Indicates the result of the associated request.

7.5.8.3 When generated

The primitive is generated in response to the corresponding request.

7.5.8.4 Effect of receipt

No behavior is specified.

7.5.9 CME-AddCrlInfo.request

7.5.9.1 Function

The primitive is used by a certificate management process to update the CME's information about the status of a certificate revocation list series.

7.5.9.2 Semantics of the service primitive

The parameters of the primitive are as follows:

CME-AddCrlInfo.request (
CRL Type,
CRL Series,
CRL Identifier,
Serial Number,
Start Period,
Issue Date,
Next Crl
)

Name	Type	Valid range	Description
CRL Type	Enumerated value	Id Only, Id and Expiry	The type of the entries in the CRL.
CRL Series	Integer	$1…2^{32}$-1	The CRL series value from the CRL.
CRL Identifier	Octet string	An octet string of length 8	The low-order eight octets of the hash of the certificate of the CA for which this CRL was issued.
Serial Number	Integer	Any	The Serial Number value from the CRL.
Start Period	Time	Any date	The start of the time period that this CRL covers.
Issue Date	Time	Any date prior to current date	The issue date of the CRL.
Next CRL	Time	Any time after *Issue Date*	The time when the next CRL is expected to be issued.

7.5.9.3 When generated

This primitive is generated by a certificate management process to provide information about a CRL to the CME.

7.5.9.4 Effect of receipt

On receipt of this primitive the CME updates the CRL information as listed above. For every known certificate that might be present on the CRL, the CME updates the *Last Received CRL Time* to *Issue Date* and the *Next Expected CRL Time* to *Next CRL*.

7.5.10 CME-AddCrlInfo.confirm

7.5.10.1 Function

The primitive acknowledges the receipt of the corresponding request primitive.

7.5.10.2 Semantics of the service primitive

This primitive takes no parameters.

7.5.10.3 When generated

This primitive is generated in response to the corresponding request primitive.

7.5.10.4 Effect of receipt

No behavior is specified.

7.5.11 CME-CrlInfo.request

7.5.11.1 Function

The primitive is used by a certificate management process to request CRL information.

7.5.11.2 Semantics of the service primitive

The parameters of the primitive are as follows:

CME-CrlInfo.request (
CRL Series,
CA Identifier
Serial Number
)

Name	Type	Valid range	Description
CRL Series	Integer	$1…2^{32}$-1	The CRL series value from the CRL.
CA Identifier	Octet string	An octet string of length 8	The low-order eight octets of the hash of the certificate of the CA for which this CRL was issued.
Serial Number	Integer	Any	The Serial Number value from the CRL.

7.5.11.3 When generated

This primitive is generated by a certificate management process to request CRL information.

7.5.11.4 Effect of receipt

On receipt of this primitive, the CME retrieves information about the CRL identified by *CRL Series*, *CA Identifier*, and *Serial Number* and returns that information, if available, to the certificate management process via CME-CrlInfo.confirm.

The following processing ensures correct output from the corresponding confirm primitive:

a) Create variables to be used to return values to the caller via CME-CrlInfo.confirm:
   1) An enumerated value, *Result Code*
   2) An enumerated value, *CRL Type*
   3) A time, *Start Period*
   4) A time, *Issue Date*
   5) A time, *Next CRL*

b) If *CA Identifier* or (*CA Identifier*, *Crl Series*) does not correspond to any entries in the CME, set *Result Code* to “unrecognized identifier”.

c) If there is an entry associated with the identifying tuple (*CRL Series*, *CA Identifier*, *Serial Number*) in the CME's internal storage of CRL information:

   1) If all of the entries on the CRL have expired, or if the cert that signed the CRL has expired, set *Result Code* to "expired" and set *CRL Type, Start Period, Issue Date, Next CRL* and *Result Code* to the corresponding values from the corresponding invocation of CME-AddCrlInfo.request.

   2) If the entry has not expired, set *Result Code to "*success" and set *CRL Type, Start Period, Issue Date, Next CRL* to the corresponding values from the corresponding invocation of CME-AddCrlInfo.request.

d) If there is no entry associated with (*CRL Series*, *CA Identifier*, *Serial Number*), but there are other known CRLs with the same *CRL Series* and *CA identifier*:

   1) If the CME has the most recent CRL in the series and the next_crl time in that CRL is in the future, set *Result code* to "not issued yet".

   2) Otherwise, set *Result Code* to "missing".

e) Invoke the corresponding confirm primitive to return *CRL Type, Start Period, Issue Date, Next CRL* and *Result Code* to the caller.

7.5.12 CME-CrlInfo.confirm

7.5.12.1 Function

The primitive returns the values calculated in the processing described for the corresponding request primitive.

7.5.12.2 Semantics of the service primitive

The parameters of the primitive are as follows

CME-CrlInfo.confirm (
Result Code,
CRL Type,
Start Period,
Issue Date,
Next CRL
)

Name	Type	Valid range	Description
Result Code:	Enumerated	Success, Unrecognized identifier, Expired, Not issued yet, Missing	Indicates the result of the associated request.
CRL Type	Enumerated value	ID Only, ID and Expiry	The type of the entries in the CRL.
Start Period	Time	Any date	The start of the time period covered by this CRL.
Issue Date	Time	Any date prior to current date	The issue date of the CRL.
Next CRL	Time	Any time after *Issue Date*	The time when the next CRL is expected to be issued.

7.5.12.3 When generated

This primitive is generated in response to the corresponding request primitive.

7.5.12.4 Effect of receipt

No behavior is specified.

7.6 PSSME-Sec SAP

7.6.1 PSSME-Sec-CryptomaterialHandle.request

7.6.1.1 Function

This primitive allows the security processing services to request a WSA signing certificate from the PSSME.

7.6.1.2 Semantics of the service primitive

The parameters of the PSSME-Sec-CryptomaterialHandle.request primitive are as follows:

PSSME-Sec-CryptomaterialHandle.request (
Permissions,
Location
)

Name	Type	Valid range	Description
Permissions	Array of elements, each element being (*PSID, Priority, SSP, LSI-S*).	Any number of valid elements; an element is valid if *PSID, Priority* and *LSI-S* take permitted values for their types, and *SSP* either is null or takes permitted values for its type.	Used to find the signing certificate.
Location	A 2D Location	Any location on the surface of the Earth.	Used to find a signing certificate that is valid at the current location.

7.6.1.3 When generated

The primitive is generated in response to PSSME-SecuredProviderService.request.

7.6.1.4 Effect of receipt

On receipt of this primitive, the PSSME locates a CMH such that:

— The CMH was previously provided to the PSSME by the PSSME-CryptomaterialHandle-Storage.request primitive.

— The certificate references by the CMH contains permissions for each of the secure provider services identified in *Permissions* by *Permissions*[*i*].*LSI-S*.

— For each secure provider service *i*, the PSID and SSP fields in its entry in the certificate match those in *Permissions*[*i*] and the maximum priority in the certificate is greater than or equal to the *Permissions*[*i*].*priority*. (The entry for a given secure provider service is obtained using the indices passed as parameters of PSSME-CryptomaterialHandleStorage.request).

— The certificate has not expired.

— The certificate is valid at *Location*.

On success, the PSSME invokes the PSSME-Sec-CryptomaterialHandle.confirm primitive to return the certificate, the corresponding private key, and the permission indices for inclusion in the WSA.

The following processing ensures correct output from the corresponding confirm primitive:

a) Create variables to be used to return values to the caller:
   1) An enumerated value, *Result Code*, initialized to “success”.
   2) An array of indices, *Permission Indices*, of length *Permissions.length*, with every entry initialized to 0.
   3) A certificate chain, *Certificate Chain*.
   4) A Cryptomaterial Handle, *CMH*.

b) Create internal variables:
   1) An empty array, *cList*, with each array element being a tuple of (CMH *cmh*, array of LSI-S *lsisArray*).
   2) An empty array of permissions, *currentCertPermissions*.
   3) A Boolean, *certificateFound*.
   4) A certificate, *c*.

5) A geographic region, *geoPermissions*.

c) (Find certificates with the appropriate set of LSI-S): For each *wsaCertInfo* = (*CMH*, *LSI-S Array*) that was provided to the PSSME by PSSME-CryptomaterialHandleStorage.request:

1) If every *Permissions*[*i*].*LSI-S* is contained within *wsaCertInfo.LSI-S Array*, add (*CMH*, *LSI-S Array*) to *cList*.

d) If *cList* is empty, set *Result Code* to "no certificate found" and go to step j).

e) Set *certificateFound* to "false".

f) For each *cList*[*i*]:

1) Set *c* = *cList[i].CMH.certificate*.

2) If *c* has expired, move to the next value for *wsaCertInfo* and return to step c).

3) (Establish that the certificate hasn't been revoked, and recover the permissions if they were inherited): Invoke CME-CertificateInfo.request with parameters *Identifier Type* = "Certificate", *Identifier* = *c*. If the *Result Code* parameter from CME-CertificateInfo.confirm is "Success", set *currentCertPermissions* and *geoPermissions* respectively equal to the *Permissions* and *Geographic Scope* parameters from CME-CertificateInfo.confirm.

4) If *geoPermissions* does not contain *Location*, move to the next entry in *cList* and return to f).

5) For each *Permissions*[*j*] (where *Permissions* refers to the input parameter *Permissions*):

i) Find *k* such that *cList*[*i*].*lsisArray*[*k*] matches *Permissions*[*j*].*LSI-S*.

ii) Denote the PsidPrioritySsp at *currentCertPermissions*[*k*] by *pps'*.

iii) If *Permissions*[*j*].*PSID* is not equal to *pps'*.`psid`, or *Permissions*[*j*].*SSP* is not equal to *pps'*.`service_specific_permissions`, move to the next entry in *cList* and return to f).

iv) If *Permissions*[*j*].*priority* is greater than *pps'*.`max_priority`, move to the next entry in *cList* and return to f).

v) (If processing reaches this step, the certificate matches this *Permissions* entry): set *Permision Indices*[*j*] to *k*.

vi) If this was the last entry in *Permissions*, set *certificateFound* to "true" and exit this loop to step g).

g) If *certificateFound* is false, set *Result Code* to "no certificate found" and go to step j).

h) Invoke CME-Function-ConstructCertificateChain with input *Identifier Type* = "certificate", *Input Certificate Array* = a one-entry array consisting of *c*. Set *Result Code* and *Certificate Chain* to the values *Result Code* and *Certificate Chain* returned from the function.

i) If *Result Code* is "success", set *CMH* = *cList*[*i*].*CMH*.

j) Invoke the corresponding confirm primitive to return *Result Code*, *Permission Indices*, *Certificate Chain* and *CMH*.

7.6.2 PSSME-Sec-CryptomaterialHandle.confirm

7.6.2.1 Function

This primitive returns the values calculated in the processing described for the corresponding request primitive

7.6.2.2 Semantics of the service primitive

The parameters of the PSSME-Sec-CryptomaterialHandle.confirm primitive are as follows:

PSSME-Sec-CryptomaterialHandle.confirm (
Result Code,
Permission Indices,
CMH,
Certificate Chain,
)

Name	Type	Valid range	Description
Result Code	Enumerated	Success, Unrecognized LSI-S, No certificate found, or Any *Result Code* value returned by CME-Certificate-Info.confirm or CME-Function-ConstructCertificateChain.	The result of the signing operation.
Permission Indices	Array of integers	Each number is between 1 and the number of ServiceInfo fields in the WSA to be signed	If *Result Code* is "success", the permission indices used to create the signed WSA, corresponding to the *Permission Indices* value set within processing for the request primitive. Otherwise, undefined.
CMH	Cryptomaterial handle	A CMH in the *Key and Certificate* state	If *Result Code* is "success", the CMH to be used to sign the WSA. Otherwise, undefined.
Certificate Chain	Certificate chain		If *Result Code* is "success", the certificate chain to be included in the signed , corresponding to the *chain* value set within processing for the request primitive WSA. Otherwise, undefined.

7.6.2.3 When generated

The primitive is generated in response to PSSME-Sec-CryptomaterialHandle.request.

7.6.2.4 Effect of receipt

Specified in the specification of WME-Sec-SignedWsa.request

7.7 CME-Sec SAP

7.7.1 CME-Sec-ReplayDetection.request

7.7.1.1 Function

This primitive allows any calling entity to determine whether received signed data is a duplicate of signed data that has already been received by that entity, and to request the CME to store that signed data for future replay detection.

7.7.1.2 Semantics of the service primitive

The parameters of the primitive are as follows:

CME-Sec-ReplayDetection.request (
LSI-S
Data
)

Name	Type	Valid range	Description
LSI-S	Integer	Any	The local service index for security that identifies the calling entity.
Data	Octet string	An octet string	The encoded SignedData that is to be checked for being a replay. The encoding of the enclosed `unsigned_data` shall include the `ext_data` field as specified in the specification in 6.2.8 of encoding a ToBeSignedData to be digested.

7.7.1.3 When generated

The primitive is generated during the execution of Sec-SignedDataVerification.request as described in the specification of that primitive.

7.7.1.4 Effect of receipt

On receipt of this primitive, the CME determines whether the input *Data* has already been received by the entity indicated by the *Local Service Entry for Security*, and stores *Data* for use in future replay detection. The result of the replay detection is returned to the security services by use of the corresponding confirm primitive.

The following processing ensures correct output from the corresponding confirm primitive:

a) Create the variable *Result Code*, to be used to return the result of the replay detection operation to the security services.

b) If the pair (*Local Service Entry for Security*, *Data*) has already been stored by the CME, set *Result Code* to “replay”. Otherwise, set *Result Code* to “not replay”.

c) Store the pair (*Local Service Entry for Security*, *Data*).

d) Invoke the corresponding confirm primitive to return *Result Code*.

NOTE 1—The comparison is over the entire body of *Data*, not simply over the data field (an efficient way to implement the check would be to hash *Data* and cache only the hash). Therefore, this check does not discard data with repeated contents if the timestamp and signature are fresh.

NOTE 2—The check described here detects an exact copy of *Data* that has already been received. A receiver might also choose to implement additional processing to reject a datagram if only the data contents are repeated, even if the security headers have a new timestamp and signature.

NOTE 3—If a (*Local Service Entry for Security*, *Data*) pair is not relevant to the service identified by *Local Service Entry for Security* (for example, if the data has expired), the CME may remove that pair from storage.

7.7.2 CME-Sec-ReplayDetection.confirm

7.7.2.1 Function

This primitive return the values calculated in the processing described for the corresponding request primitive.

7.7.2.2 Semantics of the service primitive

The parameters of the primitive are as follows:

CME-Sec-ReplayDetection.confirm (
Result Code
)

Name	Type	Valid range	Description
Result Code	Enumerated	Replay, Not replay	Indicates the result of the associated request.

7.7.2.3 When generated

The primitive is generated in response to CME-Sec-ReplayDetection.request.

7.7.2.4 Effect of receipt

Specified in the specification of the invoking process.

7.8 Internal functions

7.8.1 General

This subclause specifies processing that is used within multiple primitives. Each block of processing is defined as a function that is provided with input by some process flow, defined elsewhere, and returns outputs to be used within the invoking process flow. The description of each function consists of its name, the input, the outputs, the result of invocation, and example processing steps that give correct output. All function names start with "XXX-Function-", where XXX identifies the functional entity within the security services that implements the function.

7.8.2 CME-Function-ConstructCertificateChain

7.8.2.1 Input

Name	Type	Valid range	Description
Identifier Type	Enumerated	Certificate Array, HashedId8, CertId10	Indicates the type of input data used to identify the certificate.
Identifier	Octet string	If *Identifier Type* is HashedId8 then CertID8, If *Identifier Type* is CertID10 then CertId10, Otherwise, unused	The HashedId8 or CertId10 identifying the certificate in question.
Input Certificate Array	An ordered array of 1609.2 certificates	Used if *Identifier Type* is "certificate array". Otherwise, unused.	The first certificate in the array, at *Input Certificate Array*[0], is the end-entity or CA certificate at the bottom of the chain.
Terminate At Root	Boolean	"True" or "False"	Whether the chain should be constructed to a root (True) or only to a trust anchor (False). If not specified, defaults to False.
Maximum Chain Length (optional)	Integer	Any positive integer ≥ 2	The maximum length the chain may have.

7.8.2.2 Output

Name	Type	Valid range	Description
Result Code	Enumerated	Success, Certificate not found, Not enough information to construct chain, Chain ended at unknown root, Chain was too long, Any value returned by CME-Certificate-Info.confirm	The result of the certificate chain construction.
Certificate Chain	Array of 1609.2 certificates	Array of 1609.2 certificates	If *Result Code* is "found", the constructed certificate chain. Otherwise, undefined.
Permissions Array	Array of types, each of which may be an array of Psid, array of PsidPriority, array of PsidSsp, or array of PsidPrioritySsp	Any	If *Result Code* is "found", the permissions associated with the certificate, either inherited or explicit. Otherwise, undefined.
Geographic Scopes	Array of geographic regions	Any	If *Result Code* is "found", the geographic scope associated with the certificate, either inherited or explicit. Otherwise, undefined.

Last Received CRL Times	An array within which each value is a date or "none"	Each entry may be "none" or any date in the past	For each certificate in *Certificate Chain*, the generation time of the last CRL received that would have contained the certificate if it had been revoked.
Next Expected CRL Times	An array within which each value is a date or "unknown"	Each entry may be any date after the generation time of the last CRL received for the associated certificate. The date may be in the past or future	For each certificate in *Certificate Chain*, the next time a CRL is intended to be generated that could contain the certificate.
Verified	Array of Booleans	Any	Whether or not each certificate in the chain has been cryptographically verified.

7.8.2.3 Summary

This function constructs a certificate chain from the certificate array provided as input to a trust anchor, if all the certificates in the chain are available. The process is illustrated in Figure D.19.

NOTE—This function merely constructs the chain; it does not check the chain for consistency.

7.8.2.4 Processing

The following processing may be used to produce correct output:

NOTE—An implementation may cache the results of this processing for performance reasons.

a) Create variables to be used to return values to the caller:
   1) An enumerated value, *Result Code*, initialized to "Success".
   2) A certificate chain, *Certificate Chain*.
   3) An array, *Permissions Array,* with elements *Psid*, *PsidPriority*, *PsidSsp*, or *PsidPrioritySsp*.
   4) An array of geographic regions, *Geographic Scopes*.
   5) A time array, *Last Received CRL Times*.
   6) A time array, *Next Expected CRL Times.*
   7) A Boolean, *Verified*.

b) Create internal variables:
   1) A certificate, *certificate*, initialized to null.
   2) An integer, *i*, initialized to 0.
   3) An octet string, *signId*.
   4) A Boolean, *trustAnchor*.

c) If *Identifier Type* is "Certificate Array", set *certificate* equal to *Input Certificate Array*[0]. Otherwise set *signId* equal to *Identifier*.

d) Do until one of the return statements is hit:

1) If *i* is not 0:

   i) Set *signId* equal to the `signer_id` field from the `signer` field of the ToBeSignedCertificate within *certificate*.

   ii) Set *certificate* to null.

2) Invoke the CME-CertificateInfo.request primitive, with the following parameters:

   i) Identifier Type = CertId8, Identifier = *signId* if *certificate* is null

   ii) Identifier Type = Certificate, Identifier = *certificate* if *certificate* is not null.

   Set *Result Code*, *certificate*, *Permissions Array*[*i*], *Geographic Scopes*[*i*], *Last Receive CRL Times*[*i*], *Next Expected CRL Times*[*i*], *trustAnchor*, and *Verified*[*i*] to the parameters *Result Code*, *Certificate Data*, *Permissions*, *Geographic Scope*, *Last Received CRL Time*, *Next Expected CRL Time*, *Trust Anchor* and *Verified* returned by CME-CertificateInfo.confirm.

3) If *Result Code* is neither "found" nor "certificate not found", return.

4) (If the certificate is identified only by a CertId8 or CertId10 and was not found by the CME, look for the certificate in the input array, if one was provided): If *Result Code* is "Certificate not found" and *certificate* is null:

   i) If Identifier Type is not "certificate array", set *Result Code* to "not enough information to construct chain" and return.

   ii) For each certificate *cTemp* in *Input Certificate Array*, if the HashedId8 of *cTemp* (obtained as described in the specification of HashedId8) is equal to *signId*:

      i) If the type of *cTemp* is `root_ca`, set *Result Code* to "chain ended at unknown root" and return.[26]

      ii) Set *certificate* equal to *cTemp* and return to step d)2) without incrementing *i*.

   iii) If this step is reached (i.e. if *certificate* is still null), set *Result Code* to "not enough information to construct chain" and return.

5) (If no information about the certificate is held by the CME) If *certificate* is not null and *Result Code* is "Certificate not found", set *Verified*[i] to "False".

6) Set *Certificate Chain*[*i*] = *certificate*.

7) Set *i* = *i*+1. If *Maximum Chain Length*, was provided as an input and *i* is greater than *Maximum Chain Length*, set *Result Code* to "Chain was too long" and return.

8) If *Terminate at Root* is "False":

   i) If *Trust Anchor* is not true, go back to d).

9) If *Terminate at Root* is "True":

   i) If *certificate*.`unsigned_certificate.holder_type` is not `root_ca`, go back to d).

10) (*Trust Anchor* is true, i.e. the chain has been completed): Set *Result Code* to "success" and return.

[26] If a certificate chain does not end at a root known to the recipient, the recipient cannot verify that the chain is trustworthy. If processing reaches this line, it means that the certificate chain has terminated (the current issuing cert has type `root_ca`) but at a root that the recipient does not know (because if the recipient did know the root, it would have found it in step d)3)).

7.8.3 Sec-Function-CheckCertificateChainConsistency

7.8.3.1 Input

Name	Type	Valid range	Description
Certificate Chain	Array of 1609.2 certificates	Array of 1609.2 certificates	The constructed certificate chain as returned by CME-Function-ConstructCertificateChain.
Permissions Array	Array of types, each of which may be an array of Psid, array of PsidPriority, array of PsidSsp, or array of PsidPrioritySsp	Any	The permissions associated with each certificate, either inherited or explicit.
Geographic Scopes	Array of geographic regions	Any	The geographic scope associated with the certificate, either inherited or explicit.

7.8.3.2 Output

Name	Type	Valid range	Description
Result Code	Enumerated	Success, Incorrect CA certificate type, Inconsistent certificate holder type, Non-revocable, non-expiring certificate, Inconsistent expiry times, Inconsistent start times, or Any *Result Code* value that might be returned by Sec-Function-CheckChainPsidsConsistency, Sec-Function-CheckChainPsid-PriorityConsistency, or Sec-Function-CheckChainGeographicConsistency.	The result of the certificate chain consistency verification.

7.8.3.3 Summary

This function determines whether an input certificate chain created by CME-Function-ConstructCertificateChain is consistent (i.e., that all of its certificates are pairwise consistent as illustrated in Figure 29).

NOTE—This function checks the chain for consistency but does not cryptographically verify it or check that it is valid at the current location or time.

7.8.3.4 Processing

The following processing may be used to produce correct output.

a) Create variables to be used to return values to the caller:
   1) An enumerated value, *Result Code*, initialized to “found”.

2) A certificate chain, *Certificate Chain*.

3) An array, *Permissions Array,* with elements *Psid*, *PsidPriority*, *PsidSsp*, or *PsidPrioritySsp*.

4) An array of geographic regions, *Geographic Scopes*.

b) Check that all the issuing certificates in the certificate chain contain the `holder_type` of their subordinate certificate in their `scope.permitted_holder_types` field. If this condition is not met, set *Result Code* to "inconsistent certificate holder type" and return *Result Code*.

c) If *Permissions Array*[0] is an array of Psid or array of PsidSsp:

1) Invoke Sec-Function-CheckChainPsidsConsistency with input Permissions Array. Set *Result Code* to the output *Result Code* from Sec-Function-CheckChainPsidsConsistency. If *Result Code* is not "success", return *Result Code*.

2) Check that all the certificates in the certificate chain except for the signing certificate and the trust anchor have `holder_type` equal to `sde_ca`. If this condition is not met, set *Result Code* to "incorrect CA certificate type" and return *Result Code*.

d) If *Permissions Array*[0] is an array of PsidPriority, or array of PsidPrioritySsp:

1) Invoke Sec-Function-CheckChainPsidPriorityConsistency with input Permissions Array. Set *Result Code* to the output *Result Code* from Sec-Function-CheckChainPsidPriority-Consistency. If *Result Code* is not "success", return *Result Code*.

2) Check that all the certificates in the certificate chain except for the signing certificate and the trust anchor have `holder_type` equal to `wsa_ca`. If this condition is not met, set *Result Code* to "incorrect CA certificate type" and return *Result Code*.

e) Check that all the certificates in the chain have a non-zero value in at least one of the `expiration` field and the `crl_series` field. If this is not the case, set *Result Code* to "non-revocable, non-expiring certificate" and return *Result Code*.

f) Verify that each certificate in the chain has `expiration` equal to or earlier than the expiration time of its issuer. If not, set *Result Code* to "inconsistent expiry times" and return *Result Code*.

g) Verify that each certificate in the chain has `start_validity` , if present, is equal to or earlier than the expiration time of its issuer. If not, set *Result Code* to "inconsistent start times" and return *Result Code*.

h) Verify that for each certificate in the chain, if start_validity is present, it is equal to or earlier than the expiration value in the same certificate. If not, set *Result Code* to "start validity later than expiration" and return *Result Code.*

i) Invoke Sec-Function-CheckChainGeographicConsistency with input *Geographic Scopes*. Set *Result Code* to the output *Result Code* from Sec-Function-CheckChainGeographicConsistency.

j) Return *Result Code*.

NOTE 1—If the security services have an assurance that all CAs in the chain have processes in place to prevent issuing certificates with inconsistent permissions, the consistency checks may be skipped.

NOTE 2—If the security services have an assurance that all CAs in the chain have processes in place to prevent issuing certificates with inconsistent geographic scopes, the processing in step g) may be skipped.

NOTE 3—The CME may choose to maintain a record of certificates that are known to be invalid in order to speed up rejection of data signed with a certificate that chains back to an invalid certificate. In this case, it may choose to continue checking the consistency of chains even after an inconsistency has been found, in order to detect other inconsistencies that may exist higher up the chain and thereby detect more invalid certificates.

7.8.4 Sec-Function-CheckChainPsidsConsistency

7.8.4.1 Input

Name	Type	Valid range	Description
Permissions Array	Array of types, each of which may be an array of Psid, or an array of PsidSsp.	Any	The permissions associated with each certificate.

7.8.4.2 Output

Name	Type	Valid range	Description
Result Code	Enumerated	Success, Invalid permission type, Inconsistent permissions	The result of the consistency validation.

7.8.4.3 Summary

This function takes as input an array of permissions returned by CME-Function-ConstructCertificateChain. The permissions are expected all to be of type Psid or PsidSsp. The function determines whether they are of that type and whether they are consistent.

7.8.4.4 Processing

The following processing produces correct output:

a) If any of the entries in *Permissions Array*[0] is of a type other than Psid or PsidSsp, set *Result Code* to "invalid permission type" and return.

b) For $i = 1$ to (one less than the number of entries in *Permissions Array*):

   1) If any of the entries in *Permissions Array*[i] is of type other than Psid, set *Result Code* to "invalid permission type" and return.

   2) For every entry in *Permissions Array*[i–1], if it is not also contained in *Permissions Array* [i], set *Result Code* to "inconsistent permissions" and return.

   3) Set $i = i+1$.

b) Set *Result Code* to "success" and return.

7.8.5 Sec-Function-CheckChainPsidPriorityConsistency

7.8.5.1 Input

Name	Type	Valid range	Description
Permissions Array	Array of types, each of which may be an array of PsidPriority or an array of PsidPrioritySsp	Any	The permissions associated with each certificate.

7.8.5.2 Output

Name	Type	Valid range	Description
Result Code	Enumerated	Success, Invalid permission type, Inconsistent permissions	The result of the consistency validation

7.8.5.3 Summary

This function takes as input an array of permissions returned by CME-Function-ConstructCertificateChain. The permissions are expected all to be of type PsidPriority or PsidPrioritySsp. The function determines whether they are of that type and whether they are consistent.

7.8.5.4 Processing

The following processing produces correct output:

a) If any of the entries in *Permissions Array*[0] is of a type other than PsidPriority or PsidPrioritySsp, set *Result Code* to "invalid permission type" and return.

b) For i = 1 to (one less than the number of entries in *Permissions Array*):

   1) If any of the entries in *Permissions Array*[i] is of type other than PsidPriority or PsidPrioritySsp, set *Result Code* to "invalid permission type" and return.

   1) For every PsidPriority *pp* in *Permissions Array*[i],

      iii) Set the PsidPriority *pp*' equal to the PsidPriority in *Permissions Array* [i+1] that contains the same Psid value. If there is no such PsidPriority, set *Result Code* to "inconsistent permissions" and return.

      iv) If *pp*.`max_priority` is greater than *pp*'.`max_priority`, set *Result Code* to "inconsistent permissions" and return.

      v) Move to the next PsidPriority.

   2) Set $i = i$+1.

c) Set *Result Code* to "success" and return.

7.8.6 Sec-Function-CheckChainGeographicConsistency

7.8.6.1 Input

Name	Type	Valid range	Description
Geographic Scopes	Array of geographic regions	Any	The geographic scope associated with the certificate, either inherited or explicit.

7.8.6.2 Output

Name	Type	Valid range	Description
Result Code	Enumerated	Success, Inconsistent geographic scopes	The result of the consistency validation.

7.8.6.3 Summary

This function takes as input an array of geographic scopes returned by CME-Function-ConstructCertificateChain. The function determines whether they are consistent.

7.8.6.4 Processing

The following processing produces correct output:

a) For $i = 1$ to (one less than the number of entries in *Geographic Scopes*):
   1) If any part of *Geographic Scopes*[i–1] falls outside the region specified in *Geographic Scopes*[i], set *Result Code* to "inconsistent geographic scope" and return.
   2) Set $i = i+1$.

b) Set *Result Code* to "success" and return.

7.8.7 Sec-Function-VerifyChainAndSignature

7.8.7.1 Input

Name	Type	Valid range	Description
Certificate Chain	Array of 1609.2 certificates	Array of 1609.2 certificates	The constructed certificate chain as returned by CME-Function-ConstructCertificateChain.
Verified	Array of Booleans	An array the same length as *Certificate Chain*	Information about whether certificates are known to have been verified, returned by CME-Function-ConstructCertificateChain.
Digest	Octet string, or NULL	Any octet string of length corresponding to the appropriate hash function for the signature algorithm, or NULL	A hash of the data to be verified, or NULL.
Signature	Signature, or NULL	Any, or NULL	The signature to be verified, or NULL.

7.8.7.2 Output

Name	Type	Valid range	Description
Result Code	Enumerated	Success; Certificate verification failed; Verification failed, or Any *Result Code* value returned from CME-AddCertificate.request.	The result of the cryptographic verification.

7.8.7.3 Summary

This function verifies that all the signatures within a chain are valid. If *Digest* and *Signature* are provided, the function also verifies that *Signature* is a valid signature of *Digest* using the public key indicated by the end-entity certificate in the chain.

7.8.7.4 Processing

The following processing ensures correct output:

a) Create variables to be used to return values to the caller:
   1) An enumerated value, *Result Code*, initialized to "success".
   2) An array of Booleans, *Verified*.
   3) An octet string, *Digest*
   4) A signature, *Signature*.

b) For each *Certificate Chain*[*i*], if *Verified*[i] is False and the certificate is explicit, verify the signature with the public key from the `verification_key` field of the issuing certificate.

   If any of the verifications fail, or if any of the public keys have an unknown value for PKAlgorithm, set *Result Code* to "certificate verification failed" and return.

c) If *Signature* or *Digest* are NULL, set *Result Code* to "success" and return.

d) If *Certificate Chain*[0] is an explicit certificate, verify that *Signature* is a valid signature on *Digest* using *Certificate Chain*[0].`verification_key`.

   If the signature does not verify, set Result Code to "verification failed" and return.

e) If the leading certificates in the chain are implicit certificates:
   1) Carry out the key extraction process algorithm described in 3.5 of SEC 4, where the parameters *CertU* is the certificate and the hash function *H* of a certificate is defined as the hash of the associated PublicKeyReconstructionHashInput as described in 6.3.2.
   2) Verify the signature with the resulting public key as specified in 5.8.1. If the signature does not verify, set Result Code to "verification failed" and return.

f) For each *Certificate Chain*[*i*] for which *Verified*[*i*] is "False" invoke CME-AddCertificate.request with parameters *Certificate* = *Certificate Chain*[*i*], *Verified* = true.

NOTE—If some of the certificates in the chain were implicit, the public key extraction may be combined with ECDSA verification as discussed in Annex B of SEC 4. Alternatively, the CME may choose to recover the public keys for any or all of the implicit certificates, and cache those keys. This may improve performance when verifying subsequent signatures from the same sender.

7.8.8 Sec-Function-DecryptData

7.8.8.1 Input

Name	Type	Valid range	Description
Data	Octet string	Any	The EncryptedData to be decrypted.
Cryptomaterial Handle	Integer	Any	CMH in *Key Pair Only* or *Key and Certificate* state, where the private key is for a decryption algorithm.

7.8.8.2 Output

Name	Type	Valid range	Description
Result Code	Enumerated	Success, No decryption certificate found, Couldn't decrypt key, Couldn't decrypt	The result of the signing operation.
Content Type	Enumerated	Unsecured, Signed, Certificate request, Certificate response, Anonymous certificate response, Certificate request error, CRL request, CRL, Signed partial payload, Signed external payload, Certificate response acknowledgment	If *Result Code* is "success", the type of the decrypted content. Otherwise, undefined.
Data	Octet string	Any	If *Result Code* is "success", the decrypted data. Otherwise, undefined.

7.8.8.3 Summary

This function determines whether any of the keys provided at *Cryptomaterial Handles* is suitable to decrypt the *Data*. If one of the keys is appropriate, the function decrypts the data and returns the data from the resulting ToBeEncrypted to the caller.

7.8.8.4 Processing

The following processing produces correct output:

a) Create variables to be used to return data to the caller:
   1) An enumerated value, *Result Code*, initialized to "success".
   2) An enumerated value, *Content Type*.
   3) An octet string, *Data*.
   4) An integer, *Cryptomaterial Handle.*
b) Create a RecipientInfo, *ri*, initialized to "undefined".
c) For each *Data*.`recipients`[*i*]:
   1) Set *cId* equal to *Data*.`recipients`[*i*].`cert_id`.
   2) If *Cryptomaterial Handle* is in *Key and Certificate* state:
      i) If the HashedId8 of *Cryptomaterial Handle.certificate* is equal to *cId*, set *ri* equal to *Data*.`recipients`[*i*].
      ii) If *Data*.`symm_alg` field does not match one of the algorithms identified in *Cryptomaterial Handle.certificate*.`unsigned_certificate.encryption_key-.supported_symm_alg`, set *ri* to "undefined".
   3) If *Cryptomaterial Handle* is in *Key Pair Only* state:
      i) If the low-order eight bytes of the hash of *Cryptomaterial Handle.publicKey* is equal to *cId*, set *ri* equal to *Data*.`recipients`[*i*].
   4) If *ri* is "undefined", move to the next RecipientInfo and return to step c)1). Otherwise, continue to the next step.
d) If *ri* is still "undefined", set *Result Code* to "no decryption certificate found" and return.
e) Decrypt *ri*.`enc_key` using the associated asymmetric decryption algorithm:
   1) ECIES decryption is performed as specified in 5.8.2.
   2) If the decryption of the symmetric key fails, set *Result Code* to "couldn't decrypt key" and return.
f) Decrypt and authenticate the `ciphertext` using the extracted symmetric key:
   1) Decryption with AES-CCM is performed as specified in 5.8.3.
   2) If the decryption process outputs "failure", set *Result Code* to "couldn't decrypt" and return.
g) Set *Content Type* and *Data* as indicated by the decrypted ToBeEncrypted.
h) Set *Result Code* to "success" and return.

7.8.9 Sec-Function-CertificateRequestErrrorVerification

7.8.9.1 Input

Name	Type	Valid range	Description
Certificate Request Error	AToBeEncrypted-CertificateRequestError	Any	The ToBeEncryptedCertificateRequestError to be processed.

7.8.9.2 Output

Name	Type	Valid range	Description
Result Code	Enumerated	Success, Invalid CA signature algorithm, Overdue CRL, or Any *Result Code* value that might be returned by CME-Function-ConstructCertificateChain, Sec-Function-CheckCertificateChain-Consistency, or Sec-Function-VerifyChainAndSignature.	The result of the processing.

7.8.9.3 Summary

This function verifies a certificate request error message and returns the result of the verification to the caller.

7.8.9.4 Processing

The following processing ensures correct output:

a) Create variables to be used to return data to the caller:
   1) An enumerated value, *Result Code*, initialized to "success".
   2) A ToBeEncrypted-CertificateRequestError, *Certificate Request Error*.

b) Create variables to be used for internal processing:
   1) A certificate chain, *chain.*
   2) An array of permissions, *permissions.*
   3) An array of geographic scopes, *geoScopes.*
   4) An array of times, *Last Received CRL Times.*
   5) An array of times, *Next Expected CRL Times.*
   6) An array of Booleans, *verified.*

c) If *Certificate Request Error*.`signer_identifier.type` is `certificate_digest_with_ecdsap224`, set *Result Code* to "invalid CA signature algorithm" and return.

d) Construct the certificate chain by invoking CME-Function-ConstructCertificateChain with the following inputs:
   3) If *Certificate Request Error*.`signer_identifier.type` is `certificate_digest_with_ecdsap256`:
      i) *Identifier Type* = CertId8,
      ii) *Identifier* = *Certificate Request Error*.`signer_identifier.digest`.
   4) If *Certificate Request Error*.`signer_identifier.type` is `certificate`:
      i) *Identifier Type* = Certificate Array,
      ii) *Identifer* = an array of length 1 containing *Certificate Request Error*.`signer_identifier.certificate`.

5) If *Certificate Request Error*.`signer_identifier.type` is `certificate_chain`:

i) *Identifier Type* = Certificate Array,

ii) *Identifer* = *Certificate Request Error*.`signer_identifier.certificates`.

Set *Result Code*, *chain*, *permissions*, *geoScopes*, *Last Received CRL Times*, *Next Expected CRL Times* and *verified* respectively to the output values from CME-Function-ConstructCertificateChain: *Result Code*, *Certificate Chain*, *Permissions Array*, *Geographic Scopes*, *Last Received CRL Times*, *Next Expected CRL Times*, and *Verified.*
If *Result Code* is not "success", return *Result Code*.

e) For each value in the array *Next Expected CRL Times*, if it is in the past, set *Result Code* to "overdue CRL" and return *Result Code*.

f) Check the internal consistency of the certificate chain:

1) Invoke Sec-Function-CheckCertificateChainConsistency with parameters *Certificate Chain* = *chain*, *Permission Array* = *permissions*, *Geographic Scopes* = *geoScopes*. Set *Result Code* to the output value *Result Code*.

If *Result Code* is not "success", return *Result Code*.

g) Verify the certificate chain and signature:

1) Encode *Certificate Request Error*.`request_hash` and *Certificate Request Error*.`reason`, as specified in 6.3.37. Set *digest* equal to the hash of this encoding, calculated with the mechanism associated with the verification key.

2) Invoke Sec-Function-VerifyChainAndSignature with inputs *Certificate Chain* = *chain*, *Verified* = *verified*, *Digest* = *digest*, *Signature* = *Signed Data*.`signature`. Set *Result Code* to the output value *Result Code*.

h) Return *Result Code*.

7.8.10 Sec-Function-CertificateResponseVerification

7.8.10.1 Input

Name	Type	Valid range	Description
Certificate Response	A ToBeEncrypted-CertificateResponse	Any	The ToBeEncryptedCertificateResponse to be processed.

7.8.10.2 Output

Name	Type	Valid range	Description
Result Code	Enumerated	Success, Overdue CRL, No relevant CRL provided, or Any Result Code value returned by Sec-CRLVerification.confirm, CME-Function-ConstructCertificateChain, Sec-Function-CheckCertificate-ChainConsistency, or Sec-Function-VerifyChainAndSignature.	The result of the processing.

7.8.10.3 Summary

This function verifies a certificate response message and returns the result of the verification to the caller.

7.8.10.4 Processing

The following processing ensures correct output:

a) Create variables to be used to return data to the caller:
   1) An enumerated value, *Result Code*, initialized to "success".
   2) A ToBeEncrypted-CertificateResponse, *Certificate Response*.

b) Create variables to be used for internal processing:
   1) A certificate chain, *chain.*
   2) An array of permissions, *permissions.*
   3) An array of geographic scopes, *geoScopes.*
   4) An array of times, *Last Received CRL Times.*
   5) An array of times, *Next Expected CRL Times.*
   6) An array of Booleans, *verified.*

c) For each CRL in *Certificate Response*.`crl_path`[*i*]:
   1) Invoke Sec-CRLVerification.request with inputs *CRL* = *Certificate Response*.`crl_path`[*i*], *Overdue CRL Tolerance* = 0. Set *Result Code* to the value returned by Sec-CRLVerification.confirm.
   2) If *Result Code* is not "success", return *Result Code*.

d) Construct the certificate chain by invoking CME-Function-ConstructCertificateChain with the inputs
   6) *Identifier Type* = Certificate Array,
   7) *Identifier* = *Certificate Response*.`certificate_chain`

   Set *Result Code*, *chain*, *permissions*, *geoScopes*, *Last Received CRL Times*, *Next Expected CRL Times* and *verified* respectively to the output values from CME-Function-ConstructCertificateChain: *Result Code*, *Certificate Chain*, *Permissions Array*, *Geographic Scopes*, *Last Receive CRL Times*, *Next Expected CRL Time*, and *Verified.*
   If *Result Code* is not "success", return *Result Code*.

e) For each value in the array *Next Expected CRL Times*, if it is in the past, set *Result Code* to "overdue CRL" and return *Result Code*.

f) Check the internal consistency of the certificate chain:
   1) Invoke Sec-Function-CheckCertificateChainConsistency with parameters *Certificate Chain* = *chain*, *Permission Array* = *permissions*, *Geographic Scopes* = *geoScopes*. Set *Result Code* to the output value *Result Code*.

      If *Result Code* is not "success", return *Result Code*.

g) Verify that the crl_path field contains a CRL such that the `ca_id` field is the same as the signer_id field in the issued certificate, and both the CRL and the issued certificate have the same value in the crl_series field. If this fails, set *Result Code* to "no relevant CRL provided" and return *Result Code.*

h) Verify the certificate chain and signature:

1) Encode *Certificate Request Error*.`request_hash` and *Certificate Request Error*.`reason`, as specified in 6.3.37. Set *digest* equal to the hash of this encoding, calculated with the mechanism associated with the verification key.

2) Invoke Sec-Function-VerifyChainAndSignature with inputs *Certificate Chain* = *chain*, *Verified* = *verified*, *Digest* = *digest*, *Signature* = *Signed Data*.`signature`. Set *Result Code* to the output value *Result Code*.

i) Return *Result Code*.

NOTE—Step c)1) above adds all of the information from the CRLs to the CME.

Annex A

(normative)

Protocol Implementation Conformance Statement (PICS) prforma

A.1 Instructions for completing the PICS proforma

A.1.1 General structure of the PICS proforma

The first parts of the PICS proforma, Implementation identification and Protocol summary, are to be completed as indicated with the information necessary to identify fully both the supplier and the implementation.

The main part of the PICS proforma is a fixed questionnaire, divided into subclauses, each containing a number of individual items. Answers to the questionnaire items are to be provided in the rightmost column, either by simply marking an answer to indicate a restricted choice (usually Yes or No) or by entering a value or a set or a range of values. (Note that there are some items where two or more choices from a set of possible answers may apply. All relevant choices are to be marked in these cases.)

Each item is identified by an item reference in the first column. The second column contains the question to be answered. The third column contains the reference or references to the material that specifies the item in the main body of this standard. The remaining columns record the status of each item (i.e., whether support is mandatory, optional, or conditional) and provide the space for the answers. Marking an item as supported is to be interpreted as a statement that all relevant requirements of the subclauses and normative annexes, cited in the References column for the item, are met by the implementation.

A supplier may also provide, or be required to provide, further information, categorized as either Additional Information or Exception Information. When present, each kind of further information is to be provided in a further subclause of items labeled A<I> or X<I>, respectively, for cross-referencing purposes, where <I> is any unambiguous identification for the item (e.g., simply a numeral). There are no other restrictions on its format or presentation.

A completed PICS proforma, including any Additional Information and Exception Information, is the PICS for the implementation in question.

NOTE—Where an implementation is capable of being configured in more than one way, a single PICS may be able to describe all such configurations. However, the supplier has the choice of providing more than one PICS, each covering some subset of the implementation's capabilities, if this makes for easier and clearer presentation of the information.

A.1.2 Additional information

Items of Additional Information allow a supplier to provide further information intended to assist in the interpretation of the PICS. It is not intended or expected that a large quantity of information will be supplied, and a PICS can be considered complete without any such information. Examples of such Additional Information might be an outline of the ways in which a (single) implementation can be set up to operate in a variety of environments and configurations, or information about aspects of the implementation that are outside the scope of this standard but have a bearing upon the answers to some items.

References to items of Additional Information may be entered next to any answer in the questionnaire, and may be included in items of Exception Information.

A.1.3 Exception information

It may happen occasionally that a supplier will wish to answer an item with mandatory status (after any conditions have been applied) in a way that conflicts with the indicated requirement. No preprinted answer will be found in the Support column for this. Instead, the supplier shall write the missing answer into the Support column, together with an X<I> reference to an item of Exception Information, and shall provide the appropriate rationale in the Exception Information item itself.

An implementation for which an Exception Information item is required in this way does not conform to this standard.

NOTE—A possible reason for the situation described above is that a defect in this standard has been reported, a correction for which is expected to change the requirement not met by the implementation.

A.1.4 Conditional status

The PICS proforma contains a number of conditional items. These are items for which both the applicability of the item itself, and its status if it does apply, mandatory or optional, are dependent upon whether or not certain other items are supported.

A conditional symbol is of the form "<pred>:<S>", where <pred> is a predicate as described below, and <S> is one of the status symbols M or O.

If the value of the predicate is true, the conditional item is applicable, and its status is given by S: the support column is to be completed in the usual way. Otherwise, the conditional item is not relevant.

A predicate is one of the following:

— An item-reference for an item in the PICS proforma: the value of the predicate is true if the item is marked as supported, and is false otherwise.

— A Boolean expression constructed by combining item-references using the Boolean operator OR: the value of the predicate is true if one or more of the items is marked as supported, and is false otherwise. For compactness, item-references combined with a comma are considered to be combined with the OR operator.

Each item referenced in a predicate, or in a preliminary question for grouped conditional items, is indicated by an asterisk (*) in the Item column.

A status of O<n> indicates a mutual conditionality such that the feature is optional but that support of at least one of the items that have status O<n> is mandatory.

A status of C<n> indicates a mutual conditionality such that support of one and one only of the items that have status C<n> is mandatory.

A.2 PICS proforma—IEEE Std 1609.2[27]

A.2.1 Identification

Only the first three items are required for all implementations. Other information may be completed as appropriate in meeting the requirement for full identification.

The terms *Name* and *Version* should be interpreted appropriately to correspond with a supplier's terminology (e.g., Type, Series, Model).

Supplier	
Contact point for queries about the PICS	
Implementation Name(s) and Version(s)	
Other information necessary for full identification, [e.g., name(s) and version(s) of the machines and/or operating systems(s), system names]	

[27] Copyright release for PICS proforma: Users of this standard may freely reproduce the PICS proforma in this annex so that it can be used for its intended purpose and may further publish the completed PICS.

A.2.2 Protocol summary

Identification of protocol standard	IEEE Std 1609.2
Identification of amendments and corrigenda to this PICS proforma that have been completed as part of this PICS	Amd. : Corr. : Amd. : Corr. :
Have any exception items been required? (The answer Yes means that the implementation does not conform to IEEE Std 1609.2)	□Yes □No
Date of statement (dd/mm/yy)	

A.2.3 Conformance statement

This presents a list of the security functionality that an implementation may claim to support.

Item	Security configuration (top-level)	Reference	Status	Support
S1.	**Support 1609.2**		M	□Yes □No
S2.	**Generate Secure Data**			□Yes □No
S2.1.	Create 1609Dot2Data containing valid SignedData	4.3.1, 5.5, 7.2.13	S1:O1	□Yes □No
S2.1.1.	… with internal payload	7.2.13	S2.1:O2.1	□Yes □No
S2.1.2.	… with external payload	7.2.13	S2.1:O2.1	□Yes □No
S2.1.3.	… with partial payload	7.2.13	S2.1:O2.1	□Yes □No
S2.1.4.	Ensure that certificate used to sign data is valid (part of a consistent chain, valid at the current time and location)	5.5	S2.1:M	□Yes □No
S2.1.4.1.	Support signing with certificates containing start validity	7.2.13, 7.5.1	O	□Yes □No
S2.1.4.2.	Support signing with certificates containing lifetime as duration	7.2.13, 7.5.1	O	□Yes □No
S2.1.4.3.	Maximum number of entries in `permissions_list` field of a signing certificate	7.2.13, 7.5.1	S2.1: 32:M >32:O	Enter number: ( )
S2.1.4.4.	Sign with certificate containing circular GeographicRegion	7.2.13, 7.5.1	O	□Yes □No
S2.1.4.5.	Sign with certificate containing rectangular GeographicRegion	7.2.13, 7.5.1	O	□Yes □No
S2.1.4.5.1.	Maximum number of RectangularRegions supported in a signing certificate	6.3.14, 7.2.13, 7.5.1	S2.1.4.5: 6:M >6:O	Enter number: ( )
S2.1.4.6.	Sign with certificate containing polygonal GeographicRegion in certificate	7.2.13, 7.5.1	O	□Yes □No
S2.1.4.6.1.	Maximum number of PolygonalRegion vertices supported in a signing certificate	6.3.18, 7.2.13, 7.5.1	S2.1.4.6: 12:M >12:O	Enter number: ( )

Item	Security configuration (top-level)	Reference	Status	Support
S2.1.5.	Ensure that key and certificate used to sign are a valid pair	5.8.4	S2.1:M	□Yes □No
S2.1.6.	Use certificates of type anonymous	7.2.13	S2.1:O2.2	□Yes □No
S2.1.7.	… of type identified	7.2.13	S2.1:O2.2	□Yes □No
S2.1.8.	… of type identified not localized	7.2.13	S2.1:O2.2	□Yes □No
S2.1.9.	Include generation time in security headers	7.2.13	O	□Yes □No
S2.1.10.	Include generation location in security headers	7.2.13	O	□Yes □No
S2.1.11.	Include expiry time in security headers	7.2.13	O	□Yes □No
S2.1.12.	Support use of SignerIdentifierType certificate_chain	7.2.13	S2.1:M	□Yes □No
S2.1.12.1.	Maximum number of certificates that may be included in certificate chain sent with signed data	5.3.2, 7.8.2	S2.1: 2:M >2:O	Enter number: ( )
S2.1.13.	Support use of SignerIdentifierType certificate	7.2.13	O	□Yes □No
S2.1.14.	Support use of SignerIdentifierType certificate_digest	7.2.13	O	□Yes □No
S2.1.15.	Sign with ECDSA-224	7.2.13	S2.1:O2.3	□Yes □No
S2.1.16.	Sign with ECDSA-256	7.2.13	S2.1:O2.3	□Yes □No
S2.1.17.	Support signing with explicit certificates	7.2.13	S2.1:O2.4	□Yes □No
S2.1.18.	Support signing with implicit certificates	7.2.13	S2.1:O2.4	□Yes □No
S2.1.19.	Support signing with uncompressed points	7.2.13	S2.1:O2.5	□Yes □No
S2.1.20.	Support signing with compressed points	7.2.13	S2.1:O2.5	□Yes □No
S2.1.21.	Support signing with compressed fast verification information	7.2.13	O	□Yes □No
S2.1.22.	Support signing with uncompressed fast verification information	7.2.13	O	□Yes □No
S2.2.	Create 1609Dot2Data containing EncryptedData	7.2.15	S1:O1	□Yes □No
S2.2.1.	Create EncryptedData containing SignedData	7.2.15	O	□Yes □No
S2.2.2.	Maximum number of RecipientInfo supported in an outgoing EncryptedData	5.3.3, 7.2.15	S2.2: 6:M >6:O	Enter number: ( )
S2.2.3.	Use ECIES-256 as public-key encryption algorithm	7.2.15	S2.2:M	□Yes □No
S2.2.4.	Support encrypting to an encryption key included in an explicit cert	7.2.15	S2.2:O2.6	□Yes □No
S2.2.5.	Support encrypting to an encryption key included in an implicit cert	7.2.15	S2.2:O2.6	□Yes □No
S2.2.6.	Support encrypting to an uncompressed encryption key	7.2.15	S2.2:O2.7	□Yes □No
S2.2.7.	Support encrypting to a compressed encryption key	7.2.15	S2.2:O2.7	□Yes □No
S2.2.8.	Use AES-128 as symmetric encryption algorithm	7.2.15	S2.2:M	□Yes □No

Item	Security configuration (top-level)	Reference	Status	Support
S3.	**Receive secure data**			
S3.1.	Support use of Sec-SecureDataContent-Extraction.request or equivalent functionality to allow a secure communications entity to inspect the contents of data before verifying it	7.2.17	O	□Yes □No
S3.2.	Verify SignedData	7.2.19	S1:O1	□Yes □No
S3.2.1.	… with external payload	7.2.19	S3.2:M	□Yes □No
S3.2.2.	… with partial payload	7.2.19	S3.2:M	□Yes □No
S3.2.3.	Verify SignedData with ECDSA-224	7.2.19	S3.2:M	□Yes □No
S3.2.4.	Verify SignedData with ECDSA-256	7.2.19	S3.2:M	□Yes □No
S3.2.5.	Support receiving explicit end-entity certificates	7.2.19	S3.2:O3.1	□Yes □No
S3.2.6.	Support receiving implicit end-entity certificates	7.2.19	S3.2:O3.1	□Yes □No
S3.2.7.	Support explicit CA certificates	7.2.19	S3.2:M	□Yes □No
S3.2.8.	Support receiving implicit CA certificates	7.2.19	S3.2:O	□Yes □No
S3.2.9.	SignedData verification fails if trust anchor is not explicit certificate	5.5.2.1	S3.2:M	□Yes □No
S3.2.10.	SignedData verification fails if explicit certificate in chain is issued by implicit certificate	5.5.2.1	S3.2:M	□Yes □No
S3.2.11.	SignedData verification fails if data is inconsistent with the signing certificate	5.5.3.2	S3.2:M	□Yes □No
S3.2.12.	SignedData verification fails if the certificate chain is inconsistent	5.5.3.3	S3.2:M	□Yes □No
S3.2.13.	Receiver may specify relevance checks to be carried out	5.5.5	O	□Yes □No
S3.2.14.	Support receiving uncompressed points	7.2.19	S3.2:O3.2	□Yes □No
S3.2.15.	Support receiving compressed points	7.2.19	S3.2:O3.2	□Yes □No
S3.2.16.	Support fast verification with compressed fast verification information	7.2.19	O	□Yes □No
S3.2.17.	Support fast verification with uncompressed fast verification information	7.2.19	O	□Yes □No
S3.2.18.	Receiver may specify relevance checks to be carried out	5.5.5	O	□Yes □No
S3.2.18.1.	Allow receiver to Check Validity Based on Generation Time	7.2.19	S3.2:M	□Yes □No
S3.2.18.2.	Allow receiver to Check Validity Based on Generation Location	7.2.19	S3.2:M	□Yes □No
S3.2.18.3.	Allow receiver to Check Validity Based on Expiry Time	7.2.19	S3.2:M	□Yes □No
S3.2.18.4.	Allow receiver to reject replay	7.2.19	O	□Yes □No
S3.2.19.	Support use of SignerIdentifierType certificate_chain.	7.2.19	S3.2:M	□Yes □No
S3.2.19.1.	Maximum number of certificates that may be included in certificate chain sent with signed data	5.3.6, 7.2.19	S3.2: 1:M >1:O	Enter number: ( )

Item	Security configuration (top-level)	Reference	Status	Support
S3.2.19.2.	Maximum length of full certificate chain for signed data[28]	5.3.6, 7.2.19	S3.2: 1:M >1:O	Enter number: ( )
S3.2.19.3.	Maximum number of certificates that may be included in certificate chain sent with signed data	5.3.6, 7.2.19	S3.2: 1:M >1:O	Enter number: ( )
S3.2.20.	Support use of SignerIdentifierType certificate	7.2.19	S3.2:M	□Yes □No
S3.2.21.	Support use of SignerIdentifierType certificate_digest	7.2.19	S3.2:M	□Yes □No
S3.2.22.	Process certificates of type sde_identified_not_localized	7.2.19	S3.2:M	□Yes □No
S3.2.23.	Process certificates of type sde_identified_localized	7.2.19	S3.2:M	□Yes □No
S3.2.24.	Reject data if holder type in end-entity certificate is not sde_identified_not_localized, sde_identified_localized, or sde_anonymous	7.2.19	S3.2:M	□Yes □No
S3.2.25.	Reject data based on generation location being incompatible with certificate	7.2.19	S3.2:M	□Yes □No
S3.2.25.1.	Support circular GeographicRegion in certificate	7.2.19	S3.2:O3.3	□Yes □No
S3.2.25.2.	Support rectangular GeographicRegion in certificate	7.2.19	S3.2:O3.3	□Yes □No
S3.2.25.2.1.	Maximum number of RectangularRegions supported in a received certificate	6.3.14, 7.2.19, 7.5.1	S3.2: 6:M >6:O	Enter number: ( )
S3.2.25.3.	Support polygonal GeographicRegion in certificate	7.2.19	S3.2:O3.3	□Yes □No
S3.2.25.3.1.	Maximum number of PolygonalRegion vertices supported in a received certificate	6.3.18, 7.2.19, 7.5.1	S3.2: 12:M >12:O	Enter number: ( )
S3.2.25.4.	Support at least one certificate in the chain using a GeographicRegion of type from_issuer	7.2.19	S3.2:O	□Yes □No
S3.2.26.	Maximum number of entries in `permissions_list` field of a received certificate.	7.2.19, 7.5.1	S3.2: 8:M >8:O	Enter number: ( )
S3.2.27.	Correctly process incoming permissions of type from_issuer.	7.2.19, 7.5.1	S3.2:M	□Yes □No
S3.2.28.	Extract correct Service Specific Permissions from certificate	7.2.17.4	S3.2:M	□Yes □No
S4.	Decrypt EncryptedData	4.3.1, 7.2.17, 7.8.8	S1:O1	□Yes □No
S4.1.	Receive EncryptedData containing SignedData	7.2.17,	S4:M	□Yes □No

[28] The distinction between S3.2.19.2 and S3.2.19.3 is that S3.2.19.2 concerns the length of the (full or partial) certificate chain length that is transmitted, while S3.2.19.3 concerns the length of full certificate chain that is constructed when verifying.

Item	Security configuration (top-level)	Reference	Status	Support
S4.2.	Maximum number of RecipientInfos supported in an incoming EncryptedData	5.3.5, 7.2.17, 7.8.8	S4: 6:M >6:O	Enter number: ( )
S5.	**Signed WSA**			
S5.1.	Issue valid signed WSA	7.3.2	S1:O1	□Yes □No
S5.1.1.	Ensure that key and certificate used to sign are a valid pair	5.6.1	S5.1: M	□Yes □No
S5.1.2.	Maximum number of ServiceInfos in outgoing WSA	5.4.1, 7.3.2	S5.1: 32:M >32:O	Enter number: ( )
S5.1.3.	Support signing with explicit certificates	7.3.2	S5.1:M	□Yes □No
S5.1.4.	Support signing with uncompressed points	7.3.2	S5.1:M	□Yes □No
S5.1.5.	Support signing with no fast verification information	7.3.2	S5.1:M	□Yes □No
S5.2.	Receive Signed WSA	7.3.4	S1:O1	□Yes □No
S5.2.1.	Maximum number of ServiceInfos in received WSA	5.4.2, 7.3.4	S5.1: 32:M >32:O	Enter number: ( )
S5.2.2.	WSA verification fails if trust anchor is not explicit certificate	5.5.2.1	S5.2:M	□Yes □No
S5.2.3.	WSA verification fails if explicit certificate in chain is issued by implicit certificate	5.5.2.1	S5.2:M	□Yes □No
S5.2.4.	WSA verification fails if the certificate chain is inconsistent	5.5.3.3	S5.2:M	□Yes □No
S5.2.5.	Verify SignedWsa with ECDSA-256	7.3.4	S5.2:M	□Yes □No
S5.2.6.	Support receiving explicit CA certificates	7.3.4, 7.2.19	S5.2:M	□Yes □No
S5.2.7.	Support receiving implicit CA certificates	7.3.4, 7.2.19	S5.2:M	□Yes □No
S5.2.8.	Support receiving implicit end-entity certificates	7.3.4	S5.2:M	□Yes □No
S5.2.9.	Support receiving compressed points	7.3.4	S5.2:M	□Yes □No
S5.2.10.	Support fast verification with compressed fast verification information	7.3.4	O	□Yes □No
S5.2.11.	Extract generation time from SignedWsa	7.3.4	S5.2:M	□Yes □No
S5.2.12.	Extract generation location from SignedWsa	7.3.4	S5.2:M	□Yes □No
S5.2.13.	Extract expiry time from SignedWsa	7.3.4	S5.2:M	□Yes □No
S5.2.14.	Support use of SignerIdentifierType certificate_chain	7.3.4	S5.2:M	□Yes □No
S5.2.14.1.	Maximum certificate chain length supported on received WSA	5.4.2, 7.3.4, 7.8.8	S5.2: 5:M	Enter number: ( )
S5.2.15.	Reject Signed WSA if holder type in end-entity certificate is not wsa	7.3.4	S5.2:M	□Yes □No
S5.2.16.	Reject WSA based on generation location being incompatible with certificate	7.3.4	S5.2:M	
S5.2.16.1.	… with circular GeographicRegion in certificate	7.3.4, 7.2.19	S5.2:M	□Yes □No

Item	Security configuration (top-level)	Reference	Status	Support
S5.2.16.2.	… with rectangular GeographicRegion in certificate	7.3.4	S5.2:M	□Yes □No
S5.2.16.2.1.	Maximum number of RectangularRegions supported in a received certificate	6.3.14, 7.3.4, 7.5.1, 7.2.19	S5.2: 6:M >6:O	Enter number: ( )
S5.2.16.3.	… with polygonal GeographicRegion in certificate	7.3.4, 7.2.19	S5.2:M	□Yes □No
S5.2.16.3.1.	Maximum number of PolygonalRegion vertices supported in a received certificate	6.3.24, 7.3.4, 7.2.19, 7.5.1	S5.2: 12:M >12:O	Enter number: ( )
S5.2.16.4.	… with at least one certificate in the chain using a GeographicRegion of type from_issuer	7.3.4, 7.2.19	S5.2:M	□Yes □No
S5.2.17.	Reject WSA if a priority for a secured service in the WSA is greater than the priority allowed for that service in the certificate	7.3.4, 7.2.19	S5.2:M	□Yes □No
S5.2.18.	Maximum number of entries in permissions_list field of a received certificate	7.3.4, 7.5.1, 7.2.19	S5.2: 8:M >8:O	Enter number: ( )
S5.2.19.	Correctly process incoming permissions of type from_issuer	7.3.4, 7.5.1, 7.2.19	S5.2:M	□Yes □No
S5.2.20.	Extract correct Service Specific Permissions from end-entity certificate	7.3.4, 7.5.1, 7.2.19	S5.2:M	□Yes □No
S5.2.21.	Correctly process a WSA with some unsecured elements and some secured elements.	7.3.4, 7.2.19	S5.2:M	□Yes □No
S6.	**Certificate management**			
S6.1.	Generate certificate request	7.2.23	O	□Yes □No
S6.1.1.	Ensure that key and public key or certificate used to sign are a valid pair	5.6.1	S6.1: M	□Yes □No
S6.1.2.	Certificate request is self-signed	5.6.1.1, 7.2.23	S6.1: O6.11	□Yes □No
S6.1.2.1.	Ensure that public key in self-signed certificate request matches private key that signed it	5.8.4	S6.1.2: M	
S6.1.3.	Certificate request signed by enrolment certificate	5.6.1.1, 7.2.23	S6.1: O6.11	□Yes □No
S6.1.3.1.	Ensure that certificate request is consistent with enrolment certificate that signed it	5.6.1.2, 7.2.23	S6.1.3: M	□Yes □No
S6.1.3.2.	Sign with implicit enrolment certificate	7.2.23	S6.1.3: O6.12	□Yes □No
S6.1.3.3.	Sign with explicit enrolment certificate	7.2.23	S6.1.3:O6. 12	□Yes □No
S6.1.4.	Certificate request of type anonymous	7.2.23	S6.1: O6.13	□Yes □No
S6.1.5.	… of type identified	7.2.23	S6.1: O6.13	□Yes □No
S6.1.6.	… of type identified not localized	7.2.23	S6.1: O6.13	□Yes □No

Item	Security configuration (top-level)	Reference	Status	Support
S6.1.7.	… of type WSA signer	7.2.23	S6.1: O6.13	□Yes □No
S6.1.8.	… of type Secure Data Exchange Enrolment	7.2.23	S6.1: O6.13	□Yes □No
S6.1.9.	… of type WSA Enrolment	7.2.23	S6.1: O6.13	□Yes □No
S6.1.10.	Request explicit certificate	7.2.23	S6.1: O6.14	□Yes □No
S6.1.11.	Request implicit certificate	7.2.23	S6.1: O6.14	□Yes □No
S6.1.12.	Maximum number of entries in *Permissions Array* when requesting certificate	7.2.23	S6.1: 32:M >32:O	Enter number: ( )
S6.1.13.	Support circular GeographicRegion in certificate request	7.2.23	S6.1: O6.15	□Yes □No
S6.1.14.	Support rectangular GeographicRegion in certificate request	7.2.23	S6.1: O6.15	□Yes □No
S6.1.14.1.	Number of RectangularRegions supported in a certificate request	6.3.14, 7.2.23	S6.1.14: 6:M >6:O	Enter number: ( )
S6.1.15.	… with polygonal GeographicRegion in certificate	7.2.19	S6.1: O6.15	□Yes □No
S6.1.15.1.	Maximum number of PolygonalRegion vertices supported in a certificate request	6.3.18, 7.2.23	S6.1.15: 12:M >12:O	Enter number: ( )
S6.1.16.	Include start validity in certificate request	7.2.23	O	□Yes □No
S6.1.17.	Include lifetime as duration in certificate request	7.2.23	O	□Yes □No
S6.1.18.	Include expiration in certificate request	7.2.23	S6.1: M	□Yes □No
S6.1.19.	Verification public key is ECDSA-224	7.2.23	S6.1: O6.16	□Yes □No
S6.1.20.	Verification public key is ECDSA-256	7.2.23	S6.1: O6.16 S6.1.11:M	□Yes □No
S6.1.21.	Include encryption key in certificate request	7.2.23	O	□Yes □No
S6.1.21.1.	Encryption public key is ECIES-256	7.2.23	S6.1.21: M	□Yes □No
S6.1.22.	Response encryption key is ECIES-256	7.2.23	S6.1.21: M	□Yes □No
S6.1.23.	Encrypt request to explicit CA certificate	7.2.19	S6.1: O6.17	□Yes □No
S6.1.24.	Encrypt request to implicit CA certificate	7.2.19	S6.1: O6.17	□Yes □No
S6.2.	Decrypt certificate response	7.2.25	S6.1: M	□Yes □No
S6.3.	Parse certificate response	7.8.10	S6.1: M	□Yes □No
S6.3.1.	Accept response with certificate permissions not identical to permissions in corresponding request	5.6.2.2	S6.3: M	□Yes □No

Item	Security configuration (top-level)	Reference	Status	Support
S6.3.2.	Reject response if any CRL is not valid	5.6.4.2	S6.3: M	
S6.3.3.	Store CRLs that were included in response	7.8.10	S6.3: M	
S6.3.4.	Response verification fails if trust anchor is not explicit certificate	5.5.2.1, 7.8.10	S6.3: M	□Yes □No
S6.3.5.	Response verification fails if explicit certificate in chain is issued by implicit certificate	5.5.2.1, 7.8.10	S6.3: M	□Yes □No
S6.3.6.	Response verification fails if the certificate chain is inconsistent	5.5.3.3, 7.8.10	S6.3: M	□Yes □No
S6.3.7.	Response verification fails if explicit certificates in chain do not verify with the issuing CA's public key	5.5.3.3, 7.8.10	S6.3: M	□Yes □No
S6.3.8.	Reject response if certificate does not match private key	7.2.9	S6.3: M	□Yes □No
S6.3.9.	Accept certificate that includes start validity	5.6.2.2, 6.3.3	O	□Yes □No
S6.3.10.	Accept certificate that includes lifetime as duration	5.6.2.2, 6.3.3	O	□Yes □No
S6.3.11.	Accept certificate with ECDSA-224 verification public key	5.6.2.2, 6.3.3	S6.1: O6.18	□Yes □No
S6.3.12.	Accept certificate with ECDSA-256 verification public key	5.6.2.2, 6.3.3	S6.1: O6.18 S6.1.11:M	□Yes □No
S6.3.13.	Accept certificate including encryption key	5.6.2.2, 6.3.3	S6.1.21:M	□Yes □No
S6.3.13.1.	Encryption public key is ECIES-256	5.6.2.2, 6.3.3	S6.1.21: M	□Yes □No
S6.3.14.	Support receiving uncompressed points	5.6.2.2, 6.3.3	S6.3: O6.19	□Yes □No
S6.3.15.	Support receiving compressed points	5.6.2.2, 6.3.3	S6.3: O6.19	□Yes □No
S6.3.16.	Support fast verification with compressed fast verification information	5.6.2.2, 6.3.3	O	□Yes □No
S6.3.17.	Support fast verification with uncompressed fast verification information	5.6.2.2, 6.3.3	O	□Yes □No
S6.3.17.1.	Maximum certificate chain length supported on received certificate	5.6.2.2, 6.3.3	S6.3: 1:M >1:O	Enter number: ( )
S6.3.17.2.	Accept certificate with circular GeographicRegion	5.6.2.2, 6.3.3	S6.1.13:M	□Yes □No
S6.3.17.3.	Accept certificate with rectangular GeographicRegion	5.6.2.2, 6.3.3	S6.1.14:M	□Yes □No
S6.3.17.3.1.	Maximum number of RectangularRegions supported in a received certificate	6.3.14, 7.5.1	S6.1.14: 6:M >6:O	Enter number: ( )
S6.3.17.4.	Accept certificate with polygonal GeographicRegion	5.6.2.2, 6.3.3	S6.1.15:M	□Yes □No
S6.3.17.4.1.	Maximum number of PolygonalRegion vertices supported in a received certificate.	6.3.18, 7.5.1	S6.1.15: 12:M >12:O	Enter number: ( )

Item	Security configuration (top-level)	Reference	Status	Support
S6.3.17.5.	Accept certificates that use a GeographicRegion of type from_issuer	5.6.2.2, 6.3.3	O	□Yes □No
S6.3.18.	Maximum number of entries in `permissions_list` field.	7.5.1	S6.3: 8:M >8:O	Enter number: ( )
S6.3.19.	Accept certificates with permissions of type from_issuer.	7.5.1	S6.3:M	□Yes □No
S6.4.	Receive CRL over WSMP	5.6.4.1	O	
S6.5.	Verify CRL	5.6.4.2	O	
S6.5.1.	Reject CRL if chain cannot be constructed	5.6.4.2	S6.5:M	
S6.5.2.	Reject CRL if chain is not consistent	5.6.4.2	S6.5:M	
S6.5.3.	Reject CRL if certificate is not consistent with CRL	5.6.4.2	S6.5:M	
S6.5.4.	Reject CRL if certificate is not consistent with CRL	5.6.4.2	S6.5:M	
S6.5.5.	Reject CRL if certificate or CRL signature cannot be verified	5.6.4.2	S6.5:M	

Annex B

(informative)

IEEE 1609.2 security profiles

B.1 General

The IEEE 1609.2 security profile for a calling entity is a compact description of the security processing that that entity carries out. It is intended for use by organizations that produce specifications of applications that use Wireless Access in Vehicular Environments (WAVE) Security Services to provide a standard template for specifying the security processing of a given application. A security profile is linked to one or more PSIDs and specifies the security behavior of applications or services using that PSID.

The IEEE 1609.2 security profile specifies the WAVE Security Services primitives that a calling entity invokes and gives instructions for setting the values of the parameters of those primitives. The IEEE 1609.2 security profile may set these parameters to specific values or it may describe parameters as "variable," in which case it is helpful for profile specifiers to describe how that parameter value is set when the primitive is invoked.

The IEEE 1609.2 security profile contains four sections:

— IEEE 1609.2 security profile identification: describes the IEEE 1609.2 security profile

— Sending: describes options to be set when creating secured data for sending

— Receiving: describes options to be set when processing received secured data

— Security Management: describes constraints on the communications certificates to be used

NOTE 1—In the WAVE ecosystem, the assumption is that a single organization or entity has the right to specify application behavior. That organization is also responsible for specifying application-specific security behavior and as such for specifying the IEEE 1609.2 security profile for that application, if an IEEE 1609.2 security profile is being used.

NOTE 2—An organization that specifies a calling entity may wish to provide minimum requirements for performance for an implementation of that entity. For example, there may be a required accuracy for the estimate of the standard deviation in the time. These minimum performance requirements are in addition to any requirements specified within the security profile, but are not part of the security profile itself.

B.2 Secure data exchange

B.2.1 IEEE 1609.2 security profile identification

Name	Type	Valid range	Description
Name	Text string	Any valid string	The name to be used to refer to the profile.
PSIDs	List of PSIDs	Any list of one or more PSIDs	The PSIDs to be used by secure communications entities that use this profile.

B.2.2 Sending

This part of the IEEE 1609.2 security profile contains the following information.

Name	Type	Valid range	Description
SignData	Enumerated	Yes	The entity signs all outgoing data (with Sec-SignedData.request).
		No	The entity does not sign outgoing data.
		Variable	The entity determines whether to sign outgoing data on a per-sending basis.
SignedContentType	Array of enumerated	Array containing one or more of: Signed, Signed partial payload, Signed external payload	The type of signed data sent. If this array contains more than one entry, the profile should include guidance as to how an application determines which type to use.
SetGenerationTimeIn-SecurityHeaders	Boolean	True, False	The value set as *Set Generation Time* when invoking Sec-SignedData.request.
SetExpiryTimeIn-SecurityHeaders	Boolean	True, False	The value set as *Set Expiry Time* when invoking Sec-SignedData.request.
SetGenerationLocation-InSecurityHeaders	Boolean	True, False	The value set as *Set Generation Location* when invoking Sec-SignedData.request.
TimeBetweenFullCert-ChainTransmissions	Time Interval (for example, “one second”) or enumerated	Any valid interval of time, or “Always”, or “Variable”	Used to set *Signer Identifier Type* when invoking Sec-SignedData.request. If more than this time has elapsed since a sign operation used the full cert chain (Signer Identifier Cert Chain Length = -1) or if this is set to “always”, the Sec-SignedData.request primitive is invoked with *Signer Identifier Type* set to “certificate chain” and *Signer Identifier Cert Chain Length* set to –1. Otherwise, the Sec-SignedData.request primitive is invoked with *Signer Identifier Type* set to the value of the field SignerIdentifier-TypeIfNotFullCertChain and *Signer Identifier Cert Chain Length* set to the value of the field SignerIdentifierCertChainLength.

Name	Type	Valid range	Description
SignerIdentifierType-IfNotFullCertChain	Enumerated	Certificate, Certificate digest, Certificate chain, Variable	The value set as *Signer Identifier Type* when invoking Sec-SignedData.request if the sender is not sending a full certificate chain. If this is "variable", the entity determines on case-by-case basis what value to pass to that parameter. See discussion in E.4.8 of the factors that influence how this might be set.
SignerIdentifierCert-ChainLength	Integer or enumerated	n/a	The appropriate value if *SignerIdentifierTypeIfNotFullCertChain* is not "certificate chain".
		–256 to –1, 1 to 256, or "Max"	The value set as the *Signer Identifier Cert Chain Length* when invoking Sec-SignedData.request when the sender is sending a certificate chain but not a full certificate chain. See discussion in E.4.8 of the factors that influence how this might be set.
		"Variable"	Indicates that when the sender is not sending a certificate chain, but not a full certificate chain, the sender determines on a case-by-case basis what value to set as the *Signer Identifier Cert Chain Length* when invoking Sec-SignedData.request See discussion in E.4.8 of the factors that influence how this might be set.
EncryptData	Enumerated	No, If Possible, Always, Variable	Specifies when encryption is used and what steps are taken if no key is available for the recipient. "If possible" indicates that the sender encrypts messages if a key is available but sends messages unencrypted if a key is not available. "Always" indicates that the sender encrypts messages if a key is available and will not send messages if a key is not available.
SignWithFastVerification	Enumerated	Uncompressed, Compressed, No, Variable	The value set as *Sign With Fast Verification* when invoking Sec-SignedData.request.
ECPointFormat	Enumerated	Uncompressed, Compressed, Variable.	The value set as the *EC Point Format* when invoking Sec-SignedData.request.

B.2.3 Receiving

This part of the IEEE 1609.2 security profile contains the following information.

Name	Type	Valid range	Description
VerifyData	Enumerated	"True", "False", or "Variable"	Specifies whether or not a receiving entity attempts to verify data. If *SignData* in the sending profile is "False", this is set to False. If *SignData* in the sending profile is "True", this is set to True to denote that the receiving entity attempts to verify all incoming data, or Variable to denote that the receiving entity might process the data using methods not specified in this standard before determining whether or not to accept it.
CheckValidityBasedOnGeneration Time	Boolean	"True" or "False"	The value set as *Check Validity Based on Generation Time* to Sec-SignedData-Verification.request.
GenerationTimeSource	Enumerated	"Security Headers" or "Payload"	This need only be set if *Check Validity Based on Generation Time* is true. If *GenerationTimeSource* is "Security Headers", the Generation Time and Generation Time Standard Deviation parameters to Sec-SignedDataVerification.request are obtained from the corresponding fields in Sec-SecureDataContentExtraction.confirm. If *GenerationTimeSource* is "Payload", the Generation Time and Generation Time Standard Deviation are obtained from elsewhere (for example, from the payload) and are provided by the entity to Sec-SignedData-Verification.request.
CheckValidityBasedOnExpiryTime	Boolean	"True" or "False"	The value set as *Check Validity Based on Expiry Time* when invoking Sec-SignedData-Verification.request.
ExpiryTimeSource	Enumerated		This need only be set if *Check Validity Based on Expiry Time* is "True."
		"Security Headers"	The *Expiry Time* parameter to Sec-Signed-DataVerification.request is obtained from the corresponding fields in Sec-SecureDataContentExtraction.confirm.
		"Payload"	The *Expiry Time* is obtained from elsewhere (for example, from the payload) and is provided by the entity to Sec-SignedData-Verification.request.
CheckValidityBasedOnGeneration Location	Boolean	"True" or "False"	The value set as *Check Validity Based on Generation Location* when invoking Sec-SignedDataVerification.request.
GenerationLocationSource	Enumerated		This need only be set if *Check Validity Based on Generation Location* is "True."

Name	Type	Valid range	Description
		"Security Headers"	The *Generation Location* parameter to Sec-SignedDataVerification.request is obtained from the corresponding fields in Sec-SecureDataContentExtraction.confirm.
		"Payload"	The *Generation Location* is obtained from elsewhere (for example, from the payload) and is provided by the entity to Sec-SignedData-Verification.request.
AcceptEncryptedData	Enumerated	"Exclusively"	The entity rejects any received data that are not encrypted.
		"No"	The entity rejects any received data that are encrypted.
		"Variable"	Depending on conditions to be specified by the organization that specifies the IEEE 1609.2 security profile, the entity may accept non-encrypted data or encrypted data.
DetectReplay	Boolean	"True" or "False"	The value set as *Detect Replay in Security Services* when invoking Sec-SignedData-Verification.request.
DataValidityPeriod	Time period or "Variable"	Any positive time period or "Variable"	The value set as *Validity Period* when invoking Sec-SignedDataVerification.request. Set only if *Check Validity Based on Generation Time* is "True".
DataValidityDistance	Distance in meters or "Variable"	Any positive distance or "variable"	The value set as *Validity Distance* when invoking Sec-SignedDataVerification.request. Set only if *Check Validity Based on Generation Location* is "True".
RejectionThresholdFor-TooOldData	Real-valued number	Between 0 and 1 or "Variable"	The value set as *Rejection Threshold For Too Old Data* when invoking Sec-SignedData-Verification.request, or the algorithm for setting that value.
AcceptableFutureDataPeriod	Time	Any positive time value	The value set as *Acceptable Future Data Period* when invoking Sec-SignedData-Verification.request, or the algorithm for setting that value.
RejectionThresholdForFutureData	Real-valued number	Between 0 and 1	The value set as *Rejection Threshold For Future Data* when invoking Sec-SignedData-Verification.request, or the algorithm for setting that value.
RejectionThresholdFor-ExpiredData	Real-valued number	Between 0 and 1	The value set as *Rejection Threshold For Expired Data* when invoking Sec-SignedData-Verification.request, or the algorithm for setting that value.
Maximum Certificate Chain Length	Integer	Integer ≥ 2	The value set as *Maximum Certificate Chain Length* when invoking Sec-SignedData.request and Sec-SignedDataVerification.request.
OverdueCRLTolerance	Time period or "Variable"	Any positive time period or "Variable"	The value set as *Overdue CRL Tolerance* when invoking Sec-SignedData-Verification.request.

B.2.4 Security management

This part of the IEEE 1609.2 security profile contains the following information for each PSID for which the entity uses a 1609Dot2Data structure:

Name	Type	Valid range	Description
SigningKeyAlgorithm	Enumerated	ECDSA-224 ECDSA-256	One of the valid signing algorithms identified in 6.2.16, 6.2.17.
EncryptionAlgorithm	Enumerated	ECIES-256, none	One of the valid encryption algorithms identified in 6.2.17, 6.2.24, 6.2.25.
PublicKeyTransferType	Enumerated	Explicit	A receiver supports receiving explicit certificates only and a sender uses an explicit certificate only for a given transmission.
		Implicit	A receiver supports receiving implicit certificates only and a sender uses an implicit certificate only for a given transmission.
		Both	A receiver supports receiving both implicit and explicit certificates and a sender may choose to use either an implicit or an explicit certificate for a given transmission if it has certs of both types available.
ECPointFormat	Enumerated	Compressed, Uncompressed	How points are to be represented in certificates.
SupportedGeographicRegions	Array of enumerated	An array of entries, each of which is one of the following: None, Rectangular, Circular, Polygonal	The type of geographic region supported for conformant certificates.
CertificateStartValidity	Enumerated	Require start validity, Allow start validity, No start validity	Whether a receivers supports a start validity in the received certificate.
LifetimeIsDuration	Enumerated	Duration only, No duration, Both	Whether a receiver supports lifetime as duration in the received certificate.

NOTE—If profiles for two different PSIDs have different values in the Security Management section, it may not be possible to issue a certificate that contains both PSIDs, because the requirements placed on the certificate by one profile cannot be satisfied consistently with the requirements placed on the certificate by the other profile.

B.3 IEEE 1609.2 security profile proforma[29]

B.3.1 Instructions for completing the IEEE 1609.2 security profile proforma

The developer of an IEEE 1609.2 security profile may specify the profile by completing this proforma. The main part of the PICS proforma is a fixed questionnaire, divided into entries. Answers to the questionnaire

[29] *Copyright release for 1609.2 security profile proformas:* Users of this standard may freely reproduce the 1609.2 security profile proforma in this annex so that it can be used for its intended purpose and may further publish the completed 1609.2 security profile.

items are to be provided in the center column, and any elaboration necessary is to be provided in the rightmost column. The entries in the value column shall either be drawn from the list of permitted values given in Annex B, or be "n/a".

B.3.2 IEEE 1609.2 security profile proforma

B.3.2.1 IEEE 1609.2 security profile identification

Field	Value	Notes
Name		
PSIDs		

B.3.2.2 Sending

Field	Value	Notes
SignData		
SignedContentType		
SetGenerationTimeInSecurityHeaders		
SetExpiryTimeInSecurityHeaders		
SetGenerationLocationInSecurityHeaders		
TimeBetweenFullCertChainTransmissions		
SignerIdentifierTypeIfNotFullCertChain		
SignerIdentifierCertChainLength		
EncryptData		
SignWithFastVerification		
ECPointFormat		

B.3.2.3 Receiving

Field	Value	Notes
VerifyData		
CheckValidityBasedOnGenerationTime		
GenerationTimeSource		
CheckValidityBasedOnExpiryTime		
ExpiryTimeSource		
CheckValidityBasedOnGenerationLocation		
GenerationLocationSource		
AcceptEncryptedData		
DetectReplay		
DataValidityPeriod		
DataValidityDistance		
RejectionThresholdForTooOldData		
AcceptableFutureDataPeriod		
RejectionThresholdForFutureData		
RejectionThresholdForExpiredData		
Maximum Certificate Chain Length		
OverdueCRLTolerance		

B.3.2.4 Security management

Field	Value	Notes
SigningKeyAlgorithm		
EncryptionAlgorithm		
PublicKeyTransferType		
ECPointFormat		
SupportedGeographicRegions		
CertificateStartValidity		
LifeimeIsDuration		

Annex C

(normative)

IEEE 1609.2 security profile for specific use cases

C.1 SAE J2735 Basic Safety Message

C.1.1 General

SAE J2735 Basic Safety Message (BSM) [B20] shall conform to the IEEE 1609.2 security profile described in this clause, unless SAE, subsequent to the publication of this standard, publishes a superseding IEEE 1609 security profile for the BSM.

C.1.2 IEEE 1609.2 security profile identification

Field	Value	Notes
Name	"IEEE 1609.2 security profile for SAE J2735 BSM"	
PSIDs	0x20, 0x21, 0x22	

C.1.3 Sending

Field	Value	Notes
SignData	True	
SetGenerationTimeInSecurityHeaders	True	The BSM does not contain sufficient time information to prevent replay attacks.
SetExpiryTimeInSecurityHeaders	False	BSM does not use expiry time.
SetGenerationLocationInSecurityHeaders	False	The BSM message itself contains the generation location.
TimeBetweenFullCertChainTransmissions	Variable	Guidance will be provided at a later date by SAE.
SignerIdentifierTypeIfNotFullCertChain	Certificate digest	
SignerIdentifierCertChainLength	–1	BSM certificate chains should be limited to being of length 1.
EncryptData	No	
SignWithFastVerification	Compressed	
ECPointFormat	Compressed	

C.1.4 Receiving

Field	Value	Notes
VerifyData	Variable	A BSM-consuming entity may choose not to incur the cost of verifying data if those data are of minor relevance to the receiver.
CheckValidityBasedOnGenerationTime	True	
GenerationTimeSource	Security Headers	
CheckValidityBasedOnExpiryTime	False	Generation time is enough for BSM entities to judge relevance.
ExpiryTimeSource	n/a	Expiry time is not used.
CheckValidityBasedOnGenerationLocation	Variable	In general, generation location is in message and location relevance check is carried out by receiving entity. However the receiving entity may choose to have the check carried out by the security services.
GenerationLocationSource	External	Obtained from message.
AcceptEncryptedData	False	
DetectReplay	True	
DataValidityPeriod	Variable	Default is 5s.
DataValidityDistance	n/a	BSM security does not use generation location – generation location is carried in the payload and processed by the entity.
RejectionThresholdForTooOldData	Variable	Default is 0.5.
AcceptableFutureDataPeriod	Variable	Default is 0.1 s.
RejectionThresholdForFutureData	Variable	Default is 0.5.
RejectionThresholdForExpiredData	Variable	Default is 0.5.
Maximum Certificate Chain Length	8	The full certificate chain for BSMs should be as short as possible, ideally of length 2 or 3.
OverdueCrlTolerance	Variable	Default is: CRL freshness checking not required.

C.1.5 Security management

Field	Value	Notes
SigningKeyAlgorithm	ECDSA-256	
EncryptionAlgorithm	none	BSM does not use encryption.
PublicKeyTransferType	Both	BSM implementations shall support receiving implicit certificates.
ECPointFormat	Compressed	
SupportedGeographicRegions	None, Rectangular, Circular, Polygonal	
CertificateStartValidity	Require start validity	
LifetimeIsDuration	Both	

C.2 WSA

C.2.1 IEEE 1609.2 security profile identification

This profile does not have a standard profile as WSAs are not indicated by use of PSIDs.

C.2.2 Sending

Field	Value	Notes
SignData	True	
SetGenerationTimeInSecurityHeaders	True	
SetExpiryTimeInSecurityHeaders	True	
SetGenerationLocationInSecurityHeaders	True	
TimeBetweenFullCertChainTransmissions	Always	
SignerIdentifierType	Certificate Chain	Send certificate with every transmission.
SignerIdentifierCertChainLength	–1	
EncryptData	No	
SignWithFastVerification	No	
ECPointFormat	Uncompressed	

C.2.3 Receiving

Field	Value	Notes
VerifyData	Variable	Verify a WSA the first time it is received; do not verify duplicates.
CheckValidityBasedOnGenerationTime	True	
GenerationTimeSource	Security Headers	
CheckValidityBasedOnExpiryTime	True	
ExpiryTimeSource	Security Headers	
CheckValidityBasedOnGenerationLocation	True	
GenerationLocationSource	Security Headers	
AcceptEncryptedData	False	
DetectReplay	False	WSAs are meant to be replayed.
DataValidityPeriod	60 s	Configurable.
DataValidityDistance	500 m	Configurable.
RejectionThresholdForTooOldData	n/a	Decisions about relevance of data in the WSA are not made by the WME or security services but by user applications.
AcceptableFutureDataPeriod	n/a	Decisions about relevance of data in the WSA are not made by the WME or security services but by user applications.

Field	Value	Notes
RejectionThresholdForFutureData	n/a	Decisions about relevance of data in the WSA are not made by the WME or security services but by user applications.
RejectionThresholdForExpiredData	n/a	Decisions about relevance of data in the WSA are not made by the WME or security services but by user applications.
Maximum Certificate Chain Length	8	
OverdueCrlTolerance	n/a	WME writes overdue CRL information to UserAvailableServiceTable entries rather than making a decision about CRL overdueness.

C.2.4 Security management

Field	Value	Notes
SigningKeyAlgorithm	ECDSA-256	
EncryptionAlgorithm	n/a	
PublicKeyTransferType	Explicit	
ECPointFormat	Uncompressed	
SupportedGeographicRegions	None, Rectangular, Circular, Polygonal	
CertificateStartValidity	Require start validity	
LifetimeIsDuration	Duration only	

Annex D

(informative)

Example and Use Cases

D.1 Examples of encoded data structures

D.1.1 Signed data with signer type = certificate

This clause presents an encoded signed data structure, first as a flat encoded octet string, second as a parsed IEEE 1609.2 data structure. In the parsed data structure, the contents are presented in the form:

field_name (*offset*, *length*): *value*

Encoded:

```
02 01 03 02 02 04 f3 db 4f 6f ca b6 49 65 01 09
63 65 72 74 4e 61 6d 65 31 01 05 e0 00 00 01 00
04 00 00 00 00 00 00 00 01 00 02 d4 a8 61 1d ce
d8 8c a7 a2 e9 6a 8d 7e 49 0f 3c 9a 46 27 c0 72
26 ed 67 8d 04 74 41 02 00 03 9c b6 6f 87 4a 40
7c 21 83 40 22 db 6d 0a 80 d0 14 cb df 24 fc a0
83 f8 e2 00 81 b0 7c 14 b8 e7 02 19 90 d0 57 4b
14 d2 80 29 1f c4 e6 a6 73 12 68 74 96 77 c2 52
34 ae bb e4 29 da 16 60 61 19 74 c6 b3 53 98 0e
70 e3 3d 4f b9 03 99 76 05 44 e9 74 70 d9 92 bb
3c 37 92 c3 51 d4 7d 8e ea b1 03 0a e0 00 00 01
0c 73 6f 6d 65 20 63 6f 6e 74 65 6e 74 00 00 e7
2a dc 3e dc 09 00 00 00 00 00 00 00 00 00 00 00
02 ca bf a2 0d 82 ae 3e 25 a3 8c 9c dd 2e cf 94
9f cc 7c 7f d9 d8 83 89 f5 08 f7 aa bb 5b ef 21
bd 7a 2e 79 6c c7 de 01 af b1 93 35 5b e2 f5 88
19 76 70 e4 ae 09 cf 3b ee
```

Parsed:

```
protocol_version (0, 1): 02
type (1, 1): 01 (signed)
signed_data (2, 263):
  signer (2, 169):
    type (2, 1): 03 (certificate)
    certificates (3, 168):
      version_and_type (3, 1): 02 (explicit)
      unsigned_certificate (4, 102):
        holder_type (4, 1): 02 (identified localized)
        cf (5, 1): 04 (encryption_key)
        signer_id (6, 8): f3 db 4f 6f ca b6 49 65
        signature_alg (14, 1): 01 (ECDSA NIST P256)
        scope (15, 18):
          id_scope (15, 18):
            name_len (15, 1): 09
```

```
            name (16, 9): 63 65 72 74 4e 61 6d 65 31
            permissions (25, 7):
              type (25, 1): 01 (specified)
              permissions_list_len (26, 1): 05
              permissions_list (27, 5):
                psid (27, 4): e0 00 00 01
                service_specific_permissions_len (31, 1): 00
            region (32, 1):
              region_type (32, 1): 04 (none)
          expiration (33, 4): 00 00 00 00 (00:00:34 01 Jan 2004 UTC)
          crl_series (37, 4): 00 00 00 01
          verification_key (41, 30):
            algorithm (41, 1): 00 (ECDSA NIST P224)
            public_key (42, 29):
              type (42, 1): 02 (compressed, lsb of y is 0)
              x (43, 28):
                d4 a8 61 1d ce d8 8c a7  a2 e9 6a 8d 7e 49 0f 3c
                9a 46 27 c0 72 26 ed 67  8d 04 74 41
          encryption_key (71, 35):
            algorithm (71, 1): 02 (ECIES NIST P256)
            supported_symm_alg (72, 1): 00 (AES 128 CCM)
            public_key (73, 33):
              type (73, 1): 03 (compressed, lsb of y is 1)
              x (74, 32):
                9c b6 6f 87 4a 40 7c 21  83 40 22 db 6d 0a 80 d0
                14 cb df 24 fc a0 83 f8  e2 00 81 b0 7c 14 b8 e7
        signature (106, 65):
          ecdsa_signature (106, 65):
            R (106, 33):
              type (106, 1): 02 (compressed, lsb of y is 0)
              x (107, 32):
                19 90 d0 57 4b 14 d2 80  29 1f c4 e6 a6 73 12 68
                74 96 77 c2 52 34 ae bb  e4 29 da 16 60 61 19 74
            s (139, 32):
              c6 b3 53 98 0e 70 e3 3d  4f b9 03 99 76 05 44 e9
              74 70 d9 92 bb 3c 37 92  c3 51 d4 7d 8e ea b1 03
    unsigned_data (171, 37):
      tf (171, 1): 0a (use_generation_time, use_location)
      psid (172, 4): e0 00 00 01
      data_len (176, 1): 0c
      data (177, 12): 73 6f 6d 65 20 63 6f 6e 74 65 6e 74
      generation_time (189, 9):
        time (189, 8): 00 00 e7 2a dc 3e dc 09
            (19:08:23 20 Jan 2012 UTC)
        log_std_dev (197, 1): 00 (1.134666 ns or less)
      generation_location (198, 10):
        latitude (198, 4): 00 00 00 00
        longitude (202, 4): 00 00 00 00
        elevation (206, 2): 00 00
    signature (208, 57):
      ecdsa_signature (208, 57):
        R (208, 29):
          type (208, 1): 02 (compressed, lsb of y is 0)
          x (209, 28):
            ca bf a2 0d 82 ae 3e 25  a3 8c 9c dd 2e cf 94 9f
            cc 7c 7f d9 d8 83 89 f5  08 f7 aa bb
        s (237, 28):
```

```
            5b ef 21 bd 7a 2e 79 6c  c7 de 01 af b1 93 35 5b
            e2 f5 88 19 76 70 e4 ae  09 cf 3b ee
```

D.1.2 Signed WSA

This clause presents an encoded signed WSA, first as a flat encoded octet string, second as a parsed IEEE 1609.2 data structure. In the parsed data structure, the contents are presented in the form:

field_name (*offset*, *length*): *value*

```
02 0b 04 53 03 04 01 42  07 67 16 e8 a5 6d 3c 01
00 01 16 20 1f 00 23 1f  00 80 03 1f 00 bf e0 1f
00 bf e1 1f 00 bf f0 1f  00 04 10 97 c0 de 0f 0c
3e de 00 00 00 01 02 99  73 a4 82 47 e3 a9 10 f9
c0 c4 1d 67 3f cb 96 45  af 28 4c cf 1d cf 7c 5a
c7 c6 cd 88 8e 78 5a 02  02 05 0e 80 9a 04 04 01
14 06 0f 11 8d e1 0c c5  38 5a 62 08 fc 66 ff ff
ff ff 07 05 55 53 44 4f  54 01 23 1f 01 08 04 53
43 4d 53 09 10 20 01 18  90 11 0e a7 77 00 00 00
00 00 00 00 03 0a 02 3e  dc 01 bf e1 1f 01 08 04
53 43 4d 53 09 10 20 01  18 90 11 0e a7 77 00 00
00 00 00 00 00 03 0a 02  3e dc 02 11 b6 00 0c 14
15 01 01 03 07 08 20 01  04 70 e0 fb 99 99 00 00
00 00 00 00 00 00 40 20  01 04 70 e0 fb 99 99 00
00 00 00 00 00 00 01 20  01 04 70 e0 fb 99 99 00
00 00 00 00 00 00 01 00  00 ec 98 28 b6 d0 c8 00
00 00 ec 98 2a 80 94 64  00 00 00 00 00 00 00 00
00 00 03 13 6b 3f 4c 96  00 fb 42 11 99 ab 9a 43
a4 42 d5 f6 92 bb a3 4c  5b 4b 9d 5d 3d fb 5b 81
b0 f1 cf 29 62 ad d2 70  77 0d 24 a3 87 68 49 03
e5 71 2c 29 67 b5 f6 d9  1a 94 5b a9 df 1c ec 0d
b5 d9 68

Dot2Data (0, 339):
  protocol_version (0, 1): 02
  type (1, 1): 0b (signed WSA)
  signed_wsa (2, 337):
    signer (2, 85):
      type (2, 1): 04 (certificate chain)
      cert_chain_len (3, 1): 53
      certificates (4, 83):
        version_and_type (4, 1): 03 (implicit)
        unsigned_certificate (5, 49):
          holder_type (5, 1): 04 (wsa)
          cf (6, 1): 01 (use_start_validity)
          signer_id (7, 8): 42 07 67 16 e8 a5 6d 3c
          signature_alg (15, 1): 01 (ECDSA NIST P256)
          scope (16, 26):
            wsa_scope (16, 26):
              name_len (16, 1): 00
              permissions (17, 24):
                type (17, 1): 01 (specified)
                permissions_list_len (18, 1): 16
                permissions_list (19, 3):
```

```
                psid (19, 1): 20
                max_priority (20, 1): 1f
                service_specific_permissions_len (21, 1): 00
              permissions_list (22, 3):
                psid (22, 1): 23
                max_priority (23, 1): 1f
                service_specific_permissions_len (24, 1): 00
              permissions_list (25, 4):
                psid (25, 2): 80 03
                max_priority (27, 1): 1f
                service_specific_permissions_len (28, 1): 00
              permissions_list (29, 4):
                psid (29, 2): bf e0
                max_priority (31, 1): 1f
                service_specific_permissions_len (32, 1): 00
              permissions_list (33, 4):
                psid (33, 2): bf e1
                max_priority (35, 1): 1f
                service_specific_permissions_len (36, 1): 00
              permissions_list (37, 4):
                psid (37, 2): bf f0
                max_priority (39, 1): 1f
                service_specific_permissions_len (40, 1): 00
            region (41, 1):
              region_type (41, 1): 04 (none)
        expiration (42, 4): 10 97 c0 de (00:00:00 27 Oct 2012 UTC)
        start_validity (46, 4): 0f 0c 3e de (00:00:00 01 Jan 2012
UTC)
        crl_series (50, 4): 00 00 00 01
      reconstruction_value (54, 33):
        type (54, 1): 02 (compressed, lsb of y is 0)
        x (55, 32):
          99 73 a4 82 47 e3 a9 10  f9 c0 c4 1d 67 3f cb 96
          45 af 28 4c cf 1d cf 7c  5a c7 c6 cd 88 8e 78 5a
  unsigned_data (87, 187):
    permission_indices_len (87, 1): 02
    permission_indices (88, 2): 02 05
    tf (90, 1): 0e (use_generation_time, expires, use_location)
    data_len (91, 2): 80 9a
    data (93, 154):
      04 04 01 14 06 0f 11 8d  e1 0c c5 38 5a 62 08 fc
      66 ff ff ff ff 07 05 55  53 44 4f 54 01 23 1f 01
      08 04 53 43 4d 53 09 10  20 01 18 90 11 0e a7 77
      00 00 00 00 00 00 00 03  0a 02 3e dc 01 bf e1 1f
      01 08 04 53 43 4d 53 09  10 20 01 18 90 11 0e a7
      77 00 00 00 00 00 00 00  03 0a 02 3e dc 02 11 b6
      00 0c 14 15 01 01 03 07  08 20 01 04 70 e0 fb 99
      99 00 00 00 00 00 00 00  00 40 20 01 04 70 e0 fb
      99 99 00 00 00 00 00 00  00 01 20 01 04 70 e0 fb
      99 99 00 00 00 00 00 00  00 01
    generation_time (247, 9):
      time (247, 8): 00 00 ec 98 28 b6 d0 c8 (20:38:16 29 Mar 2012
UTC)
      log_std_dev (197, 1): 00 (1.134666 ns or less)
    expiry_time (256, 8): 00 00 ec 98 2a 80 94 64 (20:38:46 29 Mar
2012 UTC)
    generation_location (264, 10):
```

```
      latitude (264, 4): 00 00 00 00
      longitude (268, 4): 00 00 00 00
      elevation (272, 2): 00 00
  signature (274, 65):
    ecdsa_signature (274, 65):
      R (274, 33):
        type (274, 1): 03 (compressed, lsb of y is 1)
        x (275, 32):
          13 6b 3f 4c 96 00 fb 42  11 99 ab 9a 43 a4 42 d5
          f6 92 bb a3 4c 5b 4b 9d  5d 3d fb 5b 81 b0 f1 cf
      s (307, 32):
        29 62 ad d2 70 77 0d 24  a3 87 68 49 03 e5 71 2c
        29 67 b5 f6 d9 1a 94 5b  a9 df 1c ec 0d b5 d9 68
```

D.2 Secure data reception

Figure D.1illustrates a sample process flow showing use of the security processing services by a secure data exchange entity (SDEE) on receipt of secure data.

First, the SDEE invokes Sec-SecureDataContentExtraction.request. This causes the security processing services to:

- Decrypt the data if necessary.
- If the data was not signed, extract the data payload from the decrypted message.
- If the data was signed:
- Extract the SignerIdentifier from the SignedData, and the information fields from the ToBeSignedData.
- Store new certificate information with the CME via CME-AddCertificate.request (7.5.5). Any certificates stored in this case are marked as “not verified”.

The security services return the result of the request, including the SignerIdentifier from the SignedData, and the information fields from the ToBeSignedData, via Sec-SecureDataContentExtraction.confirm.

If the message was not signed (i.e., it was encrypted only, or not secured at all), the security processing is complete and the SDEE may proceed to carrying out application processing on the data and other information fields.

If the message was signed, the SDEE determines whether or not to request verification of the message.

- For some SDEEs, all messages will be verified; this is specified in the IEEE 1609.2 security profile for that SDEE.
- If the SDEE verifies only certain messages, the SDEE determines whether or not to verify by applying context-specific relevance tests to the data returned by Sec-SecureDataContentExtraction-.confirm. These tests are SDEE-specific and out of scope of this standard. They may include, for example, a test that the message is sufficiently fresh, that no more recent messages have been received, or that the message would if valid result in an alert being raised to a human. Based on these tests, the SDEE may accept the data without verification, reject the data, or determine that the data requires cryptographic verification before the SDEE acts on it.

— The SDEE may wish to ensure, before verifying, that the signed data payload is consistent with the Service Specific Permissions (SSP) from the signer's certificate. The SDEE can obtain the SSP in the following ways:

— If the SignerIdentifier is of type `certificate` or `certificate_chain`, the SDEE can read the SSP directly from the certificate.

— If the SignerIdentifier is of type `certificate_digest_with_ecdsap224` or `certificate_digest_with_ecdsap256`, the SDEE may attempt to obtain the SSP by invoking CME-CertificateInfo.request. If the digest in the SignerIdentifier corresponds to a certificate that is already known to the CME, the permissions including the SSP will be returned by CME-CertificateInfo.confirm.

Having obtained the SSP, the SDEE compares it with the data payload. The SSP syntax and semantics are specific to a PSID and the consistency test will be different for different PSIDs. The consistency test for the SSP is to be carried out by the SDEE, not by the security processing services defined in this standard. See E.5.1 and E.5.2 for further discussion of SSP.

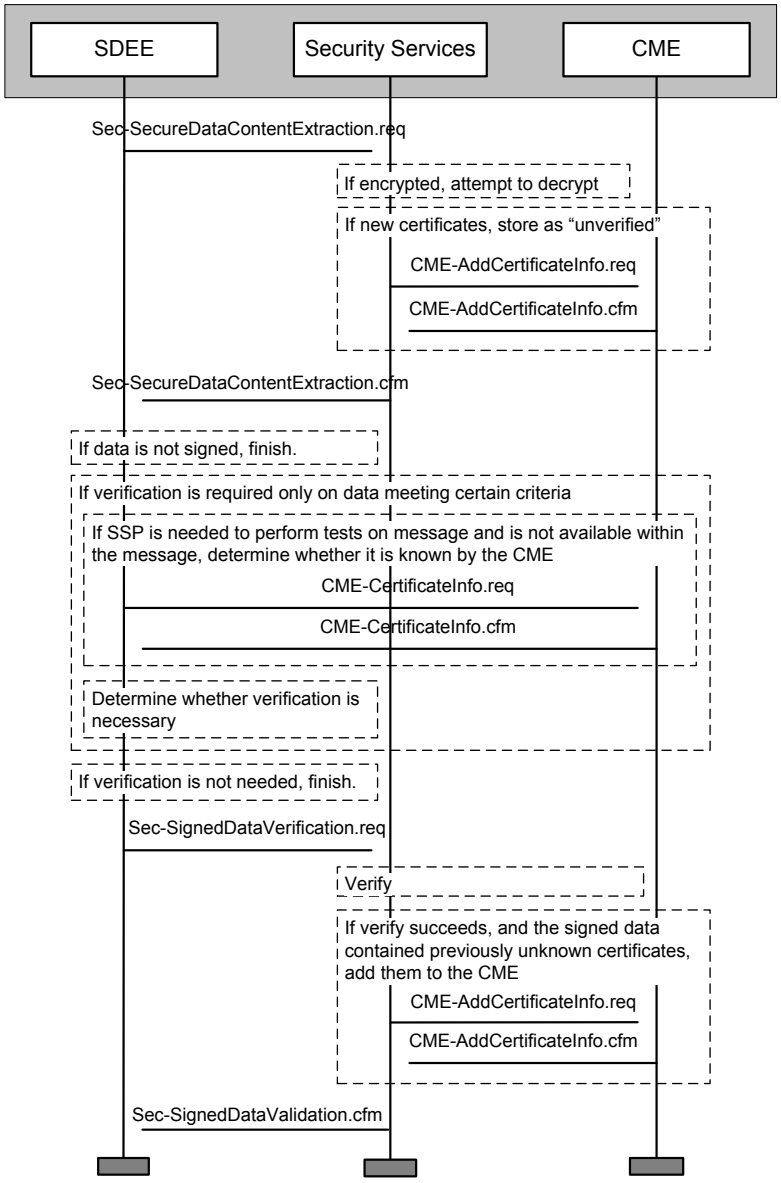


Figure D.1—Process flow for receiving secure data

If the SDEE determines that verification is necessary, it invokes Sec-SignedDataVerification.request to request that the security processing services verify the message. This causes the security processing services to take the following actions:

— Determine whether the signed data is valid using the criteria specified in 5.5.

— If data is valid, store new certificate information with the CME, via CME-AddCertificate.request (7.5.5). New certificates stored in this case may be marked as "verified". If the data is not valid no certificate information is added.

The security processing services return the result of the request via Sec-SignedDataVerification.confirm. The SDEE takes the result of verification into account when determining whether or not to take further action on the basis of the received data. If the signed data was valid, the SDEE may store the received signing certificate and use it to encrypt subsequent communications with the certificate holder.

D.3 Certificate request

D.3.1 Certificate request for secure data exchange

D.3.1.1 Functional entities

A certificate request is a request to a CA to associate a particular signing public key with a particular set of permissions within a certificate.

Figure D.2 illustrates the functional entities involved in requesting certificates for use by any secure data exchange entity. The heavy dashed lines indicate processing defined in this standard and the dot-dashed arrows indicate data flows not defined in this standard. The functional entities have the following responsibilities:

— The SDEE provides the certificate management process with some or all of the information to be included in the certificate request. Following successful completion of the certificate management process, the SDEE possesses a CMH referencing a private key and the associated certificate.

— The certificate management process is responsible for determining the information to be included in a certificate request, determining the key and certificate to be used to sign the certificate request, invoking the security processing services to create the certificate request, sending the request to the CA, invoking the security processing services to process the response received from the CA and making the returned certificates available to SDEE.

— The CA is responsible for processing a received certificate request and returning the response to the certificate management process.

— The security processing services create the certificate request and all associated keys, store the certificate and keys and process the response from the CA when it is received by the certificate management process.

NOTE—The details of the interface between the certificate management process and the SDEE are not defined in this standard.

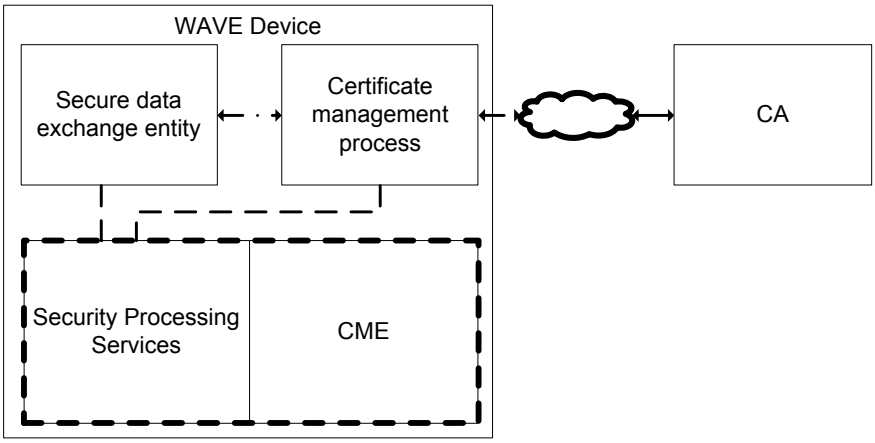


Figure D.2—Functional entities for certificate request for secure data exchange

D.3.1.2 Process Flow

Figure D.3 shows a sample process flow showing the interactions between the certificate management process and the security processing services when requesting certificates for secure data exchange. The dashed lines indicate functionality defined in this standard. Primitive names in the figure are abbreviated for compactness. The full names of the primitives are given in the text of this section.

Obtain Keying Material. The certificate management process obtains two CMHs in *Key Pair Only* state. One CMH (the "signing CMH") references a keypair for a signing algorithm. The public key from this keypair will be the verification key in the certificate request. The other CMH (the "response decryption CMH") references a keypair for an encryption algorithm. The public key from this keypair will be the response encryption key in the certificate request.

Gather Information Elements. The certificate management process obtains the information elements to be included in the certificate request. These information elements are defined in the definition of Sec-CertificateRequest.request. They may be obtained directly from the SDEE, by manual entry, from a configuration file, or from any other trustworthy source. The SDEE may be applying for certificates for multiple PSIDs, in which case this standard supports including the PSIDs in a single certificate or in multiple certificates. If the SDEE is applying for certificates for multiple PSIDs, the certificate management process uses logic outside the scope of this standard to determine which PSIDs are to be in certificates on their own and which are to be combined.

Generate Certificate Request. After obtaining the information elements to be input to Sec-Certificate-Request.request, the certificate management process invokes Sec-CertificateRequest.request and obtains a signed, encrypted certificate request via Sec-CertificateRequest.confirm (assuming all inputs are valid). Sec-CertificateRequest.confirm also returns a 10-byte hash of the plaintext of the certificate request.

Interaction with CA. The certificate management process transmits the certificate request to the CA.

The CA determines whether or not to issue the certificate and returns either the certificate response or an error message. This response is encrypted. See D.3.4 for a discussion of possible processing by the CA.

Process Response. The certificate management process receives the response and determines which decryption key to pass to Sec-CertificateResponseProcessing.request by inspecting the headers of the encrypted response, matching this to the public key that was provided to encrypt the response and matching that to a response decryption CMH. It invokes Sec-CertificateResponseProcessing.request to request that the security processing services decrypt the response and (if appropriate and possible) verify the contents.

If the CA declined to issue the certificate, the response contains an encrypted ToBeEncryptedCertificate-RequestError. The security processing services extract the information from the ToBeEncryptedCertificate-RequestError structure and return an appropriate notification to the certificate management process via Sec-CertificateResponseProcessing.confirm. This includes a 10-byte hash of the request. The hash allows the certificate management process to search the hashes of all outstanding requests as returned by Sec-CertificateRequest.confirm and thereby determine which certificate request was rejected.

If all operations succeeded and the certificate could be verified, the security processing services return the certificate to the certificate management process via Sec-CertificateResponseProcessing.confirm. and update the certificate information stored by the CME using the mechanisms supported by CME-AddCertificateRevocation.request.

As part of response processing, the security services update the CME with information received in the response.

Acknowledgement. If so requested by the CA, the receiving certificate management process may send an acknowledgment. This acknowledgment consists of a ToBeEncryptedCertificateResponseAcknowledgment

generated as described in 6.3.39 and which is then encrypted by invoking Sec-EncryptedData.request. The certificate management process sends the acknowledgement to the CA. No response from the CA is specified.

Store Certificate. On reception of a Sec-CertificateResponseProcessing.confirm indicating success and with a certificate that is acceptable to the implementation, the certificate management process stores the certificate by invoking Sec-CryptomaterialHandle-StoreCertificate.request directly or by passing the certificate back to the SDEE. If a certificate contains both a verification key and an encryption key, the certificate management process stores the certificate at both the CMH associated with the verification key and the CMH associated with the encryption key in the certificate. If the certificate was an implicit certificate, the certificate response also includes information to be used to update the private key. This information is specified in the specification of Sec-CryptomaterialHandle-StoreCertificate.request.

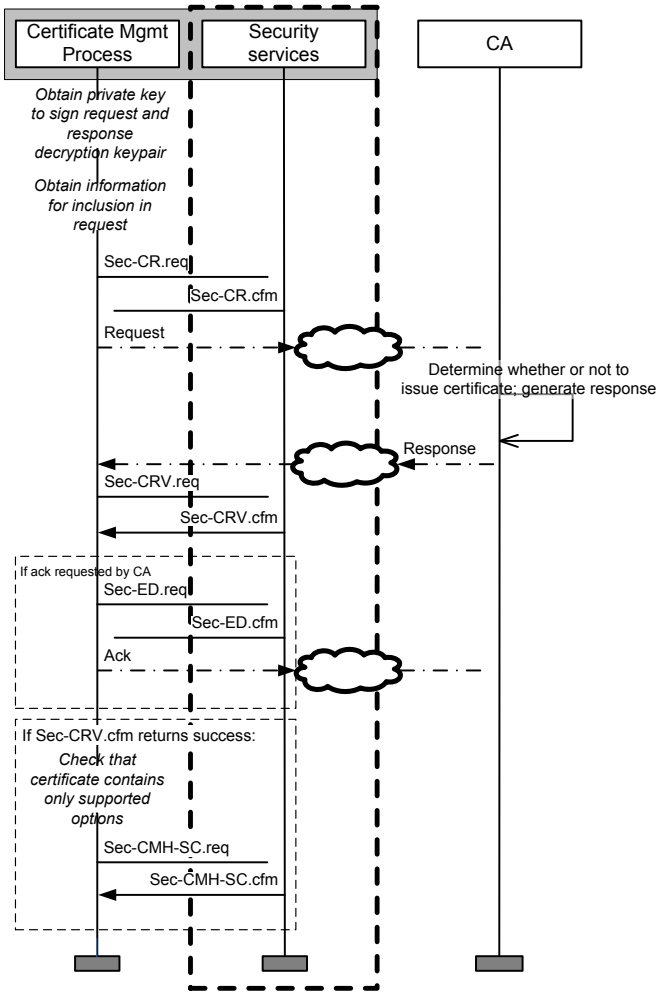


Figure D.3—Process flow for certificate request

NOTE—This standard does not define the logic or processing necessary to determine when to request certificates or which certificates to request.

D.3.2 Certificate request for secure WSAs

D.3.2.1 Functional entities

Figure D.4 illustrates the functional entities involved in requesting WSA certificates. The heavy dashed lines indicate processing defined in this standard, the dot-dashed lines indicate data flows not defined in this standard, and the dotted lines indicate data flows defined in IEEE Std 1609.3.

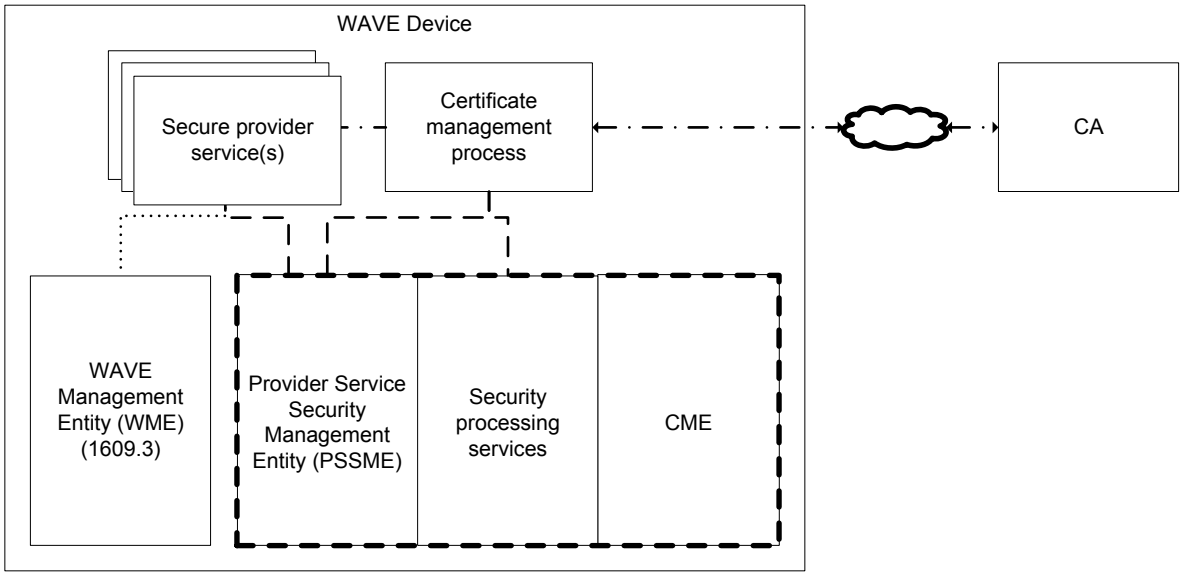


Figure D.4—Functional entities for certificate request for signed WSAs

The functional entities have the following responsibilities:

- The secure provider services provide the PSSME with information that may be included in the certificate request.
- The certificate management process is responsible for determining the information to be included in certificate request, determining the key and certificate to be used to sign the certificate request, invoking the security services to create the certificate request, sending the request to the CA, invoking the security services to process the response received from the CA, and making the returned certificates available to the secure data exchange entity.
- The CA is responsible for processing a received certificate request and returning the response to the certificate management process.
- The PSSME provides the certificate management process with information that it may use in generating the certificate request and stores the WSA certificates and private keys when the certificate management process receives them from the CA.
- The security processing services create the certificate request and all associated keys and process the response from the CA when it is received by the certificate management process.

D.3.2.2 Process Flow

This section describes an overall process flow for requesting certificates for signed WSAs. The section is illustrated by the following figures:

— Figure D.7 shows the process flow from the point of the view of the functional entities on the device that results in a secured ServiceInfo being included in a signed WSA. The figure also shows how the LSI-S is exchanged among entities to ensure that the secure provider service may consistently be identified by the security services and the networking services.

— Figure D.5 expands Figure D.7, step 2, to show the part of the process flow in which secure provider services use the PSSME to pass permission requests to the certificate management process.

Figure D.6 expands Figure D.7, step 4, to show the interactions between the certificate management process, the security processing services, and the CA.

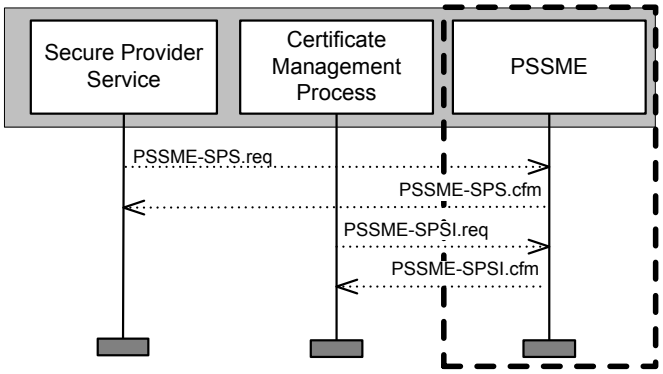


Figure D.5—Providing permission requests to the certificate management process via the PSSME

In each figure, the dashed lines indicate functionality specified in this standard. Primitive names in the figures are abbreviated for compactness.

The steps are as follows. The starting state is a higher layer entity that has not interacted with any of the security service or networking service entities shown in Figure 1 or Figure 2 (and so is not advertised in WSAs, secure or otherwise). At the end of the process, the WME is has the information needed to advertise that higher layer entity as a secure provider service in signed WSAs.

Obtain LSI-S (Figure D.7, step 1**)**. Each secure provider service obtains a Local Service Index for Security PSSME-LocalServiceIndexForSecurity.request. The PSSME returns the requested LSI-S via PSSME-LocalServiceIndexForSecurity.confirm.

Secure provider service registers permission requests (Figure D.5; Figure D.7, step 2 and step 3). The secure provider service provides the certificate management process with information to be used in creating certificate requests. This may be done using mechanisms specified in this standard or otherwise. If using mechanisms defined in this standard, the provider invokes PSSME-SecuredProviderService.request (PSSME-SPS.req in the diagram) and the certificate management process polls the PSSME with PSSME-SecureProviderServiceInfo.request (PSSME-SPSI.req in the diagram) to obtain information about all registered secure provider services (Figure D.7, step 2). The case where the information is registered by other, implementation-specific, mechanisms, is illustrated in Figure D.7, step 3.

NOTE 1—Invocations of PSSME-SecuredProviderService.request and PSSME-SecureProviderServiceInfo.request are asynchronous, as indicated by the use of the stick arrowheads in Figure D.5.

NOTE 2—Implementers may choose to implement a notification system in addition to PSSME-SecuredProviderService.request and PSSME-SecureProviderServiceInfo.request so that the certificate management process is actively informed of new PSSME-SecuredProviderService.request events rather than needing to repeatedly poll the PSSME.

Obtain Keying Material. The certificate management process obtains two CMHs in *Key Pair Only* state. One CMH (the "signing CMH") references a keypair for a signing algorithm. The public key from this keypair will be the verification key in the certificate request. The other CMH (the "response decryption CMH") references a keypair for an encryption algorithm. The public key from this keypair will be the response encryption key in the certificate request.

Generate Certificate Request (Figure D.6; Figure D.7, step 4). The certificate management process determines which certificates to request. This determination is made using logic outside the scope of this standard. See E.6.4 for discussion. For each certificate to be requested, the certificate management process then obtains a CMH for the verification keypair and a CMH for the response encryption keypair using the primitives defined in 5.2.3, requests a certificate request via Sec-CertificateRequest.request. and obtains a signed, encrypted certificate request via Sec-CertificateRequest.confirm (assuming all inputs are valid). Sec-CertificateRequest.confirm also returns a ten-byte hash of the plaintext of the certificate request. The certificate management process stores the certificate request and the associated LSI-Ss in its internal storage.

NOTE 3—This standard does not define the logic or processing necessary to determine when to request certificates or which certificates to request.

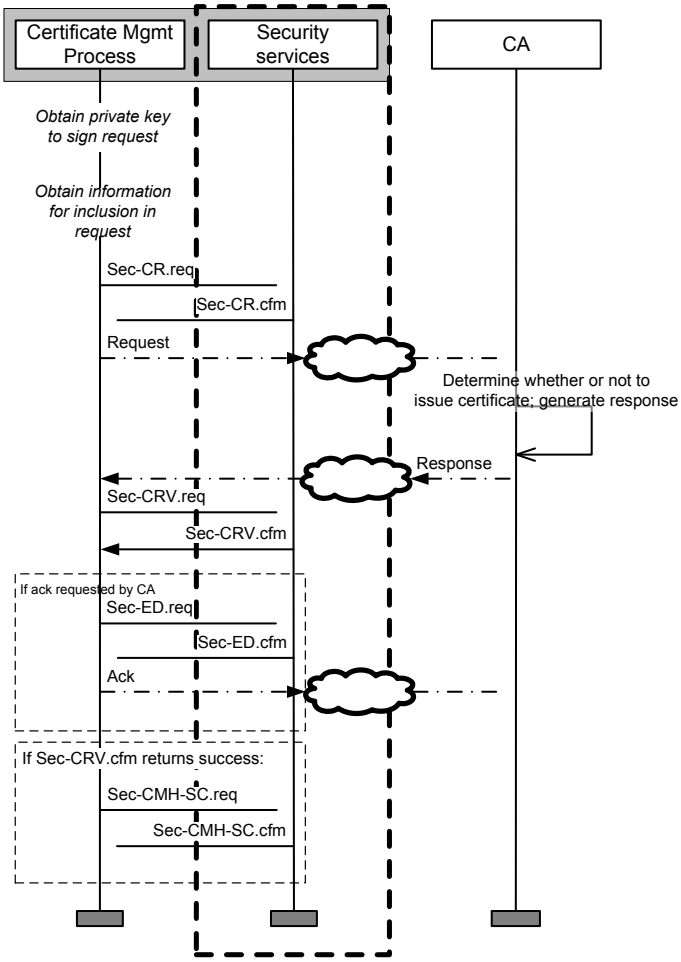


Figure D.6—Interactions between certificate management process and CA for WSA certificate request

Interaction with CA (Figure D.6; Figure D.7, step 4). The certificate management process transmits the certificate request to the CA.

The CA determines whether or not to issue the certificate and returns either the certificate response or an error message. This response is encrypted. See D.3.4 for a discussion of possible processing by the CA.

Process Response (Figure D.6; Figure D.7, step 4). The certificate management process receives the response and determines which decryption key to pass to Sec-CertificateResponseProcessing.request by inspecting the headers of the encrypted response, matching this to the public key that was provided to encrypt the response and matching that to a response decryption CMH. It invokes Sec-CertificateResponse-Processing.request to request that the security processing services decrypt the response and (if appropriate and possible) verify the contents.

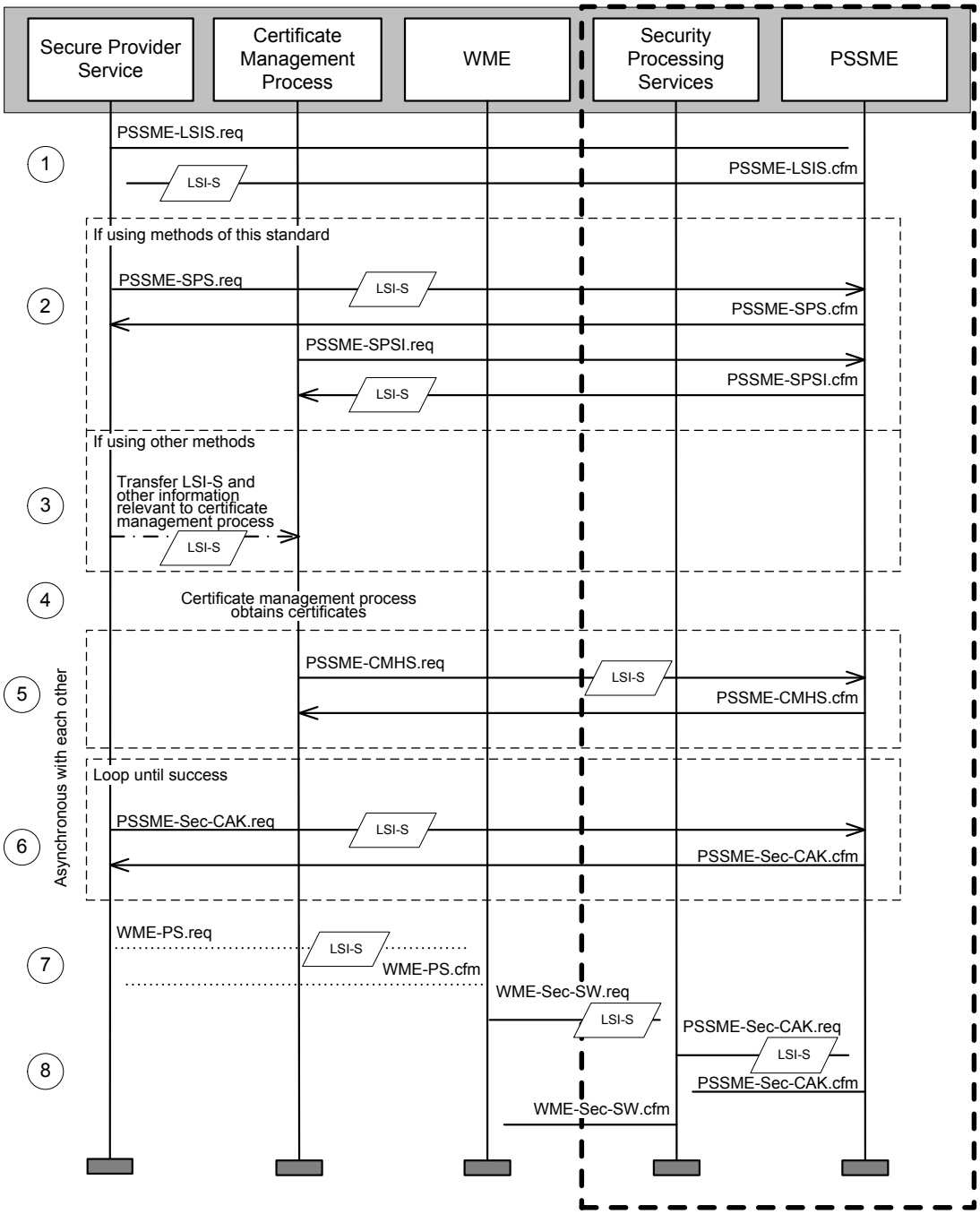


Figure D.7—Process flow from unintialized provider service to ability to send WSAs advertising that provider service

If the CA declined to issue the certificate, the response contains an encrypted ToBeEncryptedCertificate-RequestError. The security processing services extract the information from the ToBeEncryptedCertificate-RequestError structure and return an appropriate notification to the certificate management process via Sec-CertificateResponseProcessing.confirm. This includes a 10-byte hash of the request. The hash allows the certificate management process to search the hashes of all outstanding requests as returned by Sec-CertificateRequest.confirm and thereby determine which certificate request was rejected.

If all operations succeed and the certificate is verified, the security processing services return the certificate to the certificate management process via Sec-CertificateResponseProcessing.confirm. and update the certificate information stored by the CME using the mechanisms supported by CME-AddCertificateRevocation.request.

As part of response processing, the security services update the CME with information received in the response.

Acknowledgement (Figure D.6; Figure D.7, step 4). If so requested by the CA, the receiving certificate management process may send an acknowledgment. This acknowledgment consists of a ToBeEncrypted-CertificateResponseAcknowledgment generated as described in 6.3.39 and which is then encrypted by invoking Sec-EncryptedData.request. The certificate management process sends the acknowledgement to the CA. No response from the CA is specified.

Store Certificate (Figure D.6; Figure D.7, step 5**)**. The certificate management process stores the certificate and associated LSI-Ss at the CMH via Sec-CryptomaterialHandle-StoreCertificate.request. If the certificate is an implicit certificate, the certificate response also includes information to be used to update the private key. This information is specified in the specification of Sec-CryptomaterialHandle-Store-Certificate.request. The certificate management process then passes the CMH to the PSSME via PSSME-CryptomaterialHandleStorage.request.

Secure provider service registers with WME (Figure D.7, step 6 and step 7**).** The secure provider service should not request inclusion in a WSA until the device has a certificate that can be used to sign that WSA. To establish whether or not a certificate exists, the secure provider service may poll the PSSME with PSSME-Sec-CryptomaterialHandle.request. The secure provider service should not request inclusion in a WSA until PSSME-Sec-CryptomaterialHandle.confirm returns success, indicating that a certificate for that secure provider service is available to the WME.

The secure provider service requests inclusion in a WSA via WME-ProviderService.request as specified in IEEE Std 1609.3. This passes the service's LSI-S, as well as other information about the service, to the WME.

Generate signed WSA (Figure D.7, step 8): The WME uses the service's LSI-S as a parameter to WME-Sec-SignedWsa.request when it requests the signing of a WSA prior to transmission. This allows the PSSME to locate an appropriate WSA signing certificate, if one exists.

D.3.3 Certificate request using a enrolment certificate

Figure D.8 illustrates an example process flow for requesting certificates with enrolment certificates.

a) A secure data exchange entity registers with the certificate management process to request certificates.

b) The certificate management process requests a enrolment certificate from the CA in a message that may be self-signed or may be signed with another enrolment certificate.

c) When the enrolment certificate is returned, the certificate management process requests a communications certificate from the CA.

d) When the communications certificate is received, the certificate management process makes the certificate and private key available to the secure data exchange entity.

Many alternative flows are possible. For example, Figure D.8 implies that the certificate management process generates the public and private key and passes the private key to the secure data exchange entity.

Alternatively, the key pair could be generated by the secure data exchange entity and only the public key could be passed to the certificate management process.

A similar process flow may also be used to request certificates for WSA signing using a enrolment certificate.

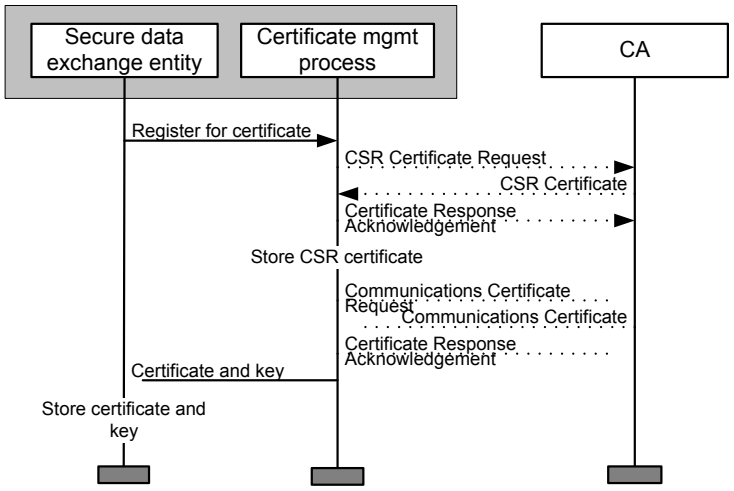


Figure D.8—High-level process flow for certificate request with enrolment certificates

D.3.4 Certificate request processing by the CA

D.3.4.1 General

The CA is a functional element of certificate management. CA operations are out of scope of this standard. This annex presents a description of reasonable processing that the CA may carry out when it receives a certificate request. This description of processing is intended to be illustrative.

D.3.4.2 Certificate request reception processing

Once a CA receives a CSR, it determines whether the requesting Wireless Access in Vehicular Environments (WAVE) device is eligible to be issued the requested certificate. This involves the following checks:

— If there is an `permissions` field in the enrolment certificate, the `permissions` field in the scope of the certificate request is consistent with it as established by the methods of 5.6.1.2.

— If there is a `region` field in the enrolment certificate, the `region` field in the certificate request represents a region that is completely contained within the `region` in the enrolment certificate.

If the CA approves the issuance of the certificate, the CA formats the appropriate ToBeSignedCertificate structure and signs it to generate a Certificate value.

For a secure data exchange certificate, the CA is not bound by the enrollee's holder type or scope requests. The CA may replace any part of the scope with `from_issuer` as appropriate. If issuing a certificate with

`holder_type sde_identified_not_localized`, it shall select the contents of the `cert_specific_data` identifier field according to its own local policies.

For a WSA certificate, the PsidPrioritySspArray in the WsaScope in a certificate should be unchanged from the one in the corresponding certificate request, so that the LSI-S-to-permission mapping that applied to the request also applies to the certificate. Implementations may define extensions to the certificate request and response structures in this standard that allow unambiguous linkage of permissions and LSI-S by some other means, and in this case the permissions may appear in a different order in the certificate than in the request.

An enrollee should accept an explicit certificate in response to a request for an implicit certificate, but it should reject an implicit certificate in response to a request for an explicit certificate.

The CA may issue a valid certificate of any type or scope of its choosing. A CA should not issue an invalid certificate.

If the CA cannot issue a certificate, it should issue a certificate request error.

D.3.4.3 Certificate request response: sending

If the CA issues a certificate, the following processing produces a correct response:

a) Generate a Certificate as specified above.

b) Fill in a ToBeEncryptedCertificateResponse structure with the freshly generated Certificate and the other fields specified in 6.3.36.

   1) If the CA desires an acknowledgement, it sets the flags field `f` to encode the value 0.

   2) The `certificate_chain` contains the certificate chain of the new certificate. This path is in order, with the most local certificate (the newly issued one) being first and each successive certificate signing the one before it. The path should be complete with the final certificate being a trust anchor. However, some implementations may choose to deliver less complete paths for space reasons.

   3) If the certificate is an implicit certificate, the field `recon_priv` is set equal to the octet string representation of *s* calculated in step 11 of 2.4.2 of SEC 4.

   4) The `crl_path` contains the CRLs necessary to verify the certificate. At minimum, it contains the most recent version of the CRL series on which the issued certificate would appear if it were revoked. In addition, the field should include CRLs corresponding to other CAs in the chain. These CRLs are not ordered.

c) Place the ToBeEncryptedCertificateResponse structure in a ToBeEncrypted structure with ContentType set to certificate_response.

d) Encrypt the ToBeEncrypted structure to form an EncryptedData. The EncryptedData contains a single RecipientInfo, in which the `cert_id` field is the low-order eight octets of the SHA-256 hash of the response_encryption_key field from the ToBeSignedCertificateRequest. The symmetric key is freshly generated.

e) Encapsulate the EncryptedData in a 1609Dot2Data.

f) Return the response to the CME on the requesting WAVE device. (How the requesting WAVE device is identified for routing purposes is out of scope of this standard.)

g) If the CA requested an acknowledgement it waits for a policy-determined length of time. If at the end of that time it has not received an acknowledgement it retransmits the response. It repeats this step a policy-determined number of times.

D.3.4.4 Certificate request error: sending

If the CA does not issue a certificate, it may issue a Certificate Request Error in response to the certificate request. The following processing produces a correctly formed Certificate Request Error:

a) Fill in a ToBeEncryptedCertificateRequestError structure with the appropriate CertificateRequest-ErrorCode.

b) Generate the signature for the ToBeEncryptedCertificateRequestError using the signing key that would have signed the certificate, if one had been issued.

c) Place the ToBeEncryptedCertificateResponse in a ToBeEncrypted structure with ContentType set to `certificate_request_error`.

d) Encrypt the ToBeEncrypted structure to form an EncryptedData. The EncryptedData contains a single RecipientInfo, in which the `cert_id` field is the low-order eight octets of the SHA-256 hash of the response_encryption_key field from the ToBeSignedCertificateRequest. The symmetric key shall be freshly generated.

e) Return the response to the CME on the requesting WAVE device. (How the requesting WAVE device is identified for routing purposes is out of scope of this standard.)

D.4 Signed WSA: full example with certificate request and WSA processing

D.4.1 General

The following is an illustrative example of the use of the LSI-S in the primitives and the `permissions_indices` field in the SignedWsa type to allow secure provider services to demonstrate their permissions to a receiving user service. Refer to D.3.2 for an overview of certificate request for WSAs. In particular, Figure D.7 gives a process flow that is followed by the processing on the provider device, up to the point where the signed WSA is scheduled for transmission.

In the description below (as in the rest of the standard) all primitive names are hyperlinked to the section in which they are defined.

D.4.2 Functional entities

This section describes the actions taken by the functional entities on the provider device and by the CA before the provider device is able to send signed WSAs. The functional entities on the provider device are shown in Figure D.9. The functional entities involved in communicating over the network are shown in Figure D.10.

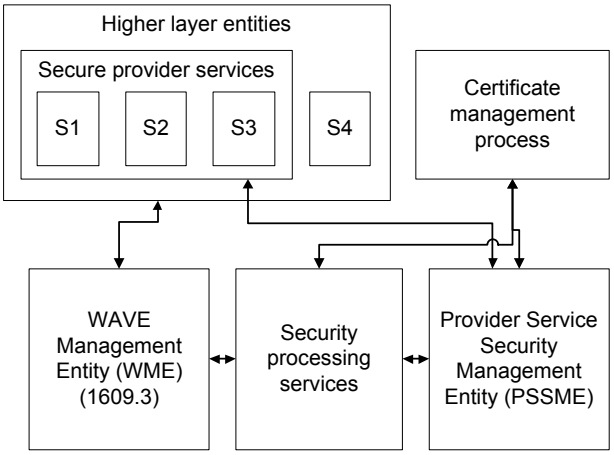


Figure D.9 — Functional entities on the provider device

NOTE— This is an illustrative example. Services used in the example are not necessarily services that will be deployed in a real system. PSIDs used in the example below are illustrative only and are not intended to imply that those PSIDs are reserved for the uses indicated; the process for applying for PSIDs, and the current list of allocated PSIDs, is defined in IEEE Std 1609.12-2012. SSPs used in the example below are designed for readability and are not intended to be an example of good SSP design.

In the example, there are four higher layer entities interacting with the WME on the device. Three of these higher layer entities are secure provider services. The entities are as follows:

— Higher Layer Entity S1 is a local tourist information service offering information about points of interest to subscribers. The service is provided by Alice, who operates a server accessible through the provider device. The service is characterized as follows:

— The service is secured (because information is available only to subscribers and so a response to the service is potentially privacy-compromising).

— The PSID is E0 00 00 01.

There are two different options for subscribers to consume information: Terse or Detailed.

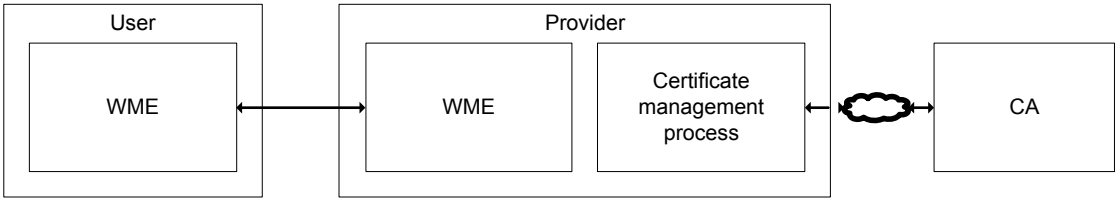


Figure D.10—Networking entities

The Terse information is less informative but is sent at a higher priority. The two options are distinguished via the SSP. (The SSP is used to identify the options because it includes the provider name, which is persistent, authenticated information, and because the choice the user makes may affect how much they need to pay for the information.)

— Higher Layer Entity S2 is a competing local tourist information service operated by Bob. It is characterized as follows:

— The service is secured (because information is available only to subscribers and so a response to the service is potentially privacy-compromising).

— The PSID is E0 00 00 01.

— The SSP is “Provider = Bob”.

— The service may be sent at priority less than or equal to 32.

— Higher Layer Entity S3 offers information about electric vehicle charging points. It is characterized as follows:

— The service is secured (to provide assurance that the information is valid).

— The PSID is E0 00 00 02.

— The SSP is “Provider = Alice”.[30]

— The sevice may be sent at priority less than or equal to 48.

— Higher Layer Entity S4 offers advertising-supported internet access. It is characterized as follows:

— The service is not secured (because responding to the service does not reveal anything about the user).

— The PSID is E0 00 00 03.

— The service may be sent at priority less than or equal to 32.

The certificate management process at the provider device is assumed to already have a enrolment certificate to be used when applying for WSA certificates. This certificate and the corresponding private key are stored at the Cryptomaterial Handle with value 1.

The certificate management process also has access to the following persistent security management information:

— The certificate of a trusted CA that is able to issue WSA certificates.

— The network address of that CA.

The certificate management process also stores the following persistent information about the provider device:

— Identifier = “RSE 123”

— Latitude = 38.87641

— Longitude = -77.00406

— Region of validity: Circular, with Latitude and Longitude as above and radius 10 m

— Certificate Lifetime = 10 days

[30] This is the same string as the SSP for service S1, but the interpretation of an SSP depends on the associated PSID and so the strings may have different meanings in context. In particular, there is no implication that the same Alice is providing both services.

D.4.3 Initialization

D.4.3.1 Obtaining LSI-S

See Figure D.7, step 1. The three secured services request and obtain LSI-Ss:

— S1 invokes PSSME-LocalServiceIndexForSecurity.request and in the corresponding confirm primitive receives the Local Service Index for Security `1`, an integer.

— S2 invokes PSSME-LocalServiceIndexForSecurity.request and in the corresponding confirm primitive receives the Local Service Index for Security `2`, an integer.

— S3 invokes PSSME-LocalServiceIndexForSecurity.request and in the corresponding confirm primitive receives the Local Service Index for Security `3`, an integer.

The unsecured service S4 does not need to request an LSI-S.

D.4.3.2 Registering secure provider services with the PSSME

See Figure D.7, step 2. The three secured services register with the PSSME so that the certificate management process can request certificates:

— S1 registers its terse tourist information service by invoking PSSME-SecuredProviderService.request with parameters: *Local Service Index for Security* = `1`; *Action* = "Add"; *PSID* = E0 00 00 01; *Maximum Service Priority* = 32; *Service Specific Permissions* = "Provider = Alice, Type = Terse".

— S1 registers its detailed tourist information service by invoking PSSME-SecuredProviderService.request with parameters: *Local Service Index for Security* = `1`; *Action* = "Add"; *PSID* = E0 00 00 01; *Maximum Service Priority* = 16; *Service Specific Permissions* = "Provider = Alice, Type = Detailed".

— S2 registers its alternative tourist information service by invoking PSSME-SecuredProviderService.request with parameters: *Local Service Index for Security* = `2`; *Action* = "Add"; *PSID* = E0 00 00 01; *Maximum Service Priority* = 32; *Service Specific Permissions* = "Provider = Bob".

— S3 registers its electric vehicle charging point service by invoking PSSME-SecuredProviderService.request with parameters: *Local Service Index for Security* = `3`; *Action* = "Add"; *PSID* = E0 00 00 02; *Maximum Service Priority* = 48; *Service Specific Permissions* = "Provider = Alice".

The unsecured service S4 does not need to register with the PSSME.

D.4.3.3 Generate certificate request

See Figure D.7, step 4. The certificate management process polls the PSSME and obtains the permissions that were registered in the previous section. Using implementation-specific logic (see E.6.4), it determines that the following two certificates should be applied for:

— One certificate containing all the permissions registered with the PSSME, for normal use.

— One certificate containing all the permissions except Alice's detailed tourist information service, for use when the channel is congested.

The certificate management process invokes Sec-CryptomaterialHandle.request for the first certificate to be requested. It receives a Sec-CryptomaterialHandle.confirm with parameters (*Result Code* = Success, *Cryptomaterial Handle* = 32). The certificate management process then invokes Sec-Cryptomaterial-Handle.request for the second certificate to be requested and receives a Sec-CryptomaterialHandle.confirm with parameters *Result Code* = Success, *Cryptomaterial Handle* = 33).

To generate a keypair for use with the first certificate, the certificate management process invokesSec-CryptomaterialHandle-GenerateKeyPair.request with parameters (*Cryptomaterial Handle* = 33, *Algorithm* = ECDSA NIST P256). It receives Sec-CryptomaterialHandle-GenerateKeyPair.confirm with parameters (*Result Code* = Success, *Public Key* = `pk1`), where `pk1` denotes a valid ECDSA NIST P256 public key. The certificate management process repeats the steps for the second certificate, obtaining `pk2`, a valid ECDSA NIST P256 public key.

The certificate management process also invokes Sec-CryptomaterialHandle.request, followed by Sec-CryptomaterialHandle.request with *Algorithm* = ECIES NIST P256, twice to obtain the response encryption keys for the certificate requests. These response encryption keys have Cryptomaterial Handle values equal to 34 and 35, and the public keys will be denoted by `pk3` and `pk4`, respectively.

The certificate management process then generates two certificate requests by invoking Sec-Certificate-Request.request. For the first certificate request it sets the parameters to Sec-CertificateRequest.request as follows:

- *Signer Type* = Certificate
- *CSR Signing Cryptomaterial Handle* = 1 (see D.4.2)
- *Certificate Holder Type* = WSA Signer
- *Public Key Transfer Type* = Implicit
- *Permissions Type* = PsidPrioritySsp
- *Permissions Array* = [

  (*PSID* = E0 00 00 01, *Max Priority* = 32, *SSP* = “Provider = Alice, Type = Terse”),

  (*PSID* = E0 00 00 01, *Max Priority* = 16, *SSP* = “Provider = Alice, Type = Detailed”),

  (*PSID* = E0 00 00 01, *Max Priority* = 32, *SSP* = “Provider = Bob”),

  (*PSID* = E0 00 00 02, *Max Priority* = 48, *SSP* = “Provider = Alice”))

  ]

  (see the definition of PsidPrioritySsp)
- *Identifier* = “RSE 123” (see D.4.2)
- *Geographic Region* =

  (`region_type` = `circle`, `center.latitude` = 38,876,410, `center.longitude` = -77,004,060), Radius = 10) (see D.4.2 and the definition of GeographicRegion)
- *Use Start Validity* = True
- *Lifetime is Duration* = True
- *Start Validity* = True
- *Expiration* = Current time + ten days(see D.4.2)
- *Verification Public Key* = `pk1`

— *Encryption Public Key* = NULL.

— *Response Encryption Key* = `pk3`

— *CA Certificate* = the known CA certificate (see D.4.2)

For the second certificate request, the parameters are identical except for the following:

— *Permissions Array* = [

(*PSID* = E0 00 00 01, *Max Priority* = 32, *SSP* = "Provider = Alice, Type = Terse"),

(*PSID* = E0 00 00 01, *Max Priority* = 32, *SSP* = "Provider = Bob"),

(*PSID* = E0 00 00 02, *Max Priority* = 48, *SSP* = "Provider = Alice"))

]

— *Verification Public Key* = `pk2`

— *Response Encryption Key* = `pk4`

The certificate management process receives two Sec-CertificateRequest.confirm primitives with the resulting certificate requests and sends them to the CA using the addressing information noted in D.4.2.

The certificate management stores the following data:

— (request1, LSI-S array = (`1`, `1`, `2`, `3`), verification CMH = 32, response encryption CMH = 34, response encryption public key = `pk3`) and

— (request2, LSI-S array = (`1`, `2`, `3`), verification CMH = 33, response encryption CMH = 35, response encryption public key = `pk4`)

D.4.3.4 Receiving certificates

See Figure D.7, step 4 and step 5. The certificate management process receives back the certificate responses from the CA. These responses are 1609Dot2Data encapsulating an EncryptedData. The certificate management process inspects the `recipients` field in each EncryptedData and determines that the first response was encrypted with `pk3` and the second with `pk4`.

The certificate management process then takes the following steps for the first response.

a) **Decrypt response**: It invokes Sec-CertificateResponseProcessing.request with parameters (*Cryptomaterial Handle* = 34, *Data* = the first response) and receives Sec-CertificateResponse-Processing.confirm with *Result Code* = Success, *Type* = Certificate Response, and *Certificate* and *Reconstruction Value* set appropriately as determined by the CA.

b) **Store certificate at CMH**: It invokes Sec-CryptomaterialHandle-StoreCertificate.request with parameters *Cryptomaterial Handle* = 32 and *Certificate* and *Private Key Transformation* set to the values obtained by Sec-CertificateResponseProcessing.confirm. This puts the CMH in the *Key and Certificate* state.

c) **Store CMH with PSSME**: It invokes PSSME-CryptomaterialHandleStorage.request with parameters *Cryptomaterial Handle* = 32, *LSI-S Array* = [1, 1, 2, 3]. This makes the certificate and private key available to the security processing services.[31]

For the second response, it takes the same steps with the following differences in parameters:

d) **Decrypt response**: *Cryptomaterial Handle* = 35, *Data* = the second response.

e) **Store certificate at CMH**: *Cryptomaterial Handle* = 33 and *Certificate* and *Private Key Transformation* set to the values obtained from the second response.

f) **Store CMH with PSSME**: *Cryptomaterial Handle* = 32, *LSI-S Array* = [1, 2, 3]. This makes the certificate and private key available to the security processing services.

D.4.4 Operation

D.4.4.1 Simple case

D.4.4.1.1 Request inclusion in a WSA

See Figure D.7, step 6 and step 7.

S1 confirms that there is a certificate available by invoking PSSME-Sec-CryptomaterialHandle.request, setting the parameters *ProviderServiceIdentifier* = E0 00 00 01; *ServicePriority* = 14; *ServiceSpecific-Permissions* = "Provider = Alice, Type = Detailed"; *LocalServiceIndexForSecurity* = 1, *Location* = (latitude = 38,876,410, longitude = -77,004,060). When the *Result Code* parameter of PSSME-Sec-CryptomaterialHandle.confirm is "Success", the service knows that a certificate is available and so that it is possible to register for a secure provider service with the WME.

The service then invokes WME-ProviderService.request as defined in IEEE Std 1609.3 to request inclusion in a WSA, setting the following parameters: *WSA Type* = Secured; *ProviderServiceIdentifier* = E0 00 00 01; *ServicePriority* = 14; *ServiceSpecificPermissions* = "Provider = Alice, Type = Detailed"; *LocalService-IndexForSecurity* = 1. The other parameters do not need to be specified for this example.

No other services request inclusion in a WSA at this time.

See D.3.2.2 and Figure D.7 for more details.

D.4.4.1.2 Generate signed WSA

See Figure D.7, step 8.

The WME creates a WSA containing a single ServiceInfo. It invokes WME-Sec-SignedWsa.request with parameters *WSA Data* = the WSA; *Permissions* = an array with a single element: (*PSID* = E0 00 00 01, *Priority* = 16, *SSP* = "Provider = Alice, Type = Detailed", *LSI-S* = 1). *Lifetime* need not be specified for this example.

[31] For purposes of this example, the entries in the permissions array in each certificate are taken to be in the same order as they were in the request. The order is straightforward to distinguish, so this is just a simplification to make the example clearer. If the permissions were in a different order, the only change is that the entries in LSI-S would have to be in the corresponding order.

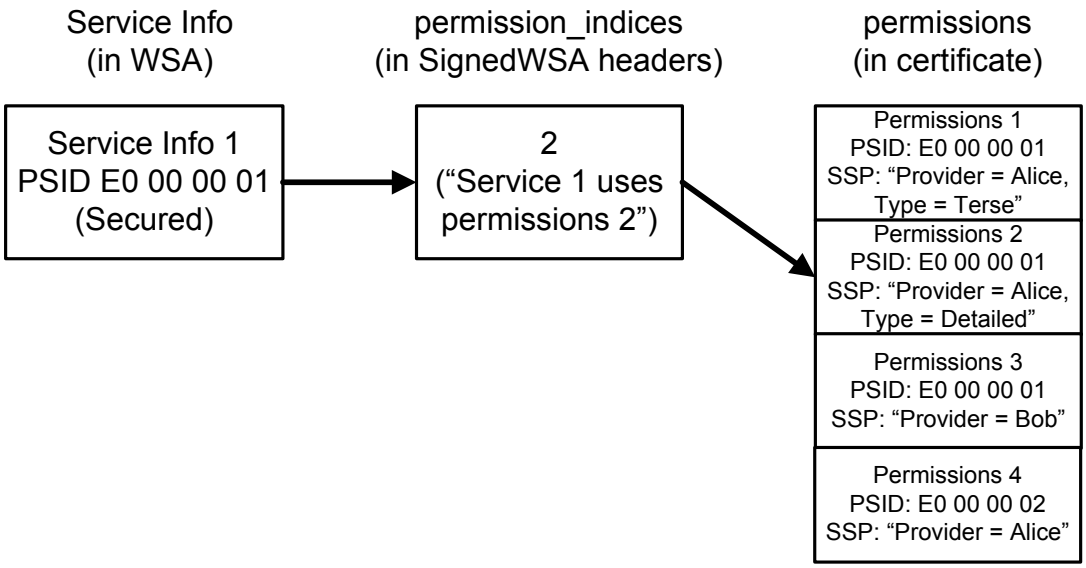


Figure D.11—Mapping from ServiceInfo to permissions via permission_indices for WSA 1

The security services invoke PSSME-Sec-CryptomaterialHandle.request with parameter *Permissions* = an array with a single element: (*PSID* = E0 00 00 01, *Priority* = 16, *SSP* = "Provider = Alice, Type = Detailed", *LSI-S* = `1`).

The PSSME obtains the certificate and generates the pemissions_indices using the following logic:

a) The certificates at CMH = 32 and CMH =33 both correspond to *LSI-S* = 1.

b) Only the certificate at CMH = 32 contains the SSP "Provider = Alice, Type = Detailed", so this is the only remaining candidate to be the signing certificate.

c) The (LSI-S, index) pairs corresponding to cert1 are ((`1`, 1), (`1`, 2), (`2`, 3), (`3`, 4)), so index 1 and 2 correspond to LSI-S `1`. The SSP at index 2 is "Provider = Alice, Type = Detailed", matching the requested SSP.

d) The `max_priority` value at index 2, 16, is greater than the requested priority, 14, so the service is requesting an authorized priority level.

e) Since the (LSI-S, SSP) at index 2 match the requested certificate, the *Permissions Indices* is an array of length 1 containing the value "1".

The PSSME invokes CME-Function-ConstructCertificateChain to construct the certificate chain, with inputs *Identifier Type* = Certificate Array and *Input Certificate Array* = an array of length 1 containing the certificate at CMH = 32. Finally, the PSSME invokes PSSME-Sec-CryptomaterialHandle.confirm to return *Permission Indices, Cryptomaterial Handle* = 32, and the certificate chain to the security services.

The security services form the ToBeSignedWsa, including the encoded `permission_indices`, and sign it with the private key from CMH = 32.

The security services invoke WME-Sec-SignedWsa.confirm to return the signed WSA to the WME. The WME schedules the signed WSA for transmission.

The mapping from ServiceInfo entries to permissions entries is illustrated in Figure D.11.

D.4.4.1.3 Receiving WSA

This section describes activities on the user device shown in Figure D.10.

The user device receives the signed WSA. The WME on the WAVE device invokes the WME-Sec-Signed-WsaVerification.request primitive.

The security processing services take the following steps:

a) Extract the `permissions_indices` field from the signed WSA and obtain an array of length 1 containing the index value "2". (WME-Sec-SignedWsaVerification.request step h)2)).

b) Confirm that there is one ServiceInfo in the WSA. If there was a different number, the signed WSA would be incorrectly formed and the security processing services would return the Result Code "invalid input" in WME-Sec-SignedWsaVerification.confirm.

c) Use the index value "2" to look up entry number 2 in the permissions field in the cert. This entry is (E0 00 00 01, 16, "Provider = Alice, Type = Detailed").

d) Confirm that the first ServiceInfo is permitted by the permissions at index 2, that is:

   1) The ServiceInfo uses PSID = E0 00 00 01

   2) The ServiceInfo is at priority equal to or below the `max_priority` value at index 2, (i.e., $14 \leq 16$)

   Since both of these hold, the WSA permissions are consistent with the certificate.

e) Invoke the WME-Sec-SignedWsaVerification.confirm primitive to return the result with the Service Specific Permissions parameter set equal to an array with a single entry, which is "Provider = Alice, Type = Detailed".

In addition, the security processing services carry out the certificate chain construction and verification as specified in the description of WME-Sec-SignedWsaVerification.request. The security services then use the WME-Sec-SignedWsaVerification.confirm primitive to return the result including Result Code_array and Service Specific Permissions.

The WME on the user device creates an *AvailableServiceEntry* within the MIB for PSID E0 00 00 01 as described in IEEE Std 1609.3.

D.4.4.2 Complex case

D.4.4.2.1 Request inclusion in a WSA

In this example, which follows the previous one, the single advertised service is replaced in the WSA with several services.

Before requesting inclusion in the WSA, each secured service confirms that a certificate is available.

— S1 invokes PSSME-Sec-CryptomaterialHandle.request, setting the parameters *ProviderServiceIdentifier* = E0 00 00 01; *ServicePriority* = 32; *ServiceSpecificPermissions* = "Provider = Alice, Type = Terse"; *LocalServiceIndexForSecurity* = `1`, *Location* = (`latitude` = 38,876,410, `longitude` = -77,004,060). When the *Result Code* parameter of PSSME-Sec-

CryptomaterialHandle.confirm is "Success", the service knows that a certificate is available and so that it is possible to register for a secure provider service with the WME.

— S2 invokes PSSME-Sec-CryptomaterialHandle.request, setting the parameters *ProviderServiceIdentifier* = E0 00 00 01; *ServicePriority* = 32; *ServiceSpecificPermissions* = "Provider = Bob"; *LocalServiceIndexForSecurity* = `1`, *Location* = (`latitude` = 38,876,410, `longitude` = -77,004,060). When the *Result Code* parameter of PSSME-Sec-Cryptomaterial-Handle.confirm is "Success", the service knows that a certificate is available and so that it is possible to register for a secure provider service with the WME.

— S4 is not secured.

S1 replaces the detailed tourist information service with the terse tourist information services. It does this in two steps: first it deregisters the detailed tourist information service by invoking WME-ProviderService with *Action* = "Delete", then it invokes WME-ProviderService to request inclusion in a WSA, setting the following parameters: *WSA Type* = Secured; *ProviderServiceIdentifier* = E0 00 00 01; *ServicePriority* = 32; *ServiceSpecificPermissions* = "Provider = Alice, Type = Terse"; *LocalServiceIndexForSecurity* = `1`. The other parameters do not need to be specified for this example.

S2 invokes WME-ProviderService to request inclusion in a WSA, setting the following parameters: *WSA Type* = Secured; *ProviderServiceIdentifier* = E0 00 00 01; *ServicePriority* = 32; *ServiceSpecificPermissions* = "Provider = Bob"; *LocalServiceIndexForSecurity* = `2`. The other parameters do not need to be specified for this example.

S4 uses WME-ProviderService to request inclusion in a WSA, setting the following parameters: WSA Type = Unsecured; *ProviderServiceIdentifier* = E0 00 00 03; *ServicePriority* = 32; *ServiceSpecific-Permissions* = NULL; *LocalServiceIndexForSecurity* = NULL. The other parameters do not need to be specified for this example.

At this point, three applications have requested inclusion in the broadcast WSA.

D.4.4.2.2 Generate signed WSA

The WME creates a WSA containing three ServiceInfos. For the sake of illustration, assume that it contains first the ServiceInfo for S4, then the ServiceInfo for S1, then the ServiceInfo for S2. It invokes WME-Sec-SignedWsa.request primitive with the following parameters (*Lifetime* is ignored for purposes of this example):

— *WSA Data* = the WSA;

— *Permissions* = [

(*PSID* = E0 00 00 03, *Priority* = 32, *SSP* = NULL, *LSI-S* = `0`),

(*PSID* = E0 00 00 01, *Priority* = 32, *SSP* = "Provider = Alice, Type = Terse", *LSI-S* = `1`),

(*PSID* = E0 00 00 01, *Priority* = 32, *SSP* = "Provider = Bob", *LSI-S* = `2`),

]

The PSSME obtains the certificate and generates the pemissions_indices using the following logic:

a) The certificates at CMH = 32 and CMH =33 both correspond to both *LSI-S* = 1 and *LSI-S* = 2.

b) Only the certificate at CMH = 33 contains the SSP "Provider = Alice, Type = Terse", so this is the only remaining candidate to be the signing certificate.

c) The (LSI-S, index) pairs corresponding to cert1 are ((1, 1), (2, 2), (3, 3)), so index 1 corresponds to LSI-S 1 and index 2 corresponds to LSI-S 2. The SSP at indices 1 and 2 both match the SSP requested for the corresponding LSI-S, so this certificate is acceptable.

d) The `max_priority` values at indices 1 and 2, both 32, are less than or equal to the requested priorities, also both 32, so the services are requesting an authorized priority level.

e) The *Permissions Indices* is an array of length 3 containing the values (0, 1, 2).

The PSSME invokes CME-Function-ConstructCertificateChain to construct the certificate chain, with inputs *Identifier Type* = Certificate Array and *Input Certificate Array* = an array of length 1 containing the certificate at CMH = 33. Finally, the PSSME invokes PSSME-Sec-CryptomaterialHandle.confirm to return *Permission Indices, Cryptomaterial Handle* = 33, and the certificate chain to the security services.

The security services form the ToBeSignedWsa, including the encoded `permission_indices`, and sign it with the private key from CMH = 33.

The security services invoke WME-Sec-SignedWsa.confirm to return the signed WSA to the WME. The WME schedules the signed WSA for transmission.

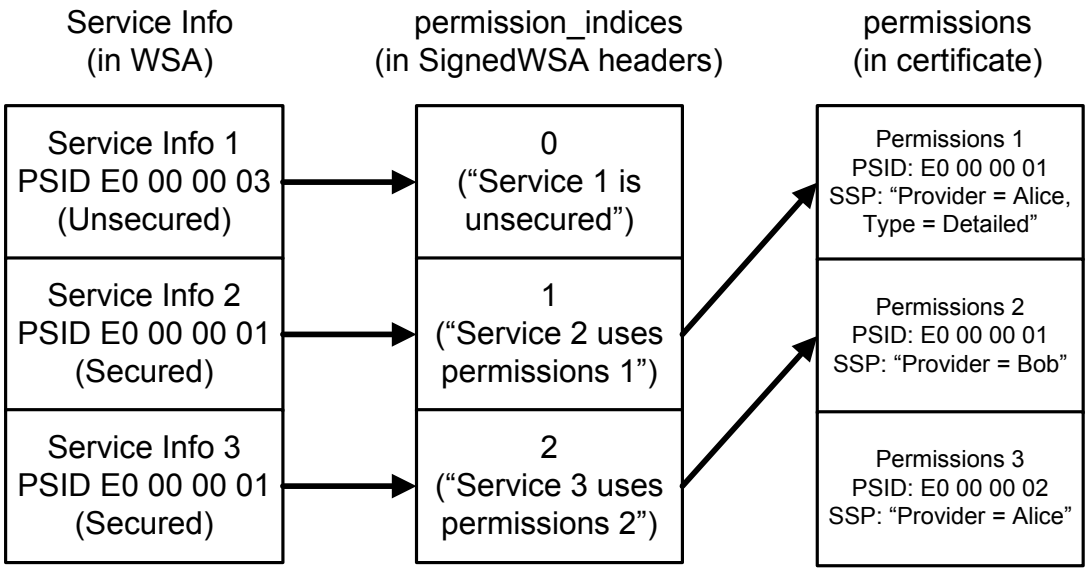


Figure D.12 — Mapping from ServiceInfo to permissions via permission_indices for WSA 2

The mapping from ServiceInfo entries to permissions entries is illustrated in Figure D.12.

D.4.4.3 Receiving WSA 2

This section describes activities on the user device shown in Figure D.12.

The user device receives the signed WSA. The WME on the WAVE device invokes the WME-Sec-Signed-WsaVerification.request primitive.

The security processing services take the following steps:

a) Extract the `permissions_indices` field from the signed WSA and obtain an array of length 3 containing the index values (0, 1, 2). (WME-Sec-SignedWsaVerification.request step h)2)).

b) Confirm that there are three ServiceInfos in the WSA. If there was a different number, the signed WSA would be incorrectly formed and the security processing services would return the Result Code "invalid input" in WME-Sec-SignedWsaVerification.confirm.

c) Use the index value "1" to look up entry number 1 in the permissions field in the cert. This entry is (E0 00 00 01, 32, "Provider = Alice, Type = Terse"). Confirm that the second ServiceInfo is permitted by the permissions at index 2, that is:

   1) The ServiceInfo uses PSID = E0 00 00 01

   2) The ServiceInfo is at priority equal to or below 32

d) Use the index value "2" to look up entry number 2 in the permissions field in the cert. This entry is (E0 00 00 01, 32, "Provider = Bob"). Confirm that the third ServiceInfo is permitted by the permissions at index 2, that is:

   1) The ServiceInfo uses PSID = E0 00 00 01

   2) The ServiceInfo is at priority equal to or below 32

e) Invoke the WME-Sec-SignedWsaVerification.confirm primitive to return the result with the Service Specific Permissions parameter set equal to an array with a three entries, which are (NULL, "Provider = Alice, Type = Terse", "Provider = Bob").

In addition, the security processing services carry out the certificate chain construction and verification as specified in the description of WME-Sec-SignedWsaVerification.request. The security services then use the WME-Sec-SignedWsaVerification.confirm primitive to return the result including Result Code_array and Service Specific Permissions.

The WME on the user device creates two *AvailableServiceEntry* values within the MIB for PSID E0 00 00 01 as described in IEEE Std 1609.3. Both of these have *UserAvailableResultCode* set equal to "success", denoting that the service within the service advertisement was secured and successfully verified.

The WME on the user device creates a third *AvailableServiceEntry* within the MIB for PSID E0 00 00 03 as described in IEEE Std 1609.3. This has *UserAvailableResultCode* set equal to "unsecured", denoting that the ServiceInfo within the service advertisement was not authenticated.

D.5 Processing CRLs

Figure D.13 illustrates a process flow for the processing of a CRL by a certificate management process using the primitives defined in this standard. Primitive names in the figure are abbreviated for compactness.

The process invokes Sec-CRLVerification.request to verify the CRL. The security services invoke CME-CertificateInfo.request to gain assurance that the CRL signing certificate is valid and then return CRLValidation.confirm indicating whether or not the CRL is valid.

If the CRL is valid, the certificate management process extracts the revocation information about the individual certificates and stores it in the CME via CME-AddCertificateRevocation.request. It provides additional information about the CRL, such as the next CRL issue date, using a CME-AddCrlInfo.request.

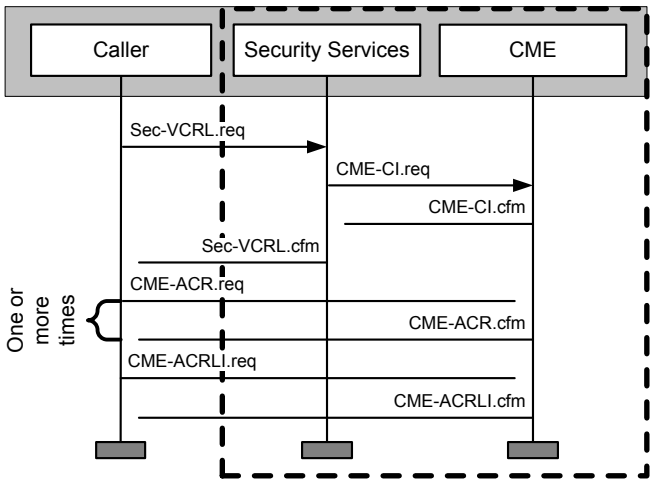


Figure D.13—Process flow for processing a received Certificate Revocation List (CRL)

The functional entities in CRL request are as follows:

— A certificate management process with access to an instance of WAVE Security Services.

— A CRL distribution center. This may be colocated with a CA or it may be in a different location.

The process flow for requesting and distributing a CRL is illustrated in Figure D.14, where heavy dashed lines indicate processing defined in this standard, solid arrows indicate primitives defined in this standard and dot-dashed arrows indicate data communications using primitives not defined in this standard.

While specification of the CRL distribution center is outside the scope of this standard, it is assumed that a CRL distribution center that receives a CRL request responds by sending the requested CRL, if available.

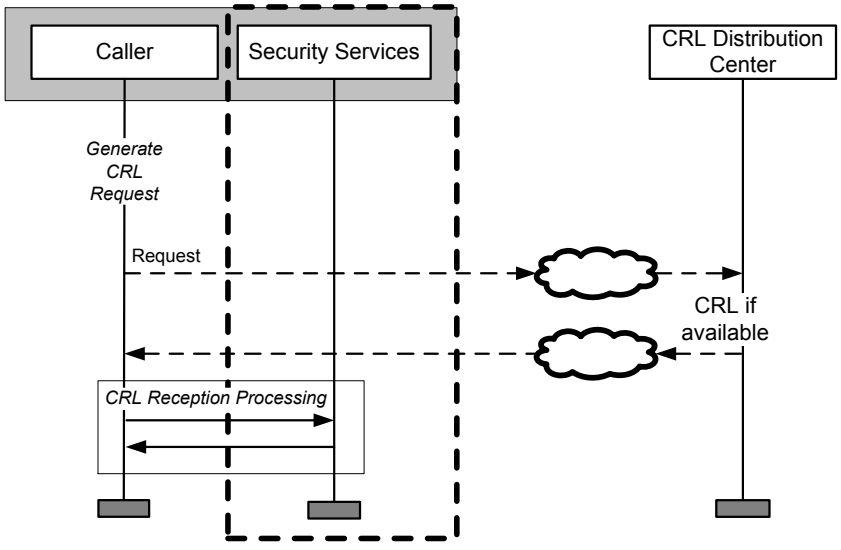


Figure D.14 — Process flow for CRL request

D.6 Constructing a certificate chain

D.6.1 Examples

Figure D.15 shows a simple certificate chain of length 2 (the length of a certificate chain is the number of certificates in it). The trust anchor is the root CA certificate. The end-entity certificate and the root CA certificate are both explicit certificates. The signer id in the end-entity certificate identifies the root CA certificate that issued it. The public key in the root CA certificate is used to verify the signature on the end-entity certificate. If the end-entity certificate is used to sign a communication, the signer field in the signed communication identifies the specific end-entity certificate that signed the communication and the public key field in the end-entity certificate is used to verify the signature on the communication.

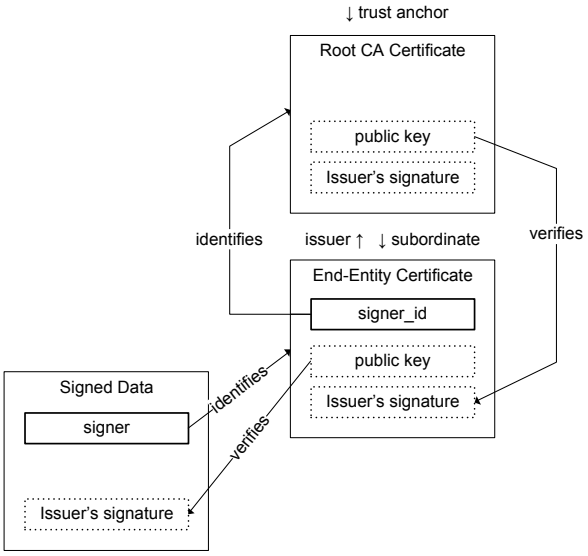


Figure D.15—Length-2 certificate chain with explicit certificates and a root CA as trust anchor

Figure D.16 shows a certificate chain of length 2, where the end-entity certificate and the trust anchor are both explicit certificates. In this case, the trust anchor is a CA certificate rather than a root certificate. Other than that, the chain is identical to the one presented in Figure D.15.

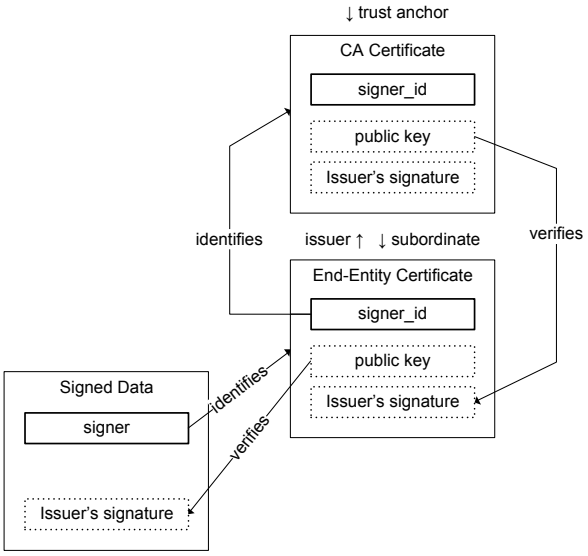


Figure D.16—Length-2 certificate chain with explicit certificates and a non-root CA as trust anchor

Figure D.17 shows a certificate chain of length 2, where the end-entity certificate is an implicit certificate. As required, the trust anchor is an explicit certificate. As with explicit certificates, the signer id in the end-entity certificate identifies the root CA certificate that issued it. However, in this case the issuing certificate does not sign the subordinate certificate. Instead, to cryptographically verify the certificate, an operation is performed combining the hash of the issuing certificate, the hash of the end-entity certificate, the reconstruction value from the end-entity certificate, and the public key from the CA certificate to reconstruct the end-entity's public key as specified in 5.8.6. If that public key cryptographically verifies the signature, this provides assurance both that the end-entity did, in fact, sign the communication and that the certificate was validly issued by the CA.

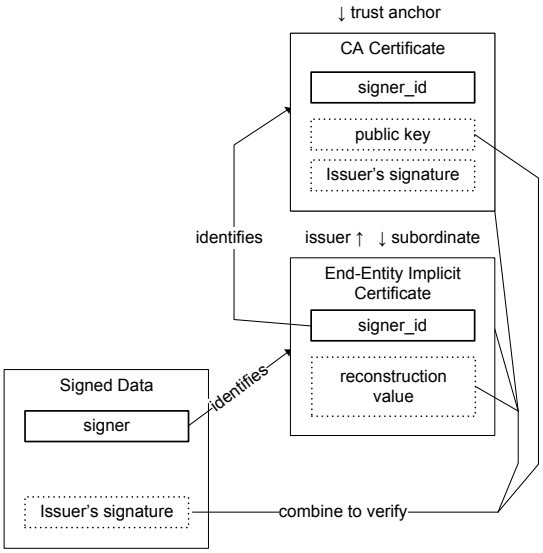


Figure D.17—Length-2 certificate chain with implicit certificates and a non-root CA as trust anchor

Figure D.18 illustrates two longer certificate chains. It shows that for each pair of certificates in the chain, the subordinate certificate contains a signer_id identifying the issuing certificate and the issuing certificate's public key verifies the subordinate certificate either explicitly or implicitly.

The chain on the left contains explicit certificates only. The public key in each issuing certificate verifies the signature of its subordinate certificate.

The chain on the right ends with two implicit certificates. The validity of the implicit certificates is demonstrated by combining the hashes of all the implicit certificates, the reconstruction values of all of the implicit certificates and the hash and public key from the first explicit certificate in the chain to verify the signature on the signed data as specified in 5.8.6.

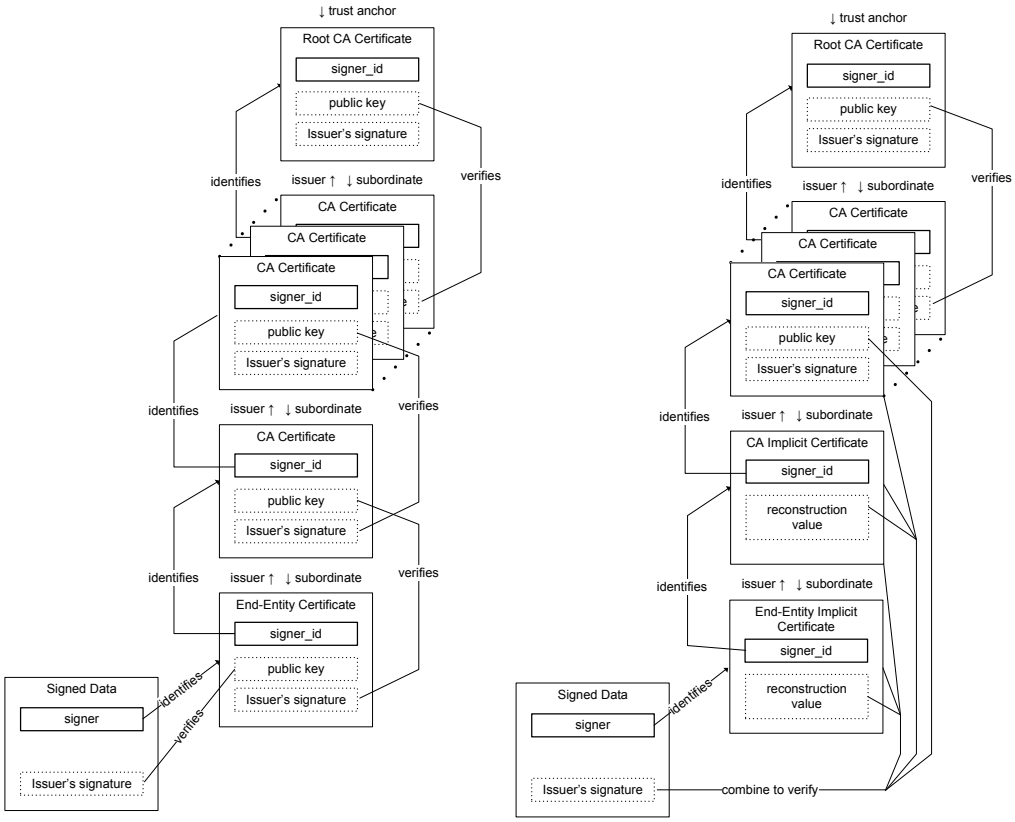


Figure D.18—Long certificate chains, one with all explicit certificates (on the left) and one ending with two implicit certificates (on the right)

D.6.2 Construction

A signed communication need not include all of the certificates in the chain. Instead, the communication may include a chain of one or more certificates that does not reach all the way back to a trust anchor or the communication may omit all certificates and instead include a reference to its signing certificate using a hash. In this case, it may be possible to construct the chain using certificates that are stored by the CME. If the security processing services encounter a certificate hash for which the corresponding certificate is not included in the protocol data unit (PDU), the services may invoke the CME-CertificateInfo.request primitive to determine whether that hash corresponds to a certificate that is already known to the CME. Figure D.19 illustrates the logic flow to be used when constructing a certificate chain. The processing cycles through the certificates that were received with the signed communications, and when no received certificate matches the current signer identifier, the processing invokes CME-CertificateInfo.request to attempt to continue building it. Processing that corresponds to this logic flow is presented in 7.8.2.

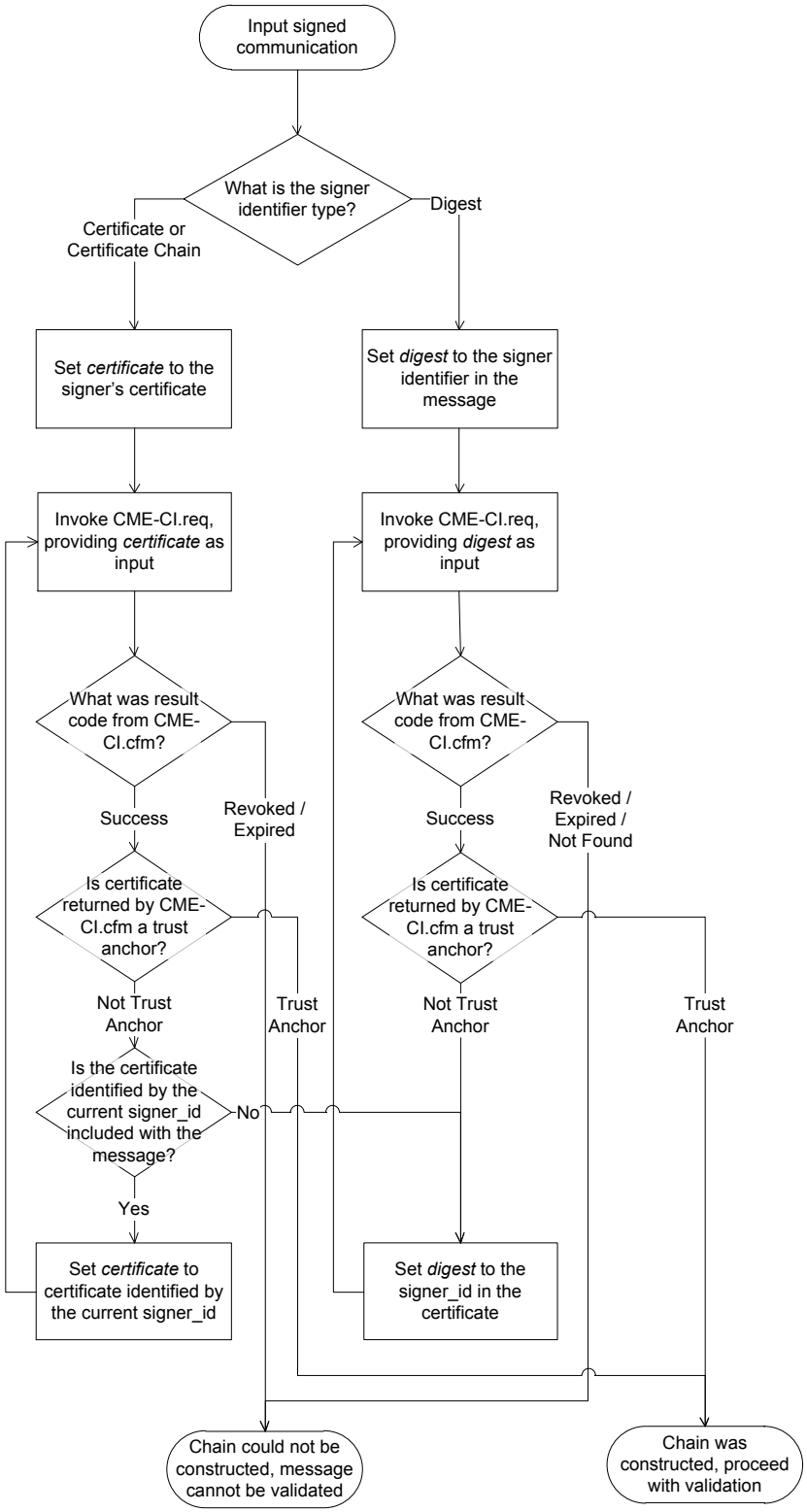


Figure D.19—Logic flow for constructing a certificate chain

Annex E

(informative)

Rationale and FAQ

E.1 Introduction

The Wireless Access in Vehicular Environments (WAVE) communication protocol stack is designed to enable communications in rapidly varying radio frequency (RF) environments, in particular vehicle-to-vehicle and vehicle-to-roadside communications. These communications provide transportation services such as alerting drivers to potential hazards, and notifying them of services of interest even at high speed or in high traffic density. This standard addresses the security requirements of the system. This annex provides informative material to help the understanding of the normative material in the main body of this standard.

E.2 General philosophy

E.2.1 What are the requirements and constraints on the system as a whole?

The WAVE system is intended to support a range of applications, including low-latency safety-of-life communications involving vehicles and personal WAVE devices; more general mobility applications designed to improve traffic flow; and e-commerce applications that use the 5.9 GHz airlink as the last hop to the WAVE device. Safety-of-life messages are time-critical, and as such, their timely and successful delivery is a major goal of the system. Information in the system will also influence driving decisions, and as such, the system needs to provide assurance that those messages are correct, relevant, and issued by the WAVE devices that claim to issue them. The security services described in this document are designed to increase bandwidth and processing time by the minimum amount consistent with meeting the security requirements.

E.2.2 What are the threat model and security goals for this standard?

This standard is designed to provide to provide authentication, authorization, integrity, and confidentiality services to WAVE users. These services should be industry-strength, standards-based, and bandwidth-optimized.

This standard does not have a formal threat model, but a good example of a threat model for a WAVE-like setting is given in the SeVeCom security requirements document [B23].

E.2.3 Why does this standard define new security mechanisms rather than reusing existing ones?

This standard defines formats and processing for secured data and certificates. In the Internet context, there are already well-established secure message and certificate formats (for example [B13]). In general, re-using established security designs is good practice, as established designs have withstood the test of time and reflect input from a wide range of contributors regarding the security requirements and specific threats. However, the Internet security mechanisms are designed for flexibility and extensibility, not to optimize bandwidth and processing time. Since bandwidth is more constrained over the 5.9 GHz airlink than over

the Internet, the designers of this standard chose to define size-optimized custom security mechanisms rather than reuse the Internet security mechanisms.

E.2.4 Where in the stack do WAVE Security Services live?

The security services defined in this standard are not at a particular location in the stack. Instead, these services may in principle be called by any functional entity on a WAVE device. See Clause 4 for an overview of the entities and interfaces in the security subsystem.

E.2.5 What security assumptions are made about the interface to the security services?

The primitives specified in this standard assume that the calling entity is trustworthy and correctly implemented. This means that certain inputs, or properties of inputs, do not have their correctness explicitly ensured by the security processing specified in this standard. For example:

— The security services do not check that an externally provided time corresponds to any local measure of time

— The security services specified in this document do not include mechanisms to ensure that a calling entity is entitled to use any Cryptomaterial Handle (CMH) that it requests the use of.

— The security services do not include mechanisms to determine that revocation or root certificate information is accurate.

Implementations should ensure that all data provided to the security services is valid and that users of keys are authorized to use those keys.

E.2.6 Are WAVE Security Services suitable for all applications?

WAVE Security Services are best used for Wave Service Advertisements (WSA), for broadcast applications, and for unicast applications with a small number of exchanges. If there will be a large number of exchanges, the processing overhead due to performing public key operations on each exchange may be excessive. If an application needs to communicate with a server over the Internet, the application may need to support a standard Internet security protocol. This may be layered over the use of IPv6 as specified in the other 1609 standards. It is anticipated that future versions of this standard will include protocols optimized for multiple settings.

E.2.7 What is the lifecycle of a device that uses security?

The lifecycle of a device that uses security is illustrated in Figure E.1. It includes the following steps:

a) **Manufacture**: The device is physically created.

b) **Acquisition**: The device is acquired by an operator and provided with system parameters common to all of that operator's devices.

c) **Configuration**: The device is configured for a specific (set of) application(s).

d) **Deployment**: The device is placed in its operating location, and possibly undergoes additional configuration appropriate to its location.

e) **Operation**: The device is in operation. This includes the following steps, serially or in parallel:

   1) **Core operations**: The application operations.

2) **Maintenance**: Minor reconfiguration and logging

3) **Administration**: Major reconfiguration and logging.

f) **Redeployment**: The device is moved to a different location.

g) **Reconfiguration**: The device is configured to support different applications or uses.

h) **Resale**: The device is transferred to a different operator.

i) **Decommissioning**: The device is physically end-of-lifed.

The lifecycle process includes managing both long-lived and short-lived security parameters:

— **Long-lived security parameters**:These include: root certificates; optionally, a key that is authorized to update the list of trusted root certificates; contact information for CAs that will issue device certificates; keying or other material to be used to prove the device is valid.

— **Short-lived security parameters**: These include: Certificates and keys that are needed by applications that send signed protocol data units (PDU); enrolment certificates; certificate revocation lists (CRL), and other revocation information.

Security parameters with different lifetimes will naturally be installed or updated at different lifecycle stages.

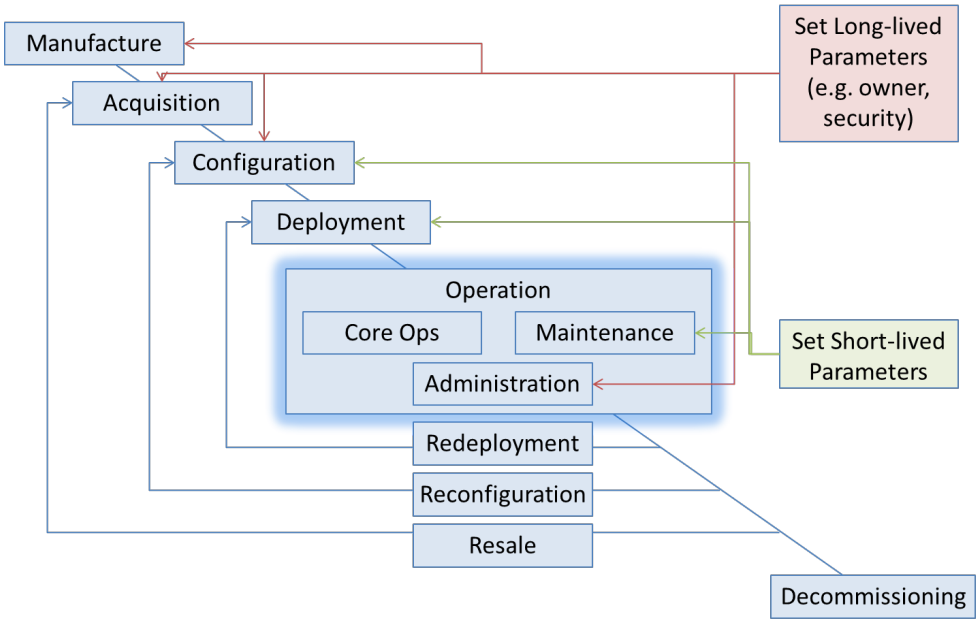


Figure E.1 — Lifecycle of a device that uses security services

E.2.8 Where do PSIDs come from?

At the time of writing, provider service identifiers (PSID) are assigned by the IEEE 1609 Working Group and are specified in IEEE Std 1609.12. Organizations that wish to reserve a PSID may request the 1609 Working Group to include the PSID assignment in an amendment to 1609.12. This process for assigning PSIDs may change.

E.3 System assumptions made in this standard

E.3.1 What are the requirements for accurate time in this standard?

A device may have multiple clocks: for example, the clock that governs channel intervals for wireless communication may be different from the clock that applications have access to. This standard assumes that the security services have access to a probability distribution function (PDF) for the current time. This standard does not impose a requirement for the accuracy of the PDF, but specifications of individual applications or services may make such a requirement.

E.3.2 How should applications respond if the time inaccuracy increases?

Different applications (or other secure communications entities) on a given WAVE device may have different needs for accurate time in both sent and received data, so the answer to the question depends on the specific application requirements.

For some applications, such as the Basic Safety Message of SAE J2735 [B20], the whole point of the application is to deliver rapidly-changing data. An error in the time as measured by the sender may result in the application building up an inaccurate picture of the world; on the other hand, since there is very little variation in propagation times, a receiver may to a first approximation set the generation time equal to the time received and recover from errors that way. Both a sender and a receiver may use this method to overcome errors in application information.

For applications that transmit ongoing information, a receiver may handle any level of inaccuracy so long as it is clear whether or not the communication is being received between its generation time and its expiry time.

If it is unclear whether the communication is within its validity period, the correct approach depends on whether false positives or false negatives are worse (i.e., is it better to include potentially out-of-date information or exclude it?). This decision varies not only from application to application but potentially from one type of message to another within an application. If an application favors false positives, it increases the validity period measures (*Validity Period*, *Acceptable Future Data Period*) or decreases the rejection thresholds (*Rejection Threshold for Too Old Data*, *Rejection Threshold for Future Data*, *Rejection Threshold for Expired Data* – a decreased threshold leads to a decreased probability of rejection). To exclude false positives, the application decreases the validity period measures and/or the rejection thresholds.

E.3.3 How can a WAVE device get accurate time?

IEEE Std 1609.3 defines WAVE Timing Advertisements to distribute timing information. WAVE devices may also use other mechanisms defined elsewhere. WAVE devices that perform channel switching are required by IEEE 1609.4 to have access to accurate TAI time. This standard's main concern is that a WAVE device measures its uncertainty accurately, not so much that the time itself is accurate.

E.3.4 How can a WAVE device get its location?

WAVE devices may have access to global positioning system (GPS) location. WAVE devices may be fixed and have their location hard-wired and available to applications. Information about location may also be obtained by other means.

E.3.5 Why are there two primitives for processing incoming signed data?

Incoming signed data are processed first via Sec-SecureDataContentExtraction.request, then via Sec-SignedDataVerification.request. This serves two purposes.

First, it allows implementations to avoid duplication of data. For example, if an application payload contains an expiry time, then this expiry time need not be contained in the security headers of the PDU. The receiver may extract the application payload from the signed data via Sec-SecureDataContent-Extraction.request, extract the expiry time from the application payload, and pass that time to Sec-Signed-DataVerification.request.

Second, it allows implementations to avoid the expensive processing associated with constructing a certificate chain and validating digital signatures. Sec-SecureDataContentExtraction.request allows the application to obtain all of the information contained in the PDU. If the application determines that the data is not relevant, it can discard the data without needing to invoke Sec-SignedDataVerification.request, saving valuable processing time.

E.3.6 How are keys stored?

This standard provides an interface that protects private keys from being revealed. An implementation should protect private keys from being trivially revealed to an attacker (for example, private keys should not be stored unencrypted on disk) and from being used by processes that do not have a right to use them.

E.4 Cryptography

E.4.1 Why sign data instead of using a message authentication code?

Digital signatures, and public key-based key management, are the preferred solution for systems that have a large number of members, where membership changes rapidly, where permissions may need to be transferred from one granter to another, and where nodes either do not have access to an online keyserver or do not wish to incur the latency associated with making a query to such a keyserver. Therefore, it is appropriate for use cases such as Basic Safety Message (BSM) to use digital signatures as defined in this standard.

Nothing in this standard requires that WAVE Security Services are used by any particular application (except that 1609.3 requires the use of IEEE 1609.2 mechanisms to secure WSAs). WAVE Security Services are simply security services that applications might choose to use to secure data. It is best suited for broadcast (one-to-many) applications. Applications with different communications patterns are free to use other security mechanisms. For example, 1609.11 defines symmetric crypto-based security for tolling.

E.4.2 Why ECDSA?

Elliptic curve digital signature algorithm (ECDSA) is standardized in IEEE Std 1363-2000 and accepted as an algorithm that may be used by devices conforming to FIPS 140 [B7]. Of all standardized algorithms, it provides the smallest keys and signatures. The working group chose ECDSA primarily because of these small keys and signatures.

E.4.3 Why ECIES?

Elliptic curve integrating encryption scheme (ECIES) is standardized in IEEE Std 1363-2000. It is based on elliptic curve cryptography, allowing the majority of the code for an implementation to be shared with an ECDSA implementation. It is suited for low-latency environments, as a sender who knows the receiver's public key may send encrypted data without any additional exchanges. Like ECDSA, ECIES has small keys and ciphertexts relative to other public key encryption systems.

E.4.4 Why ECQV (implicit) certificates?

ECQV implicit certificates, described in [B4], [B20], allow a system to transport public keys that are certified by a CA without explicitly including a CA signature. This saves 64 bytes for each certificate sent. For applications that send a large number of signed data, the overhead saving is significant.

The security of ECQV certificates with ECDSA is analyzed in [B5]. ECQV is at the time of writing undergoing standardization with ANSI ASC X9 and the industry consortium, the Standards for Efficient Cryptography Group (SECG).

E.4.5 Why are certificate hashes always calculated with the compressed form of the public key?

The compressed form of the public key is very cheap to calculate, while point decompression may be expensive. If the certificate hash (and therefore signature) is based on the compressed public key, the certificate holder may choose to send the compressed form of the public key to reduce packet size, or the uncompressed form to save processing on the recipient side.

E.4.6 Why AES-CCM?

AES-CCM is used in other wireless networking standards such as IEEE Std 802.11 [B10] and (with minor modifications) IEEE Std 802.15.4 [B11]. It is accepted by NIST as a FIPS 140 algorithm [B7]. Although other algorithms have performance advantages, the working group felt that these advantages were not significant and the benefit of using standardized cryptographic techniques was great.

E.4.7 How would an implementation choose implicit or explicit certificates? Fast verification with compression, fast verification without compression, or not fast verification? Compressed or uncompressed points? ECDSA over 224-bit or 256-bit fields for end-entities?

An implementation may choose different settings depending on how it weights considerations such as packet size, processing time, and the avoidance of patented techniques. The following are some sets of choices that an implementation might make:

— Minimize security overhead on a congested channel[32]: implicit certificates, fast verification with compression or no fast verification, compressed points, ECDSA-256.

— Minimize processing time when verifying occasional PDUs from multiple different senders[33]: implicit certificates, fast verification with compression or no fast verification, uncompressed points, ECDSA-256.

[32] In this case, it is important to attach a certificate to every message as individual packets might be lost.
[33] In this case, the certificate will need to be verified with a large proportion of messages that are verified.

— Minimize processing time when verifying multiple PDUs from the same sender: Explicit certificates, fast verification with no compression, uncompressed points, ECDSA-224

— Use only non-patented methods: explicit certificates, no fast verification, uncompressed points, ECDSA-224 or ECDSA-256.

For the SAE BSM (see profile in C.1), channel congestion is a significant concern, so the IEEE 1609.2 profile follows the approach in the first bullet above. For WSAs (see profile in C.2), the same set of options are used as for BSMs to reduce implementation complexity in an implementation that uses both BSM and WSA.

NOTE—The reason why the signing algorithm is ECDSA-256 for implicit certificates is that the mathematics of implicit certificates require the end-entity and the CA to use the same elliptic curve group, and CAs are required by 6.3.3 to always use ECDSA-256.

E.4.8 How would an implementation choose which SignerIdentifier type to use?

The sender should use a certificate if they consider that there may be recipients who: 1) Need to react to the data before the sender sends another datagram, and 2) Do not already have the sender's signing certificate, but 3) Do have all other certificates needed to verify the data (for example, if the signing certificate is directly signed by the root CA).

The sender should use a certificate digest if they consider that all recipients who may need to react to the data quickly already have the certificate.

The sender should use a certificate chain if they consider that there may be recipients who: 1) Need to react to the data before the sender sends another datagram, 2) Do not already have the sender's signing certificate, and 3) Do not have all other certificates needed to verify the data (for example, if the signing certificate is not directly signed by the root CA but by a CA that is not widely used).

The signer identifier type used should depend on a number of factors including:

— Channel congestion (with higher congestion suggesting shorter signer identifiers, such as the digest type).

— The time that a receiver will have to react (time-critical messages will need to be verified immediately, so the sender should send a full certificate chain more often). This will include considerations of the transmit range of the sender.

— How quickly the population of receivers changes (a population of receivers needs the full certificate chain only once, so a slow-changing population implies that the sender need only send the chain infrequently, while a fast-changing population implies that the sender should send the full chain frequently)

E.4.9 As noted in SEC 4, the public key may be derived from an implicit certificate separately from the use of that certificate to verify. The primitives in this standard do not support using implicit certificates that way. Why not?

Primitives are not normative or testable; only the results of processing are normative and testable. Any implementation that produces equivalent results to the primitives in this standard is acceptable. Implementers may choose to calculate the public key from an implicit certificate separately from performing the verification operation so long as the outputs of the processing are the same as the outputs from processing that uses the implicit certificate directly for verification.

NOTE—If a receiver only verifies a single PDU from a sender, it is faster to perform verification using the implicit certificate directly. If a receiver verifies multiple PDUs from the same sender, it is faster to calculate the public key and verify using that.

E.4.10 What are the requirements for randomness?

This standard does not give any testable requirements for the quality of the random number generator. However, implementers should be aware that poor quality randomness (i.e., guessable randomness, or data drawn from a distribution with a known bias) is a well-known source of weakness in cryptographic systems. Attacks based on poor quality randomness include: leakage of keys used for Secure Sockets Layer (SSL) sessions in early versions of Netscape [B8]; compromise of ECDSA keys through information leaked by poor-quality randomness when signing [B9] [B19]; and a bug in Debian Linux that caused OpenSSL keys to be selected from a set of size only 32768 ([B3], section "Centralising Randomness"; [B24]). The reader is referred to ANSI X9.82 [B1] for a more detailed discussion and suggested designs that provide high-quality (i.e., highly unguessable) randomness. In particular, private keys and ephemeral private keys used by ECDSA should be obtained at least from a strong pseudo-random number generator as described in 5.8.3.

E.5 Secure data exchange

E.5.1 How might the service specific permissions (SSP) be used?

The PSID is used in the 1609 WSMP protocol to denote the higher layer entity that is intended to receive a WAVE Short Message. However, it is conceivable that the sending entity is only entitled to issue a subset of the messages that the receiving entity can understand. For example, the Basic Safety Message of SAE J2735 [B20] includes a field stating whether or not the sending vehicle's light bar is on. If a vehicle's light bar is on it is requesting that other vehicles give it priority. Clearly, all receivers need to understand this field, but not all vehicles on the road should be able to create a message with this field set.

This ability to specify senders' permissions narrowly and receivers' interests broadly can be implemented in two ways in the PSID system: either there are a large number of PSIDs representing different sets of sender permissions, and a receiver registers to receive on each of those PSIDs, or there are a smaller number of PSIDs and fine-grained permissions are carried in a different field. The working group considered that a system based on a large number of PSIDs complicated the use of PSID within WSMP. Additionally, a system based on PSIDs only would create problems for legacy applications, as those applications would not know of new PSIDs that represented different sets of permissions on existing messages. The working group therefore decided to use a system based on PSIDs + SSPs.

E.5.2 Who defines the SSP?

The body or entity that reserves a PSID defines conformant use of that PSID, including whether an SSP is used with that PSID and, if so, a description of the syntax of the SSP.

The WAVE Security Services do not expect to be able to parse the SSP and this is not functionality that the working group anticipates adding to this standard.

E.5.3 What is the purpose of the "unknown" and similar cases in data structures?

A number of data structures have contents that depend on some enumerated value or flags field. For example, the contents of the 1609Dot2Data depend on the `type` field. Future implementers may wish to define additional values for these fields, resulting in different structures. The use of the "unknown" value

allows implementers to define these values and the associated contents without changing the version number. This allows implementations with an older implementation of this standard to parse these new structures, even if the implementations cannot understand the new contents themselves. This is valuable if, for example, an implementation receives two concatenated 1609Dot2Data, or a PDU of a different type with a 1609Dot2Data embedded within it, as the implementation may extract all of the information even if the 1609Dot2Data contains unknown fields. This approach also allows the standard to adopt new structures as and when they are useful without having to be concerned about incrementing the version number for potentially minor changes.

E.5.4 Why invoke CME-Sec-AddCertficateInfo from within Sec-SecureDataContent-Extraction.request? In other words, why store an unverified certificate?

The purpose of storing an unverified certificate is to allow the following process flow: 1) A WAVE device receives data signed with a certificate, and after invoking Sec-SecureDataContentExtraction.request decides not to invoke Sec-SignedDataVerification.request; 2) The WAVE device receives data from the same source signed with a digest, and invokes Sec-SignedDataVerification.request. If the WAVE device could not store the certificate at step 1, it would be unable to verify at step 2 as the structure in step 2 does not explicitly contain the certificate.

E.5.5 Signed data becomes invalid once the signing certificate expires. How can a sender create signed data that lasts past the expiry time of the signing certificate?

The sender can re-sign the data with the new certificate, if available.

E.5.6 Why is signed data not automatically considered invalid if its signing certificate has expired?

Because the receiver may be legitimately asking the question, “was this data valid when it was signed?” rather than “is this data valid now?”

Data that was not valid when it was signed is not valid under any circumstances, so the security processing services always check for validity at time of signing. However, data that has expired may be of interest to an application, for example if it uses historic patterns of behavior for its own internal purposes.

An application that wishes to reject expired data can request the security services to check expiry using the relevance check parameters to Sec-SignedDataVerification.request. Validly signed data cannot have an expiry date later than the expiry date of its signing certificate, so if a certificate has expired, valid data will also have expired, and this can be detected by the relevance checks.

E.5.7 If I have to encrypt to a certificate that’s currently valid, how can I encrypt to someone so they can decrypt in the future after the current certificate expires?

The decryption operations do not require that the decryption key is currently valid, so this isn’t a problem.

E.5.8 How should a replay database be managed?

If an application is potentially vulnerable to replay attacks, it should maintain a replay database and an expiry criteria associated with that replay database such that any data that meets the expiry criteria is removed from the database. If this is the case, any received data that meets the expiry criteria will

automatically be rejected. If the expiry criteria vary over time, any newly received data that would at any time have met the expiry criteria will be automatically be rejected.

E.6 Signed WSAs

E.6.1 Why would a higher layer entity request signed WSAs? Why would a user service require them?

Signing a WSA provides authentication that the sending Wave Management Entity (WME) is authorized to advertise the signed services. It does not provide a secure (encrypted, authenticated) connection between the User and the Provider. This must be provided by the higher layer entities themselves.

A higher layer entity that registers for a provider service should request that its WSAs are signed if:

— The deployer considers there is a risk to the privacy of responders (i.e., if a service is sufficiently *rarely* used, the fact that a given User responds to the service can be used to distinguish or track the user).[34]

— The deployer considers that an unauthenticated service could cause a force-multiplied denial of service attack (i.e., that the service is sufficiently *widely* used that, if it is advertised in an area of dense network traffic, so many WAVE devices will respond as to cause significant channel congestion).

A higher layer entity that registers for a user service should require valid signed WSAs if the deployer considers there is a risk to the privacy of responders. Conversely, a higher layer entity that registers for a provider or user service may choose not to require signed WSAs if there is no requirement for authentication or if the privacy of the user service is not compromised by responding to the WSA.

E.6.2 Why are the Channel Info and the WAVE Routing Advertisement (WRA) not included in the certificate?

The Channel Info and WRA are parts of the WSA as defined in IEEE Std 1609.3. They are included in the data that is signed, so the statement that is being made is that if a WAVE device is authorized to advertise any specific Channel Info and/or WRA, it is authorized to advertise all possible Channel Infos and WRAs. The working group did not see any threat that would be mitigated by including more granular permissions on the Channel Info and WRA for an already-trusted WAVE device.

[34] Consider a vehicular or personal WAVE device that hosts a rare user service, meaning a service that has only a relatively small number of users out of the entire population of WAVE devices. (For example, there might be a special service offering added-value information to Rotary Club members.) If this service was offered in almost every location, then since only a few WAVE devices will respond to the service, an eavesdropper will with high probability be able to track those devices that respond to it. If the eavesdropper can broadcast bogus WSAs advertising that service, the eavesdropper can in principle track the users of that service anywhere. There are several mitigations for this; the one in scope for the 1609 standards is to allow user services to require that WSAs are signed before responding.

Other possible responses include the following. First, since a large number of WSA-sending locations would be needed for tracking, the attacker would be easy to detect. Second, WAVE devices could be programmed to emit bogus responses to services from time to time, to confuse any attack based on traffic analysis.

E.6.3 Why might a secure provider service use the SSP in a WSA? What's the difference between an SSP and a Provider Service Context (PSC)?

The SSP allows the WSA to make authenticated statements about the status or authoritativeness of the services advertised in that WSA. For example, a WSA might offer two different traffic advisory services, one from a government source and one from a private source. A user that only wants information from government sources could choose to connect to the government service and not the private service.

The PSC may be used to encode transient information about the service being offered. For example, in the case of traffic advisory information, the PSC may include the last time that the information was updated. This allows a receiver to decide whether or not to connect to the service. The syntax of the PSC is defined by the organization that reserves the PSID. The SSP may constrain the permissible values of the PSC.

The receiving unit may make use of the SSP as follows:

— On receipt of a WSA, the receiving WME writes the SSP for each secure provider service to the UserAvailableServiceSpecificPermissions parameter in the appropriate UserAvailableServiceTable field.

— User services on the user device may make use of this information in determining whether to use the advertised provider service.

E.6.4 How does a certificate management process know what WSA certificates to apply for? If there are 100 registered services, how would it decide which ones to group in a certificate?

This is implementation specific. Factors to consider in the implementation of certificate request include:

— Certificates with fewer entries result in smaller packets and better chance of successful transmission.

— Certificates with more entries have a greater chance of containing the full set of secure provider services currently requested for inclusion.

— The number of possible certificates is exponential in the number of secure provider services, so it may not be practical to request or store all possible certificates.

If a provider device may have to support a large number of secure provider services, the implementer should provide mechanisms that allow the certificate management process to decide what groups of services are likely to be advertised at the same time and so form a certificate request containing all of the services in the group

E.6.5 If a certificate is not available for the set of requested secure provider services, what should the WME do?

This is implementation specific. The WME may attempt to form a WSA with a subset of the requested services, or to send multiple WSAs that together advertise the full set of requested services. Implementations may provide a richer interface between the provider service security management entity (PSSME) and the WME to support making this decision.

E.6.6 Why use permission indices in a signed WSA?

Not all services in the WSA need appear in the WSA certificate, not all services in the WSA certificate need appear in the WSA, and a WSA may contain two services with the same PSID but different SSPs. Therefore, a means is needed to determine which entry in the certificate corresponds to a given entry in the WSA. The `permission_indices` field provides this means. For examples of the mapping between a set of ServiceInfos and a set of permissions using `permission_indices`, see D.4.

E.6.7 What opportunities are there to improve performance with signed WSAs?

To improve performance, an implementation of the security processing services may cache recently received WSAs and automatically accept retransmissions of previously received WSAs where appropriate. In this case, the following conditions should hold:

- The security processing services cache WSAs until a time no later than their expiry time.
- For each cached WSA, the security processing services cache:
  - The exact bytes of the WSA.
  - The values returned by WME-Sec-SignedWsaVerification.confirm when that WSA was verified: *Result Codes*, *WSA Data, Service Specific Permissions, Generation Time, Generation Time Standard Deviation, Expiry Time, Generation Latitude, Generation Longitude, Generation Elevation, Last Received CRL Times, Next Expected CRL Times, Sender Certificate*.
- The security processing services cache only the most recently received WSA associated with each WSA signing certificate.
- If a CRL is received, the security processing services ensure that no cached WSAs are affected by the CRL. This may be done in a number of ways, including but not limited to the following:
  - (Time consuming but may save processing time later): Check, for each signed WSA, that none of the certificates in its chain are affected by the CRL; or
  - (Fast but may increase processing time later): Delete all cached WSAs from memory.

When a WSA is received, the security services check whether an identical WSA is already held in memory.

- If an identical WSA (based on a byte-by-byte comparison of the two encoded 1609Dot2Data structures) already exists in memory, the security services set all of the parameters to "WME-Sec-SignedWsaVerification.confirm" according to the cached return values. In this case, the implementation may choose to indicate whether a valid WSA was a duplicate of a cached WSA.
- If no identical WSA exists, the security services should run the algorithm defined in 7.3.4.

E.6.8 What should the lifetime of a signed WSA be?

The threat for a signed WSA is that an attacker will record it and replay it after the set of services is no longer valid, potentially tricking a user into revealing information about itself by its response to that WSA. The security threat of a long lifetime is that this attack will be valid for the lifetime of the WSA. The performance advantage of a long lifetime is that it reduces the verification processing burden on receivers and the signing processing burden on senders. Given these considerations, an appropriate WSA lifetime is probably on the order of minutes rather than seconds or hours.

E.7 Certificate request

E.7.1 IEEEStd 1609.2 specifies application-layer messages for certificate management, but what transport protocol do WAVE devices use to communicate with the CA?

The transport protocol is not specified in this standard, but is a likely candidate for standardization in a future version of this standard.

E.7.2 How should communications with the CA be protected?

The following table gives an informal summary of how communications with the CA should be protected to support secure operations.

Certificate request model	Communications requirements
Cryptomaterial (certificates and private keys) are installed on the WAVE device. The secure communications entity that will use that cryptomaterial does not explicitly request certificates or generate keys.	The cryptomaterial is sent over an authenticated and encrypted session.
Private and public keys are installed on the WAVE device. The secure communications entity uses these to generate certificate requests	The private and public keys are sent over an authenticated and encrypted session. The certificate request may be sent over an insecure session.
Requests for communications certificates are self-signed using keys generated on the WAVE device.	The communications certificate request is sent or confirmed over an authenticated session.
Requests for enrolment certificates are self-signed using keys generated on the WAVE device. Requests for communications certificates are signed with enrolment certificates.	The enrolment certificate request is sent or confirmed over an authenticated session. The communications certificate request may be sent over an insecure session.
An initial set of requests for enrolment certificates are self-signed using keys generated on the WAVE device. Subsequent requests for enrolment certificates are signed with enrolment certificates that already exist. Requests for communications certificates are signed with enrolment certificates.	The initial set of requests for enrolment certificates are sent or confirmed over an authenticated session. Subsequent requests for enrolment certificates may be sent over an insecure session. Requests for communications certificates may be sent over an insecure session.

E.7.3 Is a CA obliged to issue a certificate in response to a request?

This standard does not put requirements on the behavior of the CA. In particular, a CA is not required to issue a certificate in response to a request, even a valid request, and if a CA responds to a request by issuing a certificate, it is not required to include the same fields in the certificate that were included in the request.

E.7.4 Why can a certificate request contain a subset of the permissions of the enrolment certificate, instead of the whole set of permissions?

Having multiple permissions in the enrolment certificate may improve efficiency at the CA, because the CA has to issue and manage fewer enrolment certificates. Having only a single permission in the communications certificate improves privacy, because it means that the holder of that certificate does not need to reveal their other permissions.

E.8 CRL use

E.8.1 Is there an efficient way to implement revocation checking?

A CRL allows an implementation to determine if a certificate is revoked. A certificate is revoked if it, or any issuing certificate in its chain, appears on a CRL The following steps may be used following the receipt of a valid CRL to correctly determine whether certificates that are already known to the implementation are revoked:

a) Mark all certificates, known to the CME, that appear on the CRL as "revoked".

b) For each certificate known to the CME, construct the certificate chain using CME-Function-ConstructCertificateChain. This automatically marks each certificate as revoked if the CA has been revoked.

E.8.2 If a certificate can't be revoked (because it has crl_series equal to 0), how can it be removed from the system if its holder is compromised?

A CA should issue certificates with `crl_series` equal to 0 only if those certificates have a short lifetime, so that the holder has to request new certificates frequently. If the holder is discovered to be compromised, the CA can refuse to issue a new certificate. The meaning of "frequently" is context-dependent.

E.9 Security mechanisms not included in this standard

E.9.1 Can I use security mechanisms other than the ones specified in this standard?

Application PDUs can be secured with any appropriate mechanism. The mechanisms in this standard are optimized for frequently-transmitting, broadcast applications. Other mechanisms may be suited for other applications. The organization that reserves the PSID or specifies the application may specify the use of WAVE Security Services or of any other security mechanisms that are consistent with the transport mechanisms in the rest of the 1609 standards.

IEEE Std 1609.3 requires that, if WSAs are secured, they are secured with the mechanisms of this standard.

E.9.2 What about SSL?

A deployer of an application that uses IP communications can choose to use Secure Sockets Layer (SSL), now known as Transport Layer Security (TLS) [B18], or its variant, Datagram Transport Layer Security (DTLS) [B15], to protect the communications between client and server. TLS is suited for reliable transport channels, and DTLS for datagram traffic. It is anticipated that future versions of this standard will specify a version of DTLS optimized for the WAVE setting, for example, by specifying how IEEE 1609.2 certificates may be used within DTLS.

E.9.3 What about mobile IP protocols?

An application may choose to use any mobile IP protocol to support mobile IP access. Previous field tests of WAVE technology have successfully tested the Vehicle Infrastructure Integration Host Identity Protocol (V-HIP), an anonymity-enhanced version of the internet Host Identity Protocol (HIP) [B16], [B17].

If a WAVE device's user expects privacy, the device should not emit in plaintext any string that is repeated across packets and (absolutely or almost) unique to the device. If a WAVE device regularly contacts a server at a particular IP address or other fixed URL, and if few other WAVE devices contact the same server, and if the address details are sent in the clear, then the server address may compromise the WAVE device user's privacy by allowing eavesdroppers to identify activity by that WAVE device. It is recommended that service providers ensure their service can be accessed in a way that takes this into account, in order to reduce the risk of privacy compromise.

E.9.4 Are there other security protocols for individual datagrams?

The Cryptographic Message Syntax [B14] is an IETF message syntax for securing individual datagrams.

E.9.5 Why does the standard not include a full specification of anonymous certificates?

Anonymity—meaning the ability of private drivers to maintain a certain amount of privacy—is a core goal of the system. The system should support the ability of non-compromised privately operated WAVE devices to keep their activities private not just from casual eavesdroppers but from inside attackers at the CA. However, this goal might conflict with other goals such as removing bad actors and supporting law enforcement access under appropriate circumstances. This means that IEEE 1609.2 certificates for anonymous communications cannot simply be identifying certificates with the identifying information left out: there has to be support for revocation and support for privacy against the authority that knows enough information to revoke.

Revocation and privacy are in conflict with each other, and the exact tradeoff between these goals is a policy matter, with the policy to be decided by stakeholders such as (in the U.S.) the vehicle OEMs and federal and state governments. These stakeholders have not yet communicated the specific requirements to the 1609 Working Group. The 1609 Working Group therefore decided not to include an anonymous certificate specification that might fail to meet the eventual set of requirements. An anonymous certificate specification will be addressed in a future version of or amendment to this standard.

E.9.6 Are there considerations for privacy and anonymity that are outside the scope of this standard?

Yes. This standard specifies formats and processing for PDUs to support cryptographic security, but providing full privacy requires attention from designers of all different parts of a WAVE device.

First, it is useful to discuss what privacy means. In informal terms, privacy means that the owner of information has control over who gets to see that information; information is only revealed with the owner's consent. Put like that, privacy sounds very similar to the standard cryptographic service of confidentiality, and privacy does in fact use confidentiality as a mechanism, but privacy has a broader scope and includes, in addition to confidentiality, consideration of broadcast messages (which cannot use confidentiality) and of traffic analysis.

This section is not intended to be a full treatment of privacy; there are many discussions of privacy in the literature, perhaps the most thorough of which at the time of writing is the EU Preciosa project [B21] and its many deliverables. The reader is referred to that for more details of the underlying principles. This section simply summarizes mechanisms that can be used to improve the privacy properties of deployed WAVE devices.

For transactional applications, privacy mechanisms include:

— Ensure that all transactions are encrypted.

— Ensure that no identifying information is exchanged before the start of encrypted communications.

— Ensure that applications respect privacy by revealing no more information than is necessary to carry out the transaction.

For broadcast applications, privacy mechanisms include:

— Ensure that that identifiers used by a WAVE device do not link to the WAVE device's real-world identity. (An "identifier" in this sense is any string in a personal WAVE device's PDUs that is not used by any other nearby personal WAVE device). For example, if a vehicle's PDUs include its licence plate number, this obviously allows the vehicle to be tracked by anyone who can receive those PDUs.

— Ensure that identifiers change frequently.

— Ensure that identifiers change at the same time. For example, for an application sending Basic Safety Message (BSM) of SAE J2735 [B20], the identifiers are the Temporary ID within the BSM payload (see SAE J2735 for more details), the IEEE 1609.2 certificate, and the source MAC address. These identifiers should be synchronized so that when any identifier changes, all identifiers change. IEEE Std 1609.4 [B12] provides primitives to allow the source MAC address to change.

For WAVE devices as a whole, developers should take steps to ensure that WAVE devices do not inadvertently reveal the full combination of applications that are running on the device. For example, an attacker may identify a WAVE device by identifying the combination of PSIDs registered on that WAVE device by different applications.

Annex F

(informative)

Copyright statement for 6.1

Much of the material in 6.1 is derived from [B18] and is used subject to the following copyright statement.

Full Copyright Statement

Annex G

(informative)

Bibliography

Bibliographical references are resources that provide additional or helpful material but do not need to be understood or used to implement this standard. Reference to these resources is made for informational use only.

[B1] ANSI X9.82-1:2006, Random Number Generation Part 1: Overview and Basic Principles, ANSI, 2006. Available from http://webstore.ansi.org/RecordDetail.aspx?sku=ANSI+X9.82-1%3a2006.

[B2] Antipa A., R. Gallant, and S. Vanstone, "Accelerated verification of ECDSA signatures," Selected Areas in Cryptography, 12th International Workshop, SAC 2005, Kingston, ON, Canada, August 11-12, 2005,: Springer, 2005, pp. 307-318.

[B3] Bernstein D. J., T., and P. Schwabe. "The security impact of a new cryptographic library." Available from http://cr.yp.to/papers.html#coolnacl.

[B4] Brown D., R. Gallant, and S. Vanstone, "Provably secure implicit certificate schemes," *Financial Cryptography*, Springer, 2002, pp. 156-165.

[B5] Brown D. R. L., M. J. Campagna, and S.A. Vanstone, "Security of ECQV-Certified ECDSA Against Passive Adversaries," 2009, pp. 1-15.

[B6] ETSI TS 102 731 V1.1.1 (2010-09), Technical Specification: Intelligent Transport Systems (ITS); Security; Security Services and Architecture.

[B7] FIPS Pub 140-2, 'Security requirements for Cryptographic Modules,' Federal Information Processing Standards Publication 140-2, US Department of Commerce/N.I.S.T., Springfield, Virginia, June 2001 (supersedes FIPS Pub 140-1).

[B8] Goldberg I. and D. Wagner, "Randomness and the Netscape Browser: How Secure is the World Wide Web?", Doctor Dobb's Journal, January 1996. Available from http://www.cs.berkeley.edu/~daw/papers/ddj-netscape.html.

[B9] Howgrave-Graham N. and N. Smart, "Lattice Attacks on Digital Signature Schemes," Designs, Codes and Cryptography Vol 23, p 283-29, Springer, 1999. Url: http://dx.doi.org/10.1023/A:1011214926272. DOI: 10.1023/A:1011214926272.

[B10] IEEE Std 802.11™, Information technology—Telecommunications and information exchange between systems—Local and metropolitan area networks—Specific requirements—Part 11: Wireless LAN Medium Access Control (MAC) and Physical Layer (PHY) specifications.

[B11] IEEE Std 802.15.4™ -2011, IEEE Standard for Local and metropolitan area networks— Part 15.4: Low-Rate Wireless Personal Area Networks (LR-WPANs).

[B12] IEEE Std 1609.4™-2010, IEEE Standard for Wireless Access in Vehicular Environments (WAVE) –Multi-Channel Operation.

[B13] IETF Request for Comments: 3280, Internet X.509 Public Key Infrastructure Certificate and Certificate Revocation List (CRL) Profile.

[B14] IETF Request for Comments: 3852, Cryptographic Message Syntax (CMS).

[B15] IETF Request for Comments: 4347, Datagram Transport Layer Security (DTLS).

[B16] IETF Request for Comments: 4423, Host Identity Protocol (HIP) Architecture.

[B17] IETF Request for Comments: 5201, Host Identity Protocol.

[B18] IETF Request for Comments: 5246, The Transport Layer Security (TLS) Protocol Version 1.2.

[B19] Nguyen P. and I. Shparlinski, "The Insecurity of the Elliptic Curve Digital Signature Algorithm with Partially Known Nonces," Designs, Codes and Cryptography Vol 30 Issue 2, p 201-217, Springer, 2003. Url: http://dx.doi.org/10.1023/A:1025436905711. Doi: 10.1023/A:1025436905711.

[B20] Pintsov L. A. and S.A. Vanstone, "Postal revenue collection in the digital age," 2000, pp. 105-120.

[B21] Preciosa Project, PRECIOSA - Privacy Enabled Capability In Co-operative Systems and Safety Applications. Available from http://www.preciosa-project.org/ [Accessed July 11, 2011].

[B22] SAE J2735, Dedicated Short Range Communications (DSRC) Message Set Dictionary. Available from http://store.sae.org.

[B23] Sevecom – Secure Vehicle Communication, Deliverable 1.1, VANETS Security Requirements Final Version. Available from http://www.sevecom.org/Deliverables/Sevecom_Deliverable_D2.1_v3.0.pdf

[B24] Software in the Public Interest, Inc. Debian security advisory, DSA-1571-1openssl|predictable random number generator, 2008. Available from http://www.debian.org/security/2008/dsa-1571.