Document Identifier: DSP2058

Date: 2024-06-05

Version: 1.3.0

Security Protocol and Data Model (SPDM) Architecture White Paper

Supersedes: 1.2.0

Document Class: Informational

Document Status: Published

Document Language: en-US

9 DMTF is a not-for-profit association of industry members dedicated to promoting enterprise and systems management and interoperability. Members and non-members may reproduce DMTF specifications and documents, provided that correct attribution is given. As DMTF specifications may be revised from time to time, the particular version and release date should always be noted.

10 Implementation of certain elements of this standard or proposed standard may be subject to third-party patent rights, including provisional patent rights (herein “patent rights”). DMTF makes no representations to users of the standard as to the existence of such rights, and is not responsible to recognize, disclose, or identify any or all such third-party patent right owners or claimants, nor for any incomplete or inaccurate identification or disclosure of such rights, owners, or claimants. DMTF shall have no liability to any party, in any manner or circumstance, under any legal theory whatsoever, for failure to recognize, disclose, or identify any such third-party patent rights, or for such party’s reliance on the standard or incorporation thereof in its product, protocols, or testing procedures. DMTF shall have no liability to any party implementing such standard, whether such implementation is foreseeable or not, nor to any patent owner or claimant, and shall have no liability or responsibility for costs or losses incurred if a standard is withdrawn or modified after publication, and shall be indemnified and held harmless by any party implementing the standard from any and all claims of infringement by a patent owner for such implementations.

11 For information about patents held by third parties which have notified DMTF that, in their opinion, such patents may relate to or impact implementations of DMTF standards, visit https://www.dmtf.org/about/policies/disclosures.

12 This document’s normative language is English. Translation into other languages is permitted.

13 CONTENTS

14

1 Abstract

15 This white paper presents an overview of the SPDM architecture, its goals, and a high-level summary of its use within a larger solution. The intended target audience for this white paper includes readers interested in understanding the use of SPDM to facilitate security of the communications among components of platform management subsystems.

16 > **Note:** This white paper refers to this architecture as the Security Protocol and Data Model (SPDM) architecture or SPDM.

17 The SPDM architecture focuses on securing platforms against attacks facilitated by components of the platform. To enable this defense, the SPDM architecture enable components to prove their identity and integrity, and to exchange keys for secure communication. The SPDM architecture complements other standards from DMTF, including the Redfish and PMCI standards, as well as standards from alliance partner organizations.

18 This white paper is not a replacement for the individual SPDM specifications but provides an overview of how the specifications operate within a larger solution.

2 Foreword

The Security Protocols and Data Models (SPDM) Working Group of the DMTF prepared the *Security Protocol and Data Model (SPDM) Architecture White Paper* (DSP2058).

DMTF is a not-for-profit association of industry members that promotes enterprise and systems management and interoperability. For information about the DMTF, see DMTF.

This version supersedes version 1.2 and its errata versions. For a list of the changes, see ANNEX A (informative) change log.

2.1 Other important resources

The SPDM Working Group defines standards to enable security for platforms, whether for the control plane, data plane, or other infrastructure. The SPDM Working Group also provides a sample implementation of the SPDM called libspdm through its SPDM Code Task Force.

The SPDM Working Group maintains the SPDM Forum to facilitate discussion of the SPDM specifications. Public discussion is welcome on the SPDM Forum, and the SPDM Working Group is active in reviewing and responding to forum posts.

2.2 Acknowledgments

The authors want to acknowledge the following people:

Editors:

- Brett Henning — Broadcom Inc.
- Theo Koulouris — Hewlett Packard Enterprise
- Raghupathy Krishnamurthy — NVIDIA Corporation
- Masoud Manoo — Lenovo
- Viswanath Ponnuru — Dell Technologies

Contributors:

- Richelle Ahlvers — Broadcom Inc.
- Jeff Andersen — Google
- Lee Ballard — Dell Technologies
- Steven Bellock — NVIDIA Corporation
- Heng Cai — Alibaba Group
- Patrick Caporale — Lenovo
- Yu-Yuan Chen — Intel Corporation

- Andrew Draper — Intel Corporation
- Nigel Edwards — Hewlett Packard Enterprise
- Daniil Egranov — Arm Limited
- Philip Hawkes — Qualcomm Inc.
- Jeff Hilland — Hewlett Packard Enterprise
- Yi Hou — Microchip
- Guerney Hunt — IBM
- Yuval Itkin — NVIDIA Corporation
- Benjamin Lei — Lenovo
- Luis Luciani — Hewlett Packard Enterprise
- Donald Matthews — Advanced Micro Devices, Inc.
- Mahesh Natu — Intel Corporation
- Chandra Nelogal — Dell Technologies
- Edward Newman — Hewlett Packard Enterprise
- Alexander Novitskiy — Intel Corporation
- Jim Panian — Qualcomm Inc.
- Scott Phuong — Cisco Systems Inc.
- Jeffrey Plank — Microchip
- Xiaoyu Ruan — Intel Corporation
- Nitin Sarangdhar — Intel Corporation
- Vidya Satyamsetti — Google
- Hemal Shah — Broadcom Inc.
- Srikanth Varadarajan — Intel Corporation
- Peng Xiao — Alibaba Group
- Qing Yang — Alibaba Group
- Jiewen Yao — Intel Corporation

30 # 3 References

31 The following referenced documents are indispensable for the application of this white paper. For dated or versioned references, only the edition cited (including any corrigenda or DMTF update versions) applies. For references without a date or version, the latest published edition of the referenced document, including any corrigenda or DMTF update versions, applies.

- DMTF DSP0236, *MCTP Base Specification 1.3.0*
- DMTF DSP0239, *Management Component Transport Protocol (MCTP) IDs and Codes*
- DMTF DSP0274, *Security Protocol and Data Model (SPDM) Specification 1.3.0*
- DMTF DSP0275, *Security Protocol and Data Model (SPDM) over MCTP Binding Specification 1.0.1*
- DMTF DSP0276, *Secured Messages using SPDM over MCTP Binding Specification 1.1.0*
- DMTF DSP0277, *Secured Messages using SPDM Specification 1.1.0*
- DMTF DSP2015, *Platform Management Components Intercommunication (PMCI) Architecture White Paper 2.0.0*
- IETF TLS DTLS13-43, *The Datagram Transport Layer Security (DTLS) Protocol Version 1.3 draft-ietf-tls-dtls13-43*, 30 April 2021
- RFC5280, *Internet X.509 Public Key Infrastructure Certificate and Certificate Revocation List (CRL) Profile*
- NIST SP 800-57, *NIST SP 800-57 Part 1 Rev. 4, Recommendation for Key Management, Part 1: General*
- NIST SP 800-90, *NIST SP 800-90A Rev. 1, Recommendation for Random Number Generation Using Deterministic Random Bit Generators*
- NIST SP 800-193, *NIST SP 800-193, Platform Firmware Resiliency Guidelines*
- *USB Authentication Specification Rev 1.0 with ECN and Errata through January 7, 2019*

32

4 Terms and definitions

33 This white paper uses terms that the following specifications define:

- *Security Protocol and Data Model (SPDM) Specification 1.3.0*
- *Security Protocol and Data Model (SPDM) over MCTP Binding Specification 1.0.1*
- *Secured Messages using SPDM over MCTP Binding Specification 1.1.0*
- *Secured Messages using SPDM Specification 1.1.0*

34 # 5 Introduction

35 ## 5.1 Typographical conventions

- Document titles are marked in *italics*.
- Important terms that are used for the first time are marked in *italics*.

36 ## 5.2 Authentication

37 Enterprise computer platforms include many components that contain mutable elements. Each mutable component presents a potential vector for attack against the component itself, or even the use of a component to attack another component in the computer. To defend against these attacks, the Security Protocol and Data Model (SPDM) Specification enables conformant implementations to challenge a component to prove its identity and the correctness of its mutable component configuration.

38 An SPDM-conformant component generates, or is provisioned with, an asymmetric device public/private key pair. The component uses the device private key to sign requests, which proves knowledge of the private key. The Requester uses the device public key to authenticate the component-generated signature. For more details about the message exchanges, see Message details.

39 An SPDM-conformant component that is acting as a Responder can also perform authentication of the Requester, which is *mutual authentication*. By performing mutual authentication, the Responder can establish two-way trust with the Requester so that the two parties can establish a session.

40 ## 5.3 Authenticated Encryption with Associated Data (AEAD)

41 SPDM-conformant components can establish an Authenticated Encryption with Associated Data (AEAD) session. When a Requester and Responder have established an AEAD session, the Requester and Responder establish shared keys that are used to protect communication between the two endpoints. The keys can be used for authenticated communication, or for authenticated and encrypted communication.

42 Components can establish a session to protect messages from unauthorized alteration (authenticated communication) or to protect messages from unauthorized observation and alteration (authenticated and encrypted communication). This protection of messages might be used for SPDM defined messages or messages defined by another specification, such as PLDM.

43 Note that, while the SPDM specification allows for an encryption-only session, such sessions are open to additional attacks and are not recommended for most use cases.

5.4 Security Platform and Data Model (SPDM) architecture

A platform management subsystem in a modern enterprise computer platform comprises a set of components, which communicate to perform management functions within the platform. In many cases, these communications occur between components that comprise one or more mutable elements, such as firmware or software, re-programmable logic (FPGA), and re-programmable microcode. Further, a computer platform might contain immutable components, which comprise fixed logic or fixed firmware or software.

In such a platform management subsystem, stakeholders have a desire to establish trust, and to reestablish trust over time, with a component before securely communicating with that component.

The DMTF SPDM provides an authentication mechanism to establish trust, which uses proven cryptographic methods that protect the authentication process. As part of establishing trust between two endpoints, the SPDM specification enables the creation of a session to exchange secured messages between the endpoints.

For the purposes of this white paper, a component can encompass a number of component types, including PCIe adapters, Baseboard Management Controllers, purpose-built authentication components, Central Processing Units, platform components that are attached over I2C, and more. Each of these components represents a potential attack vector, through the insertion of counterfeit components, the compromise of firmware, or other attacks.

The SPDM enables these mechanisms to authenticate and secure communication with a component:

1. The retrieval of a public key certificate from a component, and a protocol to challenge the component to prove that it is the component whose identity is uniquely described by that certificate.
2. The retrieval of a signed measurement payload of mutable components from a component. These measurements can represent a firmware revision, component configuration, the Root of Trust for Measurements, hardware integrity, and more.
3. The negotiation of session keys with a component, enabling Secured Message exchanges with that component.

Finally, SPDM includes provisions for future expansion, by adding operations and capabilities while maintaining compatibility with existing deployments.

5.5 SPDM standards overview

SPDM specifies a method for managed device authentication, firmware measurement, and certificate management. SPDM defines the formats for both request and response messages that enable the end-to-end security features among the platform-management components.

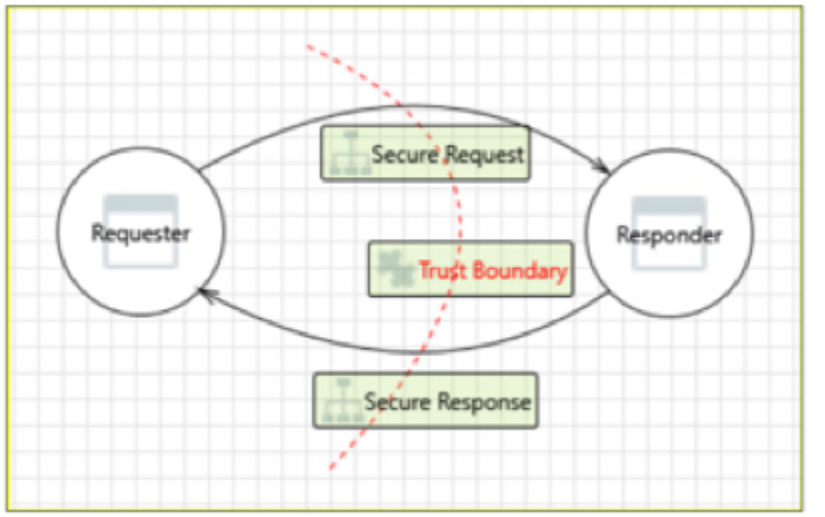
The SPDM specifications include:

- *Security Protocol and Data Model (SPDM) Specification (DSP0274)*
- *Security Protocol and Data Model (SPDM) over MCTP Binding Specification (DSP0275)*
- *Secured Messages using SPDM over MCTP Binding Specification (DSP0276)*
- *Secured Messages using SPDM Specification (DSP0277)*

5.6 Threat model

The risk assessment identifies threats and vulnerabilities related to the SPDM interactions between components. Figure 1 — SPDM threat model shows the SPDM interaction between components. The following threat model follows the STRIDE model. See *STRIDE (security)* for more details.

Figure 1 — SPDM threat model



> **Scope of this risk assessment:**
>
> The scope of this assessment includes the security controls of the component as it comprises data model security and authentication. Any limitations of the physical I2C, I3C, PCIe, GenZ, CXL, or any other network channel shall not apply to this threat assessment.

Table 1 — Threat modeling assessment and mitigations describes the threat modeling assessment and mitigations:

Table 1 — Threat modeling assessment and mitigations

STRIDE category	Description	Justification mitigation
Spoofing	Packets or messages without sequence numbers or timestamps can be captured and replayed in a wide variety of ways. Implement or use a communication protocol that supports anti-replay techniques, which investigate sequence numbers before timers, and strong integrity.	To prevent replay attacks, the Requester and Responder shall use a random nonce.
Tampering	Attackers who can send a series of packets or messages might overlap data. For example, packet 1 might be 100 bytes starting at offset 0. Packet 2 might be 100 bytes starting at offset 25. Packet 2 overwrites 75 bytes of packet 1. Ensure that you both reassemble data before filtering it and explicitly handle these sorts of cases.	To prevent intruders from tampering with exchanged data, use one or more of these strategies: • Strong authorization schemes • Hashes • Message authentication codes • Digital signatures
Information Disclosure	Custom authentication schemes are susceptible to common weaknesses, such as weak credential change management, credential equivalence, easily guessable credentials, absent credentials, downgrade authentication, or a weak credential change management system. Consider the impact and potential mitigations for your custom authentication scheme.	To prevent attacks, use one or more of these strategies as supported by the endpoint components: • Stronger authentication schemes • Versions • Cryptographic algorithms

STRIDE category	Description	Justification mitigation
Elevation of Privilege	Requester or Responder might be able to impersonate the context of the Requester or Responder to gain additional privilege.	Out of scope. The endpoint that receives the request or response must mitigate this activity. The contents of the message are not interpreted at the MCTP layer.
Repudiation	Requester or Responder claims that it did not receive data from a source outside the trust boundary. Consider using logging or auditing to record the source, time, and summary of the received data.	To mitigate attacks, use one or more of these strategies: • Digital signatures • Timestamps • Audit trails
Information Disclosure	Credentials on the wire are often subject to sniffing by an attacker. Are the credentials re-usable or re-playable? Are the credentials included in a message? For example, sending a ZIP file with the password in the email. Use strong cryptography for the transmission of credentials. Use the OS libraries, if possible, and consider cryptographic algorithm agility rather than hard-coding a choice.	To mitigate this attack, use stronger authentication schemes and cryptographic algorithms.
Denial of Service	Requester or Responder crashes, halts, stops, or runs slowly. In all cases, an availability metric is violated.	Out of Scope. To address uncorrectable errors or any type of crash, the Requester or Responder shall implement recovery mechanisms.

STRIDE category	Description	Justification mitigation
Denial of Service	External agent interrupts data flowing across a trust boundary in either direction.	If physical access is possible and the Start of Message and End of Message bits are not protected, a message can be dropped for one of the following reasons: 1. Receipt of the end packet for a message. 2. Receipt of a new start packet. 3. Timeout waiting for a packet. 4. Out-of-sequence packet sequence number. 5. Incorrect transmission unit. 6. Bad message integrity check. Only the whole MCTP message is secure. The individual MCTP packets are not secure.
Elevation of Privilege	Requester or Responder might be able to remotely execute code for the Responder.	Out of scope. The endpoint that receives the request or response must mitigate this activity. The contents of the message are not interpreted at the MCTP layer.
Elevation of Privilege	Attacker might pass data into the Requester or Responder to change the flow of program execution within Requester or Responder to the attacker’s choosing.	Out of scope. The endpoint that receives the request or response must mitigate this activity. The contents of the message are not interpreted at the MCTP layer.

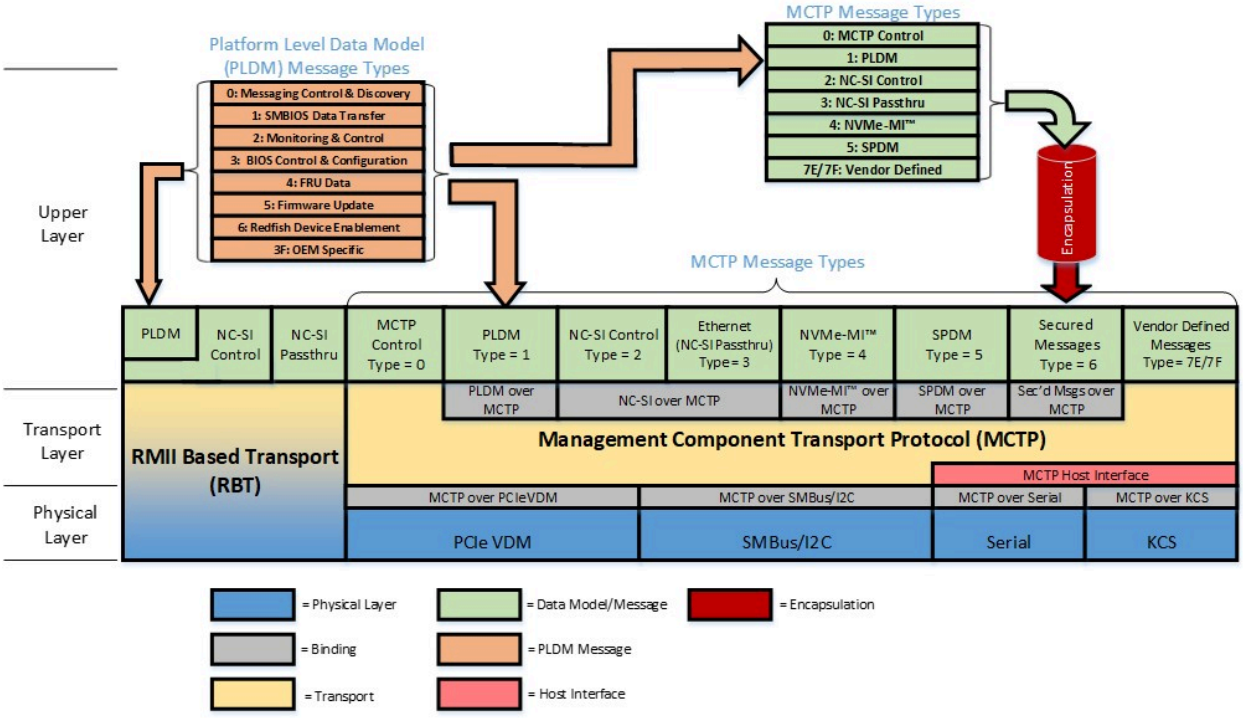
62 # 6 SPDM concepts

63 ## 6.1 PMCI stack

64 Figure 2 — SPDM over MCTP shows the relationship among SPDM messages and other messages that use MCTP. Messages that the SPDM specification defines use MCTP message type 5, and might be used in conjunction with other MCTP message types. Messages that provide authentication support use MCTP message type 5. MCTP message type 6 is used in conjunction with other MCTP message types to enable Secured Messages.

65 **Figure 2 — SPDM over MCTP**

66

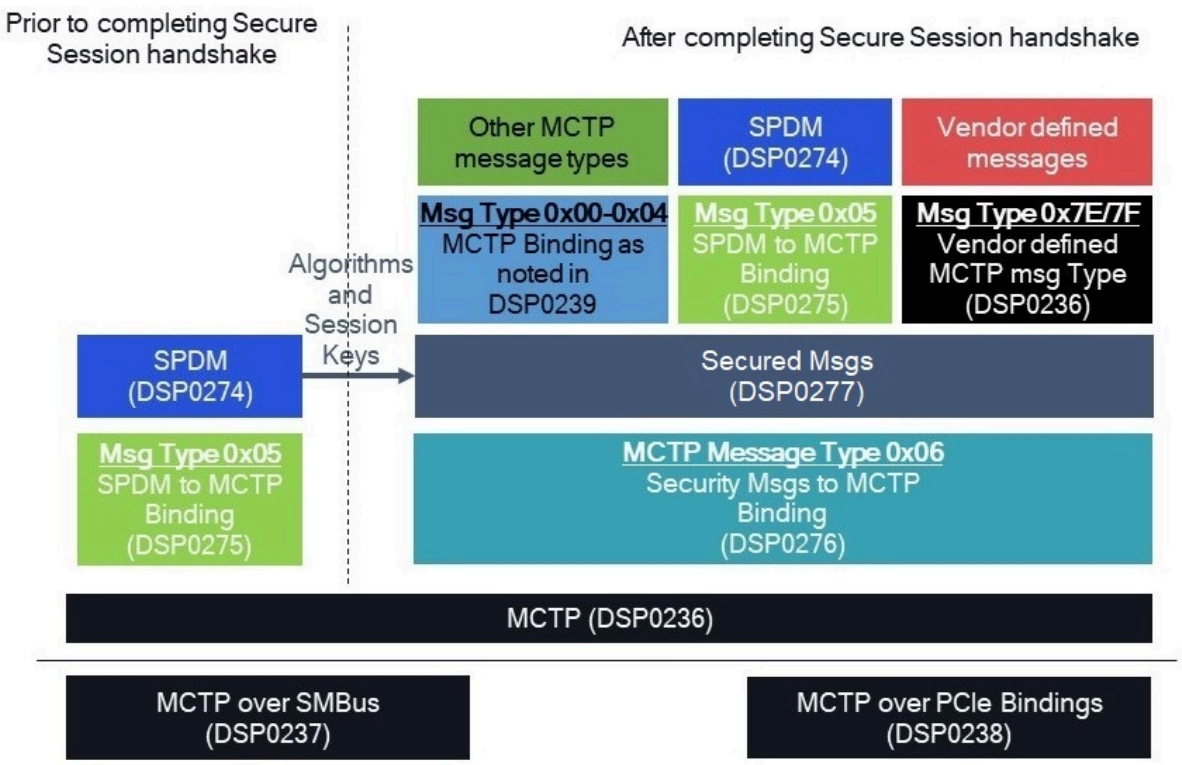


67 For details on the relationships among PMCI specifications, see the *Platform Management Components Intercommunication (PMCI) Architecture White Paper* (DSP2015).

68 Figure 3 — SPDM security stack shows the relationship among the security related specifications produced by the PMCI Working Group, and the relationships to other specifications produced by the PMCI Working Group.

69 **Figure 3 — SPDM security stack**

70



71 The *Security Protocol and Data Model Specification* (DSP0274) defines the contents of the messages, supported exchanges, and requirements.

72 The *Security Protocol and Data Model (SPDM) over MCTP Binding Specification* (DSP0275) defines the method for transporting SPDM messages over an MCTP transport.

73 The *Secured Messages using SPDM over MCTP Binding Specification (DSP0276)* binds Secured Messages using SPDM specification (DSP0277) to the MCTP transport.

74 The *Secured Messages using SPDM Specification (DSP0277)* defines the methodology that various PMCI transports can use to communicate various application data securely by utilizing SPDM.

75

6.2 Other bindings

76 Other standards bodies can create binding specifications that enable SPDM on transports other than those defined by DMTF. While many of the concepts in this white paper might apply to those implementations, the details of non-DMTF SPDM bindings are beyond the scope of this white paper.

77 For more information related to other binding specifications, see Partner implementations.

78 ## 6.3 Integration with Redfish

79 DMTF strives to achieve tight integration between DMTF standards to ensure a complete ecosystem is available to solution providers. Redfish has resources available to represent SPDM information as part of this goal. ComponentIntegrity is the main resource that indicates SPDM state and status between endpoints, and there are actions on this resource to obtain measurements. Additionally, the Certificate resource, along with Certificates collections, is used to represent the certificates used in SPDM and the reader can find them under nearly every resource in the model. There is also a CertificateService along with CertificateLocations to help a security administrator. As Redfish grows, additional resources may be added over time and this is not a complete list of security related resources, as there are resources for TPMs and other items of interest to a security administrator.

80

7 SPDM trusted computing base

81 The SPDM protocol provides authentication of devices and attestation of firmware running on a device (including firmware configuration). This means that the SPDM software stack becomes a part of the trusted computing base (TCB) for a device and a verifier, and the code must be implicitly trusted. As is typical of any TCB, a compromise in the TCB is undetectable and the trustworthiness of attestation reports are only as trustworthy as the TCB. There is no mechanism prescribed by the SPDM specification for protection, detection and recovery of the TCB. To provide higher security assurances around the TCB, device manufacturers and implementers can use methods outside the specification to protect, detect, and recover the TCB.

82 # 8 Certificates

83 If a Responder supports the certificate-related SPDM `GET_DIGESTS`, `GET_CERTIFICATE`, and `CHALLENGE` requests, the Responder must be provisioned with at least one certificate chain. If a Responder only supports the `GET_MEASUREMENTS` request, but cannot perform signature generation, it does not require a certificate chain or need to follow the guidance in the rest of this clause. A less capable component might be implemented in such a manner so that it does not require as much processing power or because such an implementation is conformant to the component's requirements. Whether a Requester accepts such a component is dependent on the Requester's security policy.

84 ## 8.1 Background on certificates

85 The SPDM specification uses X.509 v3 certificates, as defined in RFC5280, to communicate identity information between two components. The use of X.509 v3 certificates has the following advantages:

- When properly validated, X.509 v3 certificates are resistant to tampering.
- X.509 v3 certificates are standards based and widely supported.
- X.509 v3 certificates can use extensions to capture and convey other information, including information structures that DMTF defines.

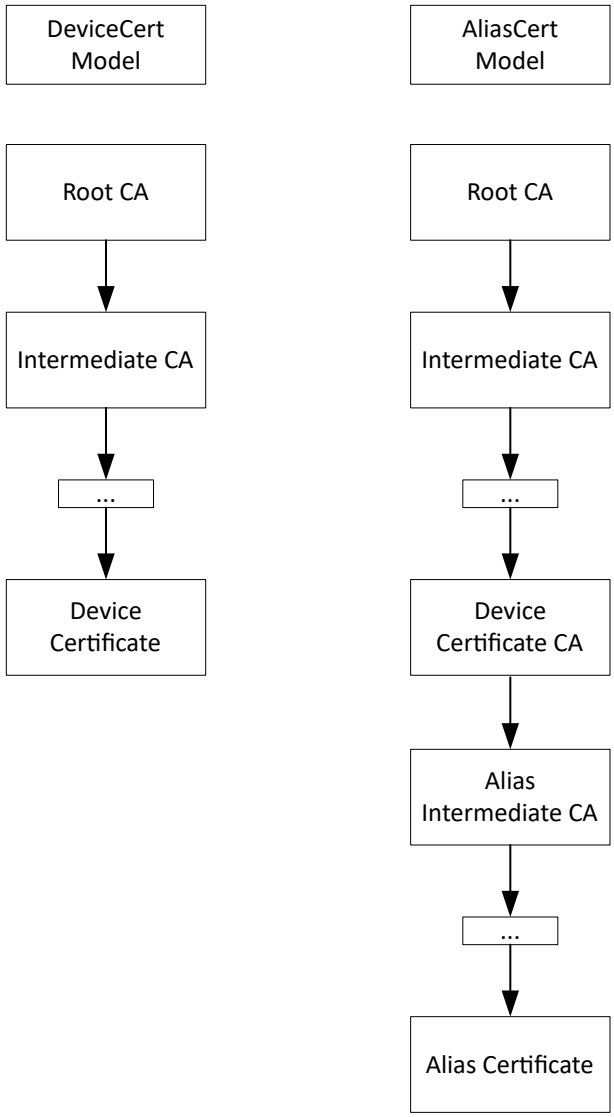
86 ## 8.2 Certificate overview

87 During the certificate-related SPDM request sequence, the Requester attempts to determine the identity of the Responder based on the certificate chain that the Responder returns. To report its identity, the Responder returns a chain of linked certificates that include at least a device certificate and a certificate issued by a CA that the Requester trusts. The certificate that the Requester trusts could be a root certificate or an intermediate certificate.

88 Figure 4 — Example certificate chain shows an example certificate chain:

89 **Figure 4 — Example certificate chain**

90



91 Table 2 — Certificate chain elements summarizes the roles of the elements that Figure 4 — Example certificate chain shows.

92 **Table 2 — Certificate chain elements**

Certificate chain element	Description
Root certificate	Conceptually the highest certificate in the chain. Contains a record of the issuing authority and is self-signed.

Certificate chain element	Description
Intermediate certificate	A certificate chain typically contains one or more of these certificates, which enable the allocation of separate intermediate certificates to different device families or product divisions within a company. This enables flexibility in establishing complex hierarchies of certificates for easier revocation and to protect the root certificate private key which might be kept offline.
Device certificate	Uniquely identifies the component. Should not change over the life of a component, unless the component is re-provisioned. If an operation changes the Device key pair, then the device certificate must be replaced. When a component uses the DeviceCert model, the device certificate is the lowest level certificate in the certificate chain and is referred to as the leaf certificate. When a component uses the AliasCert model, the device certificate is a certificate authority that signs certificates below it.
Alias intermediate CA	When a component uses the AliasCert model, one or more alias intermediate certificate authorities might be present. The alias intermediate CAs might contain information related to firmware layers or component configuration, and the key pair associated with an alias intermediate CA might change as the result of a firmware update or another operation.
Alias certificate	When a component uses the AliasCert model, the component presents an alias certificate as the lowest level certificate, which is also referred to as the leaf certificate. The alias certificate might contain information related to firmware layers or component configuration, and the key pair associated with an alias certificate might change as the result of a firmware update or another operation.

93 ### 8.2.1 Certificate chain models

94 The SPDM specification defines two structures for a certificate chain. As shown in Figure 4 — Example certificate chain, a certificate chain can follow either the DeviceCert model or the AliasCert model. When a component uses the DeviceCert model, the device certificate is the leaf certificate. When a component uses the AliasCert model, the

device certificate is a certificate authority that signs additional certificates below the device certificate. In the case of the AliasCert model, the leaf certificate is referred to as the alias certificate.

95 Prior to version 1.2 of the SPDM specification, certificate chains used in an SPDM conformant implementation were understood to follow only the DeviceCert model. The AliasCert model was an addition to the 1.2 specification. Version 1.3 of the SPDM specification adds the GenericCert model, which allows the use of a Generic Certificate that does not fit any of the other SPDM certificate models.

96 A Responder indicates the use of the AliasCert model by setting `ALIAS_CERT_CAP=1` in the `CAPABILITIES` response. The Requester does not have a corresponding flag in the `GET_CAPABILITIES` request. For a variety of reasons, a Responder might not be able to switch between the DeviceCert and AliasCert models. Since a Requester cannot control the certificate chain model that a Responder uses, Requesters that conform to this version of the SPDM specification are recommended to be capable of processing certificate chains for either model.

97 8.2.1.1 Certificate model use cases

98 Support for multiple certificate chain formats allows flexibility in component implementations. Components might select between the different certificate chain models for a variety of reasons. An implementer might find the DeviceCert model to be easier to implement. However, an AliasCert model might be selected to enable certificates to convey additional information, or to comply with standards created by partner standards bodies. An implementer might use a GenericCert model to enable features that are not supported in the SPDM specification.

99 Certificate chain slots 1-7 are general purpose slots, but certificate chain slot 0 is different and is intended to store a certificate chain that conveys at least the hardware identity of the component. Because a generic certificate chain is not required to conform to most certificate requirements in the SPDM specification and may not convey component hardware identity, a GenericCert model certificate chain is not allowed in slot 0.

100 Since the SPDM specification only allows a GenericCert in slots 1-7, the certificate chain in slot 0 must follow either the DeviceCert or AliasCert model. The SPDM specification only permits a mix of certificate chain models when `MULTI_KEY_CAP=1`.

101 **Table 3 — Certificate chain models**

Certificate chain model	Description
DeviceCert	The DeviceCert model was introduced in version 1.0 of the SPDM specification. A DeviceCert chain uses a single device certificate to convey hardware identity and any other component details.

Certificate chain model	Description
AliasCert	The AliasCert model was introduced in version 1.2 of the SPDM specification. An AliasCert chain uses a device certificate plus one or more alias certificates to convey hardware and firmware identity, plus it can optionally convey configuration details. The AliasCert model allows any certificate chain that follows the general format specified in the SPDM specification, but was specifically introduced to enable the use of TCG DICE certificate chains.
GenericCert	The GenericCert model was introduced in version 1.3 of the SPDM specification. The GenericCert chain supports certificate chains that do not fit cleanly into one of the other certificate chain models.

102 #### 8.2.1.2 Embedded certificate authority protection

103 A Responder that implements the AliasCert model might make use of an Embedded Certificate Authority (ECA), which would be used to generate the mutable certificates in the AliasCert chain. This type of implementation carries risks, as an attacker that gains control of an ECA may use it to issue fraudulent certificates. A component is recommended to only issue a certificate to a firmware layer that it has correctly authenticated using a Root of Trust for detection.

104 #### 8.2.1.3 Hardware identity

105 The SPDM specification makes references to certificates that contain hardware identity. These can be a Device Certificate or another type of certificate. If the component uses a GenericCert chain, it is still recommended to have a certificate that contains hardware identity, but that certificate is not required to meet the other requirements associated with a Device Certificate. Hardware identity is typically a method for identifying the unique instance of hardware that is associated to the identity. The inclusion of the hardware identity OID (`id-DMTF-hardware-identity`) in the correct certificate can help a verifier determine which certificate should be evaluated for the hardware identity. The following are suggestions for how a certificate can convey hardware identity:

- A public key that is unique to a component instance.
- A combination of the component type and component serial number.

106 ### 8.2.2 Certificate chain validation

107 Before a Requester uses the contents of a certificate chain, it must validate the certificate chain to ensure that it is properly formed. RFC5280 specifies the detailed process for validating a certificate chain. To assist the reader, the process is summarized here (note, the discussion in this section is based on the diagram in Figure 4 — Example certificate chain):

- Check each certificate to ensure that it references the certificate above it in the chain.
- Validate the signature in each certificate using the public key from the certificate above it in the chain.
- Read the validity dates, key usage policies, and other constraining information from the certificates to verify that the certificate and its associated key pair are being used correctly.
- Ensure that the root certificate is a known and trusted certificate.
- Ensure that none of the certificates in the certificate chain has been revoked.

108 After the RFC5280 based certificate chain validation is complete, the Requester knows that the certificate chain is correctly formed but this information is insufficient. The Requester still must ensure that the Responder is the component that should be returning this certificate chain. This check is performed by verifying that the Responder has knowledge of the private key associated with the public key in the leaf certificate by using the `CHALLENGE` message exchange.

109 The Requester may perform the validation of the Responder's certificate chain on the Requester device itself or have a remote verifier (such as a back-end server) perform the validation. A remote verifier is usually more capable of certificate validation, as it may have access to trusted time and latest certificate revocation lists published by certificate issuers. Some use cases may allow the Requester to use the public key in the Responder's leaf certificate before the certificate chain is validated by a remote verifier, as long as no asset would be compromised.

110 ## 8.3 SPDM certificate slots

111 The SPDM specification defines a total of eight slots for storing certificate chains, with each slot storing a complete and independent certificate chain. Further, the SPDM specification states that the component uses the same asymmetric key pair for the leaf certificate located in each slot. The certificate chain for each slot can contain different root certificates. While SPDM supports up to eight certificate slots, only slot 0 is required to be present for components that use certificates. Further, a component can implement fewer than eight certificate slots, such as three slots.

112 The certificate chain in slot 0 has a special role in the system because the component manufacturer provisions the contents of slot 0 during manufacturing. The certificate chain in slot 0 represents the manufacturer, and this certificate chain is often immutable, though immutability is not required by the SPDM specification. This certificate chain is also known as the *manufacturer certificate chain*.

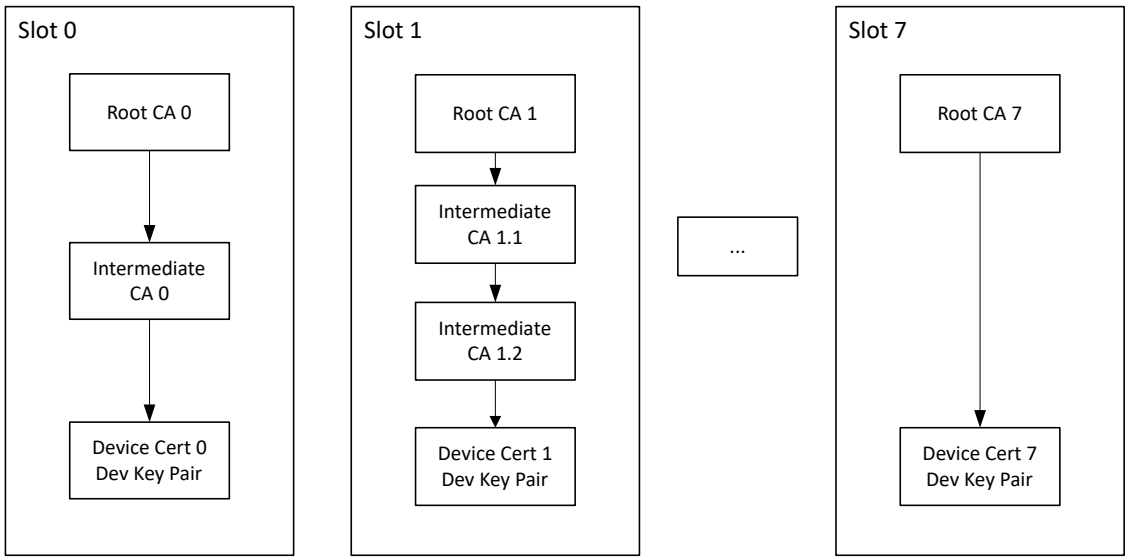
113 Some deployment use cases might make use of certificate slots 1 to 7. For instance, an administrator can claim ownership of a component by installing a certificate chain belonging to the administrator in one or more of the additional slots (certificate slots 1 to 7). The use of these additional slots enables the administrator to authenticate the component using a certificate chain that is owned and managed by the administrator. Another use of additional certificate slots is to set certificate validity ranges that expire in a shorter time-frame than the certificate chain installed by the component vendor.

114 The SPDM specification requires that the certificate chains in all slots use the same key pair in their leaf certificates. An implication of this requirement is that all slots must use the same certificate chain model, either the DeviceCert model or the AliasCert model.

115 Figure 5 — Example certificate slots shows an example of the use of certificate slots:

116 **Figure 5 — Example certificate slots**

117



118 ### 8.3.1 Stored certificate chain format

119 The SPDM specification indicates that the certificate chain returned to the Requester is formatted such that the first certificate is signed by the root certificate, or is the root certificate itself, and each subsequent certificate is signed by the preceding certificate until the leaf certificate. The returned certificate chain is also to include a hash of the root certificate. Implementers are recommended to store the entire certificate chain in a slot, including the root certificate, so that the hash can be generated with the currently negotiated algorithm.

120 A Responder can choose to send one of the two certificate chain formats (with or without the root certificate) depending on the situation. For instance, a Responder could send the certificate chain formatted without the root certificate when using a slower transport.

121 ## 8.4 Certificate chain algorithms

122 A leaf certificate chain is implicitly tied to `BaseAsymSel` or `ExtAsymSel`, as the `ALGORITHMS` message exchange defines. The negotiated `BaseAsymSel` or `ExtAsymSel` field must match the algorithms used in the Subject Public Key Info in the leaf certificate on the Responder. For compatibility purposes, a component vendor can provision a component with certificate chains that correspond to multiple `BaseAsymSel` and/or `ExtAsymSel` values. For instance, a component can have one set of certificate chain slots that it uses for `TPM_ALG_ECDSA_ECC_NIST_P384`, and another set of certificate chain slots that it uses for `TPM_ALG_RSASSA_3072`. In this case, the Responder uses the negotiated algorithm set to select among its different sets of certificate chain slots. In such an implementation, it's feasible that the populated certificate slots could differ between the different sets of certificate chain slots. The definition and reporting of a slot management mechanism such as this is out of scope for the SPDM specification.

123 ### 8.4.1 Certificate chain verifier compatibility

124 The set of cryptographic algorithms that the Requester and Responder negotiate during the `ALGORITHMS` exchange match the cryptographic algorithms used in the leaf certificate. However, a Responder typically returns a certificate chain with multiple certificates in the `CERTIFICATE` response. When validating the returned certificate chain, the Requester should not assume that all certificates in the certificate chain use the same cryptographic algorithms as the leaf certificate. For the sake of compatibility, a Responder should constrain itself to use cryptographic algorithms specified in the SPDM `NEGOTIATE_ALGORITHMS` exchange, and Requesters should support the use of all cryptographic algorithms specified in the SPDM `NEGOTIATE_ALGORITHMS` exchange.

125 ## 8.5 Certificate requirements

126 Certificate chains follow the X.509 v3 format, and are DER-encoded. Certificate chains can be long compared to other SPDM messages, so Requesters should ensure that buffers are large enough to receive them. The maximum length of a certificate chain that can be conveyed by SPDM is 64 KiB. The support to verify signatures of different cryptographic algorithms on the certificate chains remain the responsibility of Requester and Responder implementations. It is expected that they support verification of commonly accepted algorithms to promote interoperability.

127 The leaf certificate in the certificate chains must conform to the SPDM specification, *Leaf certificate* clause defined format. The certificate format guidance in SPDM is based on RFC5280. Table 4 — Optional leaf certificate attributes describes the leaf certificate attributes that the SPDM specification specifies as optional.

128 ### 8.5.1 Certificate retrieval

129 If a Requester cannot allocate a buffer for the maximum certificate chain size of 64 KiB, the Requester can issue a `GET_CERTIFICATE` request with the `Length` field set to a small number, such as four bytes. In this case, the Responder returns the requested portion of the certificate chain and the remaining length in the `RemainderLength` field. SPDM provides a mechanism to segment a certificate chain using the `Offset` and `Length` fields in the `GET_CERTIFICATE` request to retrieve the certificate chain in smaller increments. This mechanism can compensate for Requesters, Responders, or transports that cannot transfer an entire certificate chain in one response message.

130 A Requester should anticipate that a Responder might not be capable of sending the entire certificate chain in one transaction, even if the Requester is capable of allocating a sufficiently large buffer.

131 The SPDM specification does not prohibit a Requester from reading only a portion of multiple certificate chains, for instance, reading the root certificate from each slot or toggling between two slots. However, implementers should be aware that there may be a performance penalty for a component to switch between slots, such as repeated buffer clears and flash reads.

132 ### 8.5.2 Certificate fields

133 X.509 v3 certificates contain multiple fields, as defined by RFC5280. In addition, the SPDM specification specifies usage of some X.509 v3 defined fields.

134 **Table 4 — Optional leaf certificate attributes**

Attribute	Description
`Validity` (`notBefore`)	If present, it is recommended that the `notBefore` field of the `Validity` attribute should be set to `19700101000000Z` , which is the minimum `Validity` date. Because most Requester and Responder pairs do not contain a real-time clock, the use of the minimum `Validity` date ensures that the Requester ignores the `notBefore` field.
`Validity` (`notAfter`)	If present, it is recommended that the `notAfter` field of the `Validity` attribute should be set to `99991231235959Z` , which is the maximum `Validity` date. Because most Requester and Responder pairs do not contain a real-time clock, the use of the maximum `Validity` date ensures that the Requester ignores the `notAfter` field.
`Subject Alternative Name`	Recommended. It enables reporting of more detailed and standardized component identification.

Attribute	Description
`Extended Key Usage (EKU)`	If present, the Extended Key Usage extension indicates one or more purposes for which the public key should be used. The following Extended Key Usage purposes are defined for SPDM certificate authentication: SPDM Responder Authentication (`1.3.6.1.4.1.412.274.3`): The presence of this OID shall indicate that a leaf certificate is used for Responder authentication purposes. SPDM Requester Authentication (`1.3.6.1.4.1.412.274.4`): The presence of this OID shall indicate that a leaf certificate is used for Requester authentication purposes. The presence of both OIDs shall indicate that the leaf certificate is used for both Requester and Responder authentication purposes. A Responder device that supports mutual authentication should include the SPDM Responder Authentication OID in the Extended Key Usage field of its leaf certificate. A Requester device that supports mutual authentication should include the SPDM Requester Authentication OID in the Extended Key Usage field of its leaf certificate.

135 Though not required, the SPDM specification details the `Subject Alternative Name` for components that are SPDM conformant. Standards bodies that create additional binding specifications for SPDM should specify appropriate guidelines for the `Subject Alternative Name` and Common Name fields (see Partner implementations). All standards bodies that use the SPDM specification should retain the `Serial Number` field in the certificate definition.

136 A certificate should use the `otherName` field in the `Subject Alternative Name` to provide detailed information about the manufacturer, product, and serial number.

137 The OID in the `otherName` field is `1.3.6.1.4.1.412.274.1`. This value represents a UTF8String in the `<manufacturer>:<product>:<serialNumber>` format.

138 The following example string shows the format of the SPDM defined `Subject Alternative Name` `otherName` field:

```
otherName:1.3.6.1.4.1.412.274.1;UTF8STRING:ACME:WIDGET:0123456789
```

139 The X.509 v3 certificates can include the `Authority Key Identifier`, which assists authentication of the certificate

chain. This assistance is especially important for the certificate that is immediately below the root certificate because the `Authority Key Identifier` can help the Requester locate the root certificate in its trust store. The presence of the `Authority Key Identifier` can also help with debug of certificate chain problems, by illustrating how certificates are intended to connect.

140 ## 8.6 Interpreting certificate contents

141 A certificate chain contains information that a Requester can interpret to make policy decisions about a given Responder. Once a certificate chain has been validated, as described in Certificate chain validation, a Requester can use the Certificate fields to interpret the information contained in the certificate chain. While many of the fields are interpreted as defined in RFC5280, some fields are defined by the SPDM specification.

142 Table 5 — Interpretation of select certificate fields summarizes potential use cases for select SPDM specification defined Certificate fields.

143 **Table 5 — Interpretation of select certificate fields**

Field	Required or optional	Interpretation
`Subject Alternative Name` `otherName`	Optional	The `otherName` field provides identifying details for the component in a machine parsable manner. A Requester could use this field to match the identity of the component with the same information obtained through other channels, to create an entry for the component in a database, or to display information about the component to a user.

144 ### 8.6.1 Comparison of SPDM certificates to other standards

145 In many cases, identity of devices and of the platform in a system will be presented via a collection of SPDM certificates as well as certificates specified by other industry standards. Table 6 — Comparison of X.509 identity certificate fields summarizes how SPDM and other standards define the contents of X.509 certificate fields.

146 **Table 6 — Comparison of X.509 identity certificate fields**

Field	SPDM	IEEE/TCG DevID
Version	V3 (encoded as 2).	V3 (encoded as 2).
Serial number	Shall be present with a positive integer value.	Must be a unique (per CA) integer. Must be a positive integer of up to 20 octets.
Signature algorithm	Shall be present.	Refer to RFC5280, RFC3279, RFC4055, RFC4491, RFC5480, RFC8692.
Issuer	CA distinguished name shall be present.	Refer to RFC5280.
Validity: notBefore	If present, should be `19700101000000Z`.	Shall be the date of certificate creation.

Field	SPDM	IEEE/TCG DevID
Validity: notAfter	If present, should be `99991231235959Z`.	Should use the value `99991231235959Z`.
Subject	Subject name shall be present and shall represent the distinguished name associated with the leaf certificate.	Must comply with IEEE 802.1AR: 1) Must be present. 2) Must be unique in domain of signing CA. 3) Should contain device serial number encoded as `X520SerialNumber`.
Subject Public Key Info	Device public key and the algorithm shall be present.	Refer to RFC5280.
Authority Key Identifier	Not specified in DSP0274, recommended in DSP2058.	Required for compliance with IEEE 802.1AR.
Subject Key Identifier	Not specified.	Required for compliance with IEEE 802.1AR. Not recommended for leaf certs.
Key Usage	Shall be present and key usage bit for digital signature shall be set.	`digitalSignature` (only) recommended, `digitalSignature` and `dataEncipherment` (combined) permissible.
Extended Key Usage (EKU)	May have `id-DMTF-eku-requester-auth` and/or `id-DMTF-eku-responder-auth`	May have `tcg-kp-EKCertificate`.
Certificate Policy	None.	Multiple OIDs used to identify certificate type, TPM residency, etc.
Subject Alternative Name	`otherName` encoding defined.	Can be present. `hardwareModuleName` describes the TPM hardware version. `PersistentIdentifier` identifies TPM based on TPM endorsement key certificate.
Basic Constraints	If included, CA shall be set to false in the leaf certificate.	Must be included and set to critical CA=FALSE.

8.7 Example leaf certificate

The following example shows a leaf certificate:

```
          Certificate:
              Data:
                  Version: 3 (0x2)
                  Serial Number: 4097 (0x1001)
                  Signature Algorithm: ecdsa-with-SHA256
                  Issuer: C = US, ST = NC, L = City, O = ACME, OU = ACME Devices, CN = CA
                  Validity
                      Not Before: Jan  1 00:00:00 1970 GMT
                      Not After : Dec 31 11:59:59 9999 GMT
                  Subject: C = US, ST = NC, L = City, O = ACME Widget Manufacturing, OU = ACME Widget
```

```
Manufacturing Unit, CN = w0123456789
                Subject Public Key Info:
                    Public Key Algorithm: rsaEncryption
                        Public-Key: (2048 bit)
                        Modulus:
                            00:cc:41:73:a3:f1:ff:78:ff:78:f5:e1:a7:3c:2e:
                            ae:40:82:db:04:eb:ad:e8:54:e7:8f:4a:76:3c:a2:
                            21:77:72:e7:70:a6:0a:b3:7a:a3:e8:af:49:5c:ec:
                            57:00:6b:6e:0b:09:b7:f0:be:35:c4:ec:e8:f8:28:
                            0c:0a:b8:59:48:a7:14:47:88:05:c5:8c:1e:e5:79:
                            5a:2b:31:fe:14:27:12:eb:ba:53:40:74:43:5b:e0:
                            f4:be:45:93:f8:87:b6:a3:13:f1:7c:72:5f:c1:aa:
                            a6:be:fd:e8:c4:3a:ae:24:0e:81:25:c6:f2:6c:fd:
                            53:27:89:4c:f6:37:22:cf:25:5d:51:b9:30:54:61:
                            fe:0b:23:2f:dd:e3:1b:87:30:a4:b3:16:41:48:51:
                            1e:17:29:3a:2b:57:1c:41:67:27:62:15:08:6e:c1:
                            59:8d:d7:c3:0f:33:05:26:a0:1b:b9:f5:b4:36:0d:
                            bb:ec:24:5d:bb:c9:0b:b2:57:1b:7b:18:21:d4:c0:
                            ec:fd:0a:03:33:4e:b0:55:e7:3f:26:b1:96:1f:b3:
                            2a:18:2d:88:4d:cd:9c:26:08:2c:d7:fc:5f:87:b4:
                            e8:06:ad:6d:ce:65:0f:88:26:85:7d:aa:54:6d:57:
                            34:34:ae:40:83:15:ee:cf:2c:06:ee:69:52:92:9b:
                            b0:77
                        Exponent: 65537 (0x10001)
                    X509v3 extensions:
                        X509v3 Basic Constraints:
                            CA:FALSE
                        X509v3 Authority Key Identifier:
                            CB:0C:55:D9:4F:18:EE:B9:54:25:3D:08:1A:4C:02:24:80:BF:CF:FE
                        X509v3 Key Usage: critical
                            Digital Signature
                        X509v3 Subject Alternative Name:
                            otherName: 1.3.6.1.4.1.412.274.1::ACME:WIDGET:0123456789
                Signature Algorithm: ecdsa-with-SHA256
                Signature Value:
                    30:44:02:20:3d:c9:e5:59:43:a5:f1:56:3e:8f:cb:ef:96:e1:
                    bc:4d:bd:ca:d1:a7:69:7e:10:0e:58:74:5b:89:2a:b4:b2:59:
                    02:20:2a:0d:95:4e:52:05:c0:fe:44:7b:61:ec:38:f7:87:95:
                    8b:60:c5:89:03:d8:4e:c4:1c:0b:57:a3:de:67:45:83
```

149 8.8 Certificate provisioning

150 If a component supports the SPDM certificate-related commands, the manufacturing process for that component must provision a certificate chain to each component instance.

151 Possible methods to create a certificate chain include:

- Generate a certificate signing request (CSR) using the firmware of the component. A CA checks the CSR and signs it to create the appropriate certificate chain.

- Export the information required to form a CSR to an external utility, which generates the CSR. A CA checks the CSR and signs it to create the appropriate certificate chain.
- If a component uses an externally-provisioned key, generate the necessary certificate as part of the external key-generation process and load the generated key and certificate chain into the component. See Key provisioning.

152 After import, the component should check the certificate chain to ensure that its public key matches the component's Device public key.

153 This type of procedure could be used to provision a certificate chain to one of the slots numbered 1-7. Exact mechanisms to implement such procedures are outside the scope of this white paper and are not part of the SPDM specification.

154 Any approach for generating a certificate chain should occur as part of a secure manufacturing process. Keep intermediate certificates above the device certificate in a trusted environment that is not directly accessible to the component so that the component cannot sign a device CSR.

155 ### 8.8.1 Certificate slot management

156 SPDM defines messages that a Requester can use to manage and interact with certificate chain slots. SPDM defines a request attribute for the `GET_CERTIFICATE` request that allows the Requester to query the maximum size of a certificate chain that can be stored in a slot. To use this feature, a Requester issues a `GET_CERTIFICATE` request with bit 1 of `Param2` set to 1b. A Requester should note that a component may change the available storage for a slot between `SET_CERTIFICATE` operations, as the underlying storage may be shared between multiple slots.

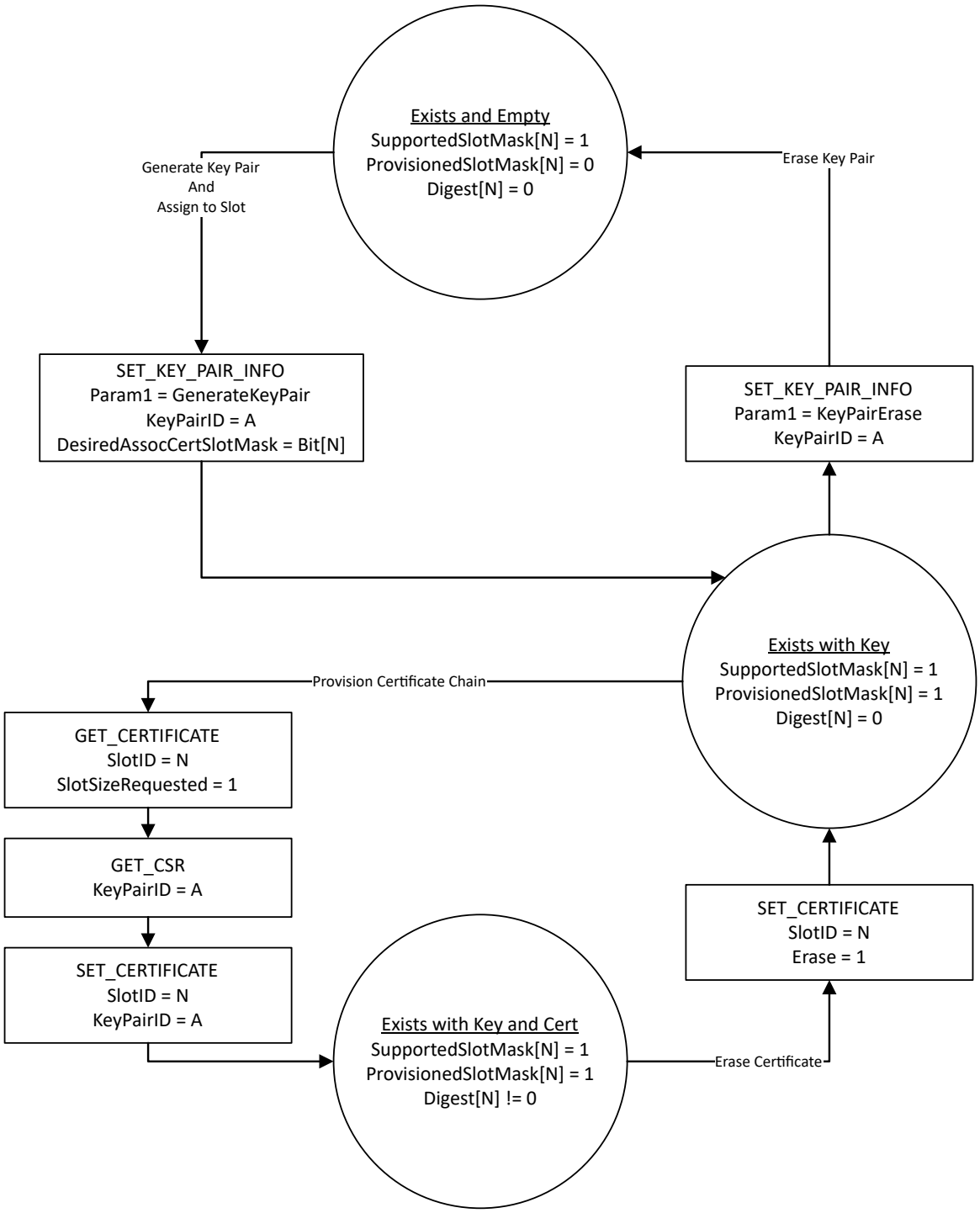
157 Version 1.3 of the SPDM specification introduces several states for certificate chain slots. The status of a slot on a component can be discovered using the `DIGESTS` exchange. If a certificate chain slot does not exist, the corresponding bit in the `SupportedSlotMask` field of the `DIGESTS` response will be set to zero. The SPDM Specification requires that if a certificate chain slot does not exist, then it will continue not to exist for at least the life of the current SPDM connection. A component might change the number of slots that exist between certain events such as resets, though any such behavior is implementation specific.

158 Starting from the Exists and Empty state, a Requester uses the `SET_KEY_PAIR_INFO` request to generate a key pair and assign it to slot N. This changes slot N to the Exists with Key state. Next, the Requester provisions a slot using three operations. The Requester optionally issues a `GET_CERTIFICATE` request to query the size of slot N, then issues a `GET_CSR`, and finally issues a `SET_CERTIFICATE` to write the certificate chain to slot N. This sequence changes slot N to the Exists with Key and Cert state. When the Requester deletes a certificate, it does so by issuing the `SET_CERTIFICATE` request to slot N, with the Erase request attribute set to 1. Deleting the certificate moves the slot back to the Exists with Key state. If the Requester needs to delete the key, it can now do so by issuing the `SET_KEY_PAIR_INFO` command with the KeyPairErase parameter, returning the slot to the Exists and Empty state.

159 Figure 6 — Certificate chain slot state machine shows a state machine for a certificate chain slot when the slot exists, and how to modify the slot state using SPDM commands. The states and associated variables in the figure are based on the section *GET_DIGESTS request and DIGESTS response messages* in the SPDM Specification. The variable `N` in the figure represents the slot in use for the operation, and the variable `A` represents the KeyPairID in use.

160 **Figure 6 — Certificate chain slot state machine**

161


Exists and Empty
SupportedSlotMask[N] = 1
ProvisionedSlotMask[N] = 0
Digest[N] = 0
Generate Key Pair
And
Assign to Slot
Erase Key Pair
SET_KEY_PAIR_INFO
Param1 = GenerateKeyPair
KeyPairID = A
DesiredAssocCertSlotMask = Bit[N]
SET_KEY_PAIR_INFO
Param1 = KeyPairErase
KeyPairID = A
Exists with Key
SupportedSlotMask[N] = 1
ProvisionedSlotMask[N] = 1
Digest[N] = 0
Provision Certificate Chain
GET_CERTIFICATE
SlotID = N
SlotSizeRequested = 1
GET_CSR
KeyPairID = A
SET_CERTIFICATE
SlotID = N
KeyPairID = A
SET_CERTIFICATE
SlotID = N
Erase = 1
Exists with Key and Cert
SupportedSlotMask[N] = 1
ProvisionedSlotMask[N] = 1
Digest[N] != 0
Erase Certificate

8.8.1.1 Slot erase and overwrite

163 SPDM provides commands to manage the contents of certificate slots, specifically intended to help manage the contents of slots 1-7. A Requester might wish to write a new certificate chain to a slot that already contains a certificate chain. In this case, a Requester should erase the contents of the target slot (using `SET_CERTIFICATE` with the `Erase` bit set) before writing a new certificate chain to the slot. A Responder might also allow a Requester to overwrite the contents of a slot, but this behavior is optional. In SPDM v1.2, the only way to replace the contents of a slot was to overwrite it. SPDM v1.3 adds the `Erase` request attribute to allow the slot to be cleared before being written. A Responder is allowed to reject overwrite attempts, so a Requester can use an erase and then write sequence to avoid a try/fail sequence.

164 The process of erasing the contents of a slot and writing a new certificate chain requires two separate `SET_CERTIFICATE` requests (one to erase and another to write). This requires two requests to keep the endpoint processing and error condition reporting simple, as both the erase and the write can generate an error. Further, multi-key support in SPDM means that a Requester might perform one or more key management commands between the erase and the write of the new certificate chain.

165 If the Responder returns an `ERROR` response to a `SET_CERTIFICATE` request, the existing contents of the slot are unaffected.

8.8.1.2 Key and slot management

167 The SPDM multi-key feature allows keys to be mapped to certificate slots. A Responder is allowed to restrict mapping a given key pair to a subset of the available slots. SPDM does not provide a command to discover this restriction, so a Requester can only discover this restriction when the Responder returns an `ERROR` with `ErrorCode` = `InvalidRequest`.

8.9 Device key pair

169 Each component must contain a public and private key pair, or a device key pair, that is statistically unique to that component. To maintain consistency of the hardware identity, the component should retain the same device key pair, unless the key is reprovisioned or altered by another operation. Any operation that alters the device key pair invalidates any certificate chain that uses it, which causes the component to fail any authentication request that depends on the current certificate chain. The Requester should handle the case where the key changes and appropriately determine the new state of the component.

170 Only one device key pair should be used for any of the occupied certificate chain storage slots. The SPDM specification supports multiple encryption and hashing algorithms. The component manufacturer chooses the algorithm for the leaf certificate from the available list in accordance with the needs of the manufacturer.

8.9.1 Key provisioning

172 There are two primary methods for provisioning a device key pair to a component, though there are multiple

mechanisms available to accomplish each of the methods. Any component that supports SPDM certificate or measurement-related command sets must provision device key pairs.

173 8.9.1.1 Internal key generation

174 If capable, a component should generate its own device key pair. A component can better protect a device private key that it generates on the component by ensuring that the device private key is never made visible outside of the component.

175 This process must be a repeatable process that always results in the generation of the same device key pair because this is the foundation of the identity of the component. A component that generates its own device key pair can follow a model, such as the DICE model of the Trusted Computing Group, that results in a key pair of similar quality.

176 A component that generates its own device key pair must:

- Be provisioned with or generate and retain a cryptographically strong random number that can be used as the Unique Device Secret (UDS).
  - All random numbers and entropy sources should conform to the NIST SP800-90 standards.
- Have sufficient processing power or hardware support to generate a key pair by using the chosen algorithm.
- Protect the source data that the key generation process uses, as discussed in Key protection.

177 8.9.1.2 External key provisioning

178 If a component cannot meet the requirements for internal key generation, it must use an external provisioning process. The external provisioning process allows the component manufacturer to rely on external tools and components, such as a Hardware Security Module (HSM), to meet requirements that the component cannot meet on its own. For instance, a manufacturer can use an external tool to provide a true random number to a component that cannot generate sufficient entropy on its own, and use the component to complete the rest of the process.

179 External key provisioning has a trade-off because the component is in an open state until the component is provisioned with the device key pair. To maintain trust in the component, the supply chain and manufacturing facilities must be highly secure.

180 Any random number used as part of the key generation process should be generated in a manner that conforms to the NIST SP800-90 standards.

181 In some cases, a user might need to re-provision a device key pair that has been provisioned to a component. However, a component must ensure that re-provisioning cannot occur except when authorized by the user; otherwise, the component may be vulnerable to a key hijack attack. The user must also ensure that the device key pair is only re-provisioned in a trusted environment. The means to provide these protections is outside the scope of the SPDM specification.

182 8.9.1.3 Firmware update impact

183 A firmware update or activation might impact the keys or certificates used by a component. A Responder might have a delay between its storage of an updated firmware version and the activation of this firmware, for example, if the

Responder requires a reset to activate the new firmware version. If the Responder has a requirement to measure the stored firmware image prior to activation, it can assign a measurement index to the stored firmware image. The component manufacturer should document the measurement index or indexes used to measure firmware images that are stored but not active.

184 After a reset, the Responder might return measurements in one of the following ways. The details of the Responder's behavior should be documented by the component vendor.

- The Responder might swap measurement indexes between the new and previous firmware versions.
- The Responder might update the indexes that hold the running firmware measurements and have matching values in the pending firmware measurement indexes.
- The Responder might update the indexes that hold the running firmware measurements and disable the pending firmware measurement indexes.

185 Some component implementations might generate new certificates, such as a new alias certificate, that use different keys or capture different measurements when new firmware is activated. In some implementations, the generation of these new certificates and/or keys might require a component reset. The SPDM specification states that a component returns `ErrorCode=ResetRequired` in response to SPDM requests when the component requires a reset to generate new certificates or keys. A Requester can also force a Responder to restart device authentication at any time by sending a `VERSION` command.

186 8.9.2 Key protection

187 When using SPDM, the device key pair forms the foundation for proof of identity, and the device private key must be protected from disclosure to an unauthorized party. A component should ensure that the device key pair cannot be accessed, regenerated, or replicated if an attacker gains access to the component. The protection mechanisms should protect the secret values from access through debug ports, an API, or other interfaces.

188 Some items that the component should protect are:

- The basis of the component identity, such as the UDS.
- The device private key.
- Any values that were used to derive or store other protected values, such as a key encryption key for the device private key.
- When processing the SPDM specified Key Schedule, a component should erase input key material, such as `Salt_1` and the handshake secrets, as soon as they are no longer needed.

189 When the device private key is in plaintext form, it should only be stored in the internal memory of the component. To protect the device private key, the component should clear it from memory as soon as it is no longer needed. A component can use non-volatile memory to store its device private key, but the non-volatile memory should be protected against unauthorized access, including attempts to gain physical access to the non-volatile memory, such as removing a flash part.

190 Any session keys should be protected from external observation and should be erased when no longer needed. Because the session keys typically exist during runtime, the protection should include protection against reads from a debug facility and reads through an API.

191 This protection can be implemented through a hardware mechanism that prevents unauthorized access. If the device

key pair storage is protected through encryption, the encryption key must not be one of the device keys because this violates the NIST SP800-57 requirement that a key is used for only one purpose.

192 The device should provide adequate protection for the device private key in-use (secure signing) and at-rest (secure storage). The device private key should never be exposed in any form outside of the device trust boundary. The device private key should only be accessible to device hardware, immutable firmware, or a similarly protected layer for establishing additional certificate chains such as in the alias certificate model.

193 8.10 Alternatives to certificate chains

194 8.10.1 Pre-Shared Key

195 Components provisioned with a Pre-Shared Key might not require an asymmetric key pair or the use of X.509 v3 certificates. Because the use of a Pre-Shared Key requires that the Requester and Responder both have knowledge of the Pre-Shared Key, the Requester can use the Responder's knowledge of the Pre-Shared Key as proof of the Responder's identity.

196 8.10.2 Provisioned public key

197 As an alternative to certificates, an SPDM endpoint can support the ability to import a public key. This capability is reported by setting `PUB_KEY_ID_CAP=1` in the `CAPABILITIES` exchange. The use case for this capability includes enabling devices that are not able to manage X.509 certificates. In this mode, the Responder's public key is provisioned to the Requester (and vice versa for mutual authentication). Following is an example sequence for this provisioning process:

1. The Responder generates or is provisioned with a key pair. See Key provisioning for more details.
2. In a trusted environment, the Responder's public key is provisioned to the Requester. The means by which the Responder's public key is provisioned to the Requester is outside of the scope of the SPDM specification, and might use a component's private API.
3. After deployment, the Responder signs responses (when required) using the private key that corresponds to the public key that was provisioned to the Requester. To maintain security, the Responder must protect the private key, as noted in Key protection.

198 > **Note:** The previous provisioning step must occur in a trusted environment. Because the public key is not part of a certificate, which is endorsed by a trusted root certificate, the source of the public key cannot be programmatically verified. Instead, the security associated with the public key must be enforced through physical security. Vendors should also provide protections to ensure that once a public key has been provisioned, another one cannot be provisioned for the same purpose unless authorized to do so. Further, the user should ensure that all affected components are placed back in a trusted environment before any re-provisioning occurs.

199 **8.10.2.1 Public key provisioning details**

200 When a provisioned public key is used (`PUB_KEY_ID_CAP=1`), there are a number of considerations that are required for the Requester and Responder. These include:

- The process for determining whether the provisioning is happening in a trusted environment, or even the definition of such an environment, is out of scope for the SPDM specification.
- The process for provisioning the public key from one endpoint to another endpoint is out of scope for the SPDM specification. The expectation is that this step would be performed using a vendor defined API.
- The mechanism for determining whether a public key has been provisioned between a pair of endpoints is out of scope for the SPDM specification.
- A Requester might need additional information to locate the previously provisioned public key, such as information that can be used to identify the instance of the device. The two endpoints might use transport-specific identifying information for this purpose. For MCTP based implementations, the intent is to use the `UUID` from the `Get Endpoint UUID` command for this purpose.
- If a Responder supports mutual authentication, it behaves as a Requester when performing mutual authentication. This essentially describes a role reversal.

201 # 9 SPDM messages

202 ## 9.1 Compatibility between versions

203 ***Version encoding*** in the SPDM specification discusses the standard for determining whether changes are considered backwards compatible when determining whether a change causes a minor or major version update. This section provides additional discussion of the thought process behind this standard.

204 As the SPDM specification is a security specification, it is not reasonable to expect the SPDM specification to allow implementations that use different versions of the SPDM specification to interoperate without any modifications. Instead, the SPDM specification requires both the Requester and Responder to agree on the same major and minor versions in order to interoperate. This requirement can require a component to implement a solution that supports multiple versions of the SPDM specification, taking into account the behavioral differences between them.

205 Other than the `VERSION` exchange, the SPDM specification does not impose a requirement for backwards compatibility to previous specification versions (major or minor). A component vendor can choose to remove support for earlier versions of the SPDM specification for reasons of solution simplification or due to the vendor's security policy.

206 The SPDM specification might change computations and other operations between different minor versions of the specification. These changes are only allowed when the differences are dependent on the value in the `SPDMVersion` field. With this standard in place, an implementation might need to perform different operations depending on the SPDM specification version in use. See the following pseudo-code for an example of the type of operational difference that is considered acceptable under this standard.

```
/* compute a signature over input 'data' */
if (spdm_version == 0x10)
    spdm10_compute_signature(data);
else if (spdm_version == 0x11)
    spdm11_compute_signature(data);
```

207 The SPDM specification can add new values to bit fields and enumerations in newer minor versions though the existing values are retained (though possibly deprecated). The SPDM specification makes every effort to ensure bit-wise compatibility with previous versions to ease the implementation burden. Implementers should take care to use fields as defined. For instance, if an enumeration only provides 0 and 1 as possible values, an implementer should be careful not to use bit-wise operations with the field as future versions of the SPDM specification might expand the list of enumerated values to 0, 1, and 2.

208 The SPDM specification makes every effort to ensure bit-wise compatibility between errata versions and the base version that is modified by the errata release. However, there may be bit-wise changes when required to fix a security issue or due to a mistake in the base version or a previous errata release. Correctly setting the `UpdateVersionNumber` field in the `VERSION` exchange ensures that both the Requester and Responder have enough information to correctly determine when behavioral differences may exist. One notable example is that several capabilities flags were added

to the `CAPABILITIES` response in version 1.2.1 of the SPDM specification. While a Requester that supports v1.2.0 and a Responder that supports v1.2.1 are compatible, the Requester does not have knowledge of the new capabilities flags and might need to try the commands and process a failure to discovery whether the Responder supports the capabilities.

209 The SPDM specification can add functionality to fields that were reserved in previous minor versions. Because reserved fields are defined as being set to 0, newer minor versions of the SPDM specification can safely add functionality to reserved fields, using the value of 0 to indicate previous behavior. The following guidelines apply to reserved fields:

- A component always sets reserved fields to 0.
- Do not check the contents of reserved fields. The SPDM specification states that the contents of reserved fields are ignored by the receiver, which means that a receiver does not generate an error when a reserved field contains a non-zero value.
- Do not modify the contents of a reserved field, as this changes transcript hashes.

210 This behavior accommodates cases where a component that supports multiple minor versions of the SPDM specification might fill in information in reserved fields while operating at less than its highest supported minor version number, thus simplifying implementations.

211 Functionality that is no longer recommended for use is marked as deprecated. A component might receive a message with a value in a deprecated field, and the component can either process the message properly or return an error. Field and value definitions associated with deprecated items are not reused within minor revisions of the same major version.

212 9.2 Message details

213 9.2.1 GET_VERSION and VERSION exchange

214 The `VERSION` exchange creates an agreement between the Requester and the Responder on the major and minor SPDM version that they use for subsequent messages. The `VERSION` exchange remains backwards compatible in all future versions of SPDM.

215 A Requester must not issue commands or include parameters that the Responder does not support. The supported command and parameter set is determined by the agreed SPDM version and the Requester’s and Responder’s supported capabilities.

216 The SPDM specification does not mandate that the list of supported versions, returned in a `VERSION` response, be unique or in any particular order. However, it is recommended that a Responder implementation return a list that does not contain duplicate entries and is ordered from highest to lowest supported version. It is also recommended that a Requester implementation make no assumptions about the order or uniqueness of the supported versions returned by a Responder.

217 ### 9.2.2 GET_CAPABILITIES and CAPABILITIES exchange

218 The `CAPABILITIES` exchange enables a Requester to query the SPDM capabilities that the Responder supports. The goals of the message exchange are:

- Enable a Requester and Responder to discover which optional message exchanges and capabilities the Responder and Requester supports
- Allow a Responder to inform the Requester of its cryptographic timeout requirements

219 The `CTExponent` enables a Responder to return its required cryptographic operation time. Because cryptographic operations can take longer than a non-cryptographic exchange, `CTExponent` enables the cryptographic timeout to respond to the needs of the individual Responder. Because the SPDM supports a variety of component types, the `CTExponent` values for separate components in a system can vary greatly.

220 A Requester only issues commands that the Responder supports, with the supported command set determined by the agreed SPDM version and the Requester's and Responder's supported capabilities.

221 Per the `CAPABILITIES` flags, most commands in the SPDM specification are optional. These commands are optional to allow implementation flexibility for Responders. The Requester has responsibility to ensure that the Responder supports enough optional commands to satisfy the Requester's security policy.

222 The `GET_CAPABILITIES` request can be issued with `Param1` set to 0 to exchange capabilities with the Responder, which is the legacy behavior for `GET_CAPABILITIES`. If `Param1` is set to 1, the Responder returns its Supported algorithms block, if supported by the Responder.

223 #### 9.2.2.1 CAPABILITIES flags

224 This clause provides background information on each of the optional capabilities in the `Flags` field in the `CAPABILITIES` response message.

225 Table 7 — Optional Flag field capabilities describes the optional capabilities in the `Flags` field in the `CAPABILITIES` response message:

226 **Table 7 — Optional Flag field capabilities**

Capability	Description
`CACHE_CAP`	If the Responder can cache certain messages through a reset, the Requester might skip issuing the cached requests after a reset and instead rely on cached values. If a Responder that sets `CACHE_CAP=1` has invalidated or lost its cached values, it responds to the next request, other than `GET_VERSION`, with an `ERROR` of `RequestResynch`, which indicates to the Requester that it is required to restart from `GET_VERSION`. See CACHE_CAP flag for more details.
`CERT_CAP`	`GET_DIGESTS` and `GET_CERTIFICATE` requests are related to each other. If a Responder supports `CERT_CAP`, it should also support `CHAL_CAP` and/or `MEAS_CAP`.

Capability	Description
CHAL_CAP	Indicates support for CHALLENGE. Support for the CHALLENGE exchange is optional because a Responder might not support the cryptographic operations or other capabilities required for the CHALLENGE_AUTH response. A Requester might support a standalone CHALLENGE or use MEASUREMENTS to accomplish a challenge. However, Requesters should remember that if a Requester sends a GET_MEASUREMENTS without first completing a CHALLENGE exchange, the transcript is nullified and the Requester does not know whether an entity altered the response data.
MEAS_CAP	Indicates support for MEASUREMENTS. Support is optional because a Responder might not support the cryptographic operations or other capabilities required for the MEASUREMENTS response. A Requester might either support a standalone CHALLENGE or use MEASUREMENTS to accomplish a challenge operation.
MEAS_FRESH_CAP	Indicates whether the Responder supports the ability to recompute measurements in response to a GET_MEASUREMENTS request. The value of this capability can influence the Requester's policy. A device that does not support fresh measurements must be reset to capture new measurements.
ENCRYPT_CAP	Indicates support for encryption. Requires either PSK_CAP or KEY_EX_CAP so that keys can be established for the secure session. Use of ENCRYPT_CAP requires use of MAC_CAP when using SPDM Secured Messages.
MAC_CAP	Indicates support for authenticated messages. Requires either PSK_CAP or KEY_EX_CAP so that keys can be established for the secure session. Can be used with ENCRYPT_CAP.
MUT_AUTH_CAP	Indicates support for mutual authentication. If set, it requires support for encapsulated requests. Note, mutual authentication requires either certificate support (CERT_CAP = 1) or a provisioned public key (PUB_KEY_ID_CAP = 1).
KEY_EX_CAP	Indicates support for key exchange, which is used with ENCRYPT_CAP or MAC_CAP.
PSK_CAP	Indicates support for Pre-Shared Key. Pre-Shared Key enables the use of Secured Messages by less capable devices. If supported, ENCRYPT_CAP or MAC_CAP are set.
HANDSHAKE_IN_THE_CLEAR_CAP	If set, the Responder can only send and receive SPDM defined messages without encryption and message authentication during the Session Handshake Phase. Whether a Requester accepts a Responder that does not set this bit is a function of the Requester's security policy.
PUB_KEY_ID_CAP	If set, the public key of the Responder was provisioned to the Requester using a mechanism that is out of scope for the SPDM specification.
CHUNK_CAP	Indicates that the component supports SPDM Large message transfers.
ALIAS_CERT_CAP	Indicates that the Responder uses the AliasCert model. Only the Responder sets this capability. A Responder is assumed not to have the ability to dynamically change its certificate chains between the DeviceCert and AliasCert models, so the ALIAS_CERT_CAP capability bit exists so that the Responder can inform the Requester of the model of certificate chain that the Responder will return.

Capability	Description
`SET_CERT_CAP`	Indicates that the Responder supports the `SET_CERTIFICATE_RSP` exchange. Only the Responder sets this capability, as it must inform the Requester of whether a `SET_CERTIFICATE` command will work.
`CSR_CAP`	Indicates that the Responder supports the `CSR` exchange. Only the Responder sets this capability, as it must inform the Requester of whether a `GET_CSR` command will work.
`CERT_INSTALL_RESET_CAP`	Indicates that the Responder might require a reset between the installation of a new certificate chain and the use of that certificate chain. Only the Responder sets this capability, and is intended to be a hint regarding what might happen when a certificate chain is installed. Using this hint, a Requester can plan its certificate chain installation policy appropriately, such as during a scheduled maintenance window, rather than being surprised by receiving an `ERROR` message of `ErrorCode=ResetRequired` following a `SET_CERTIFICATE` request (which is the expected behavior).
`EP_INFO_CAP`	Indicates that the component supports the `ENDPOINT_INFO` exchange. This capability can be set by both the Requester and the Responder, as the Responder might send the `GET_ENDPOINT_INFO` request as part of mutual authentication.
`EVENT_CAP`	Indicates support for the SPDM Event notification mechanism.
`MEL_CAP`	Indicates that the Responder supports measurement extension logs. Only the Responder sets this capability, though the Requester makes the ultimate determination whether it will request a measurement extension log.
`MULTI_KEY_CAP`	Indicates the component's level of support for Multiple Asymmetric Keys. This capability can be set by both the Requester and the Responder, as the Responder might encounter multiple keys on the Requester during mutual authentication.
`GET_KEY_PAIR_INFO_CAP`	Indicates that the Responder supports the `KEY_PAIR_INFO` exchange. Only the Responder sets this capability.
`SET_KEY_PAIR_INFO_CAP`	Indicates that the Responder supports the `SET_KEY_PAIR_INFO_ACK` exchange. Only the Responder sets this capability.

227 #### 9.2.2.2 CACHE_CAP flag

228 ##### 9.2.2.2.1 Multiple caching Requesters

229 For components that support `CACHE_CAP`, the support of a cached `Negotiated State` requires the component to be able to distinguish between Requesters so that it can correctly associate the cached `Negotiated State` with the appropriate Requester. Per the SPDM specification, the `Negotiated State` is between a given Requester and Responder pair, and remains valid until the next issuance of `GET_VERSION` or until the Responder decides to delete the associated `Negotiated State`. The mechanism to identify that a request originated from a different Requester is out of scope for the SPDM specification because it might require information from the transport layer. Any implementation of such a mechanism is transport specific, but an example of a mechanism is that an MCTP-based

implementation can track the `Source Endpoint ID` associated with a state identifier (using a mechanism that is out of scope for the SPDM specification) and invalidate the cached `Negotiated State` on any request that originates from a different `Source Endpoint ID` . Note that implementations should take care to reliably identify devices across resets, especially on buses that re-enumerate themselves and might allocate different identifiers to devices after each reset.

9.2.2.2.2 Negotiated State validity

Support for `CACHE_CAP` requires both the Requester and Responder to manage the validity of the `Negotiated State` . Requesters and Responders should only save a `Negotiated State` after a successful `CHALLENGE` exchange. Prior to a successful `CHALLENGE` exchange, a `Negotiated State` is subject to attack.

After a `Negotiated State` has been established, a Requester should take steps to detect a firmware update on the Responder. If the Requester detects a firmware update, the Requester should invalidate the current `Negotiated State` , issue the `GET_VERSION` request through `CHALLENGE_AUTH` request, and establish a new `Negotiated State` .

9.2.3 NEGOTIATE_ALGORITHMS and ALGORITHMS exchange

The `ALGORITHMS` exchange enables the Requester and Responder to agree on the cryptographic algorithms that the components use for subsequent exchanges. The Responder should select the strongest algorithms that the Requester provides. After the `ALGORITHMS` exchange is complete, the Requester and Responder have an agreed set of algorithms to use in subsequent message exchanges. Certain values in the response message depend on fields in the `CAPABILITIES` exchange.

The extended `ExtAsym` and `ExtHash` algorithm fields in the `ALGORITHMS` exchange enable expansion to additional algorithms to meet custom requirements. The Requester and Responder should prefer the `BaseAsymAlgo` and `BaseHashAlgo` fields if they can agree on them.

If the Responder has set `CERT_CAP=1` and/or `CHAL_CAP=1` , the Responder must select algorithms that correspond to a certificate chain that the Responder possesses. To ensure compatibility, the Requester should support a variety of algorithms.

9.2.3.1 Use of BaseAsymAlgo and ReqBaseAsymAlg

The SPDM specification defines two fields in the `ALGORITHMS` exchange that are similar, but serve different purposes. The `BaseAsymSel` and `ExtAsymSel` fields specify the asymmetric algorithm used for signature generation from the Responder to the Requester. The `ReqBaseAsymAlg` algorithm structure defines `AlgSupported` and `AlgExternal` fields that specify the asymmetric algorithm used for signature generation from the Requester to the Responder during mutual authentication.

9.2.3.2 Supported algorithms block

In the nominal flow for the `ALGORITHMS` exchange, the Requester sends its algorithm capabilities to the Responder, and then the Responder selects which of the offered algorithms the pair of endpoints should use. The assumption in this use case is that a Responder will select the strongest algorithms from the set offered. However, there is no enforcement of the Responder’s selection. A Responder might select algorithms that it can process efficiently, rather

than selecting the strongest, or the Responder implementer might have a different opinion than the Requester implementer has of which algorithm is the strongest. The Supported Algorithm Block was added to SPDM to address these cases.

241 The Supported Algorithms Block is requested by issuing a `GET_CAPABILITIES` request with `Param1` set to 1. The Responder uses the Supported Algorithms Block to return all of the algorithms that it supports, which can be used by the Requester to restrict the algorithms offered to the Responder to algorithms that the Responder supports and algorithms that meet the Requester's security policy.

242 The Supported Algorithms Block reuses the `NEGOTIATE_ALGORITHMS` request message format, returning `Param1` through the end of the request data, inclusive. The existing `NEGOTIATE_ALGORITHMS` request message format was reused rather than creating a new message format because `NEGOTIATE_ALGORITHMS` has all of the fields that are required for the Supported Algorithms Block, is an existing structure that may lead to efficiency through reuse of code, and the `NEGOTIATE_ALGORITHMS` request is already defined to allow multiple bits in each bit mask to be selected.

243 9.2.4 GET_DIGESTS and DIGESTS exchange

244 The `DIGESTS` exchange enables the Requester to retrieve the digests (hashes) of the certificate chain(s) stored on the Responder. The Requester can use the `DIGESTS` exchange to determine whether the certificate chain(s) stored on the Responder have changed. The Requester should store at least the public key from the leaf certificates along with the digest(s). The Requester can use the `DIGESTS` exchange as a shortcut to skip the retrieval of individual certificate chains, as the retrieval process can be slow on slower interfaces.

245 The `DIGESTS` response is not signed, so it is susceptible to replay attacks. It should be followed with a `CHALLENGE` or `GET_MEASUREMENTS` command to ensure that the Responder knows the private key.

246 9.2.5 GET_CERTIFICATE and CERTIFICATE exchange

247 The `CERTIFICATE` exchange enables a Requester to retrieve one or more certificate chains from the Responder. The `CERTIFICATE` response is potentially very large so a Requester might use the `Offset` and `Length` fields in the `GET_CERTIFICATE` request to issue multiple requests.

248 9.2.5.1 GET_CERTIFICATE and GET_DIGESTS in a session

249 The Requester is allowed to send both `GET_DIGESTS` and `GET_CERTIFICATE` requests during a session, as well as during session establishment. Sending one or both of these commands during a session is helpful when a Responder takes an action (such as a firmware update) that changes the contents of one or more certificates, but does not reset the session as a result. By reading the certificate chain from the Responder, the Requester has the information needed to validate the `CHALLENGE_AUTH` and `MEASUREMENTS` responses from the Responder.

250 9.2.6 CHALLENGE and CHALLENGE_AUTH exchange

251 The `CHALLENGE` exchange enables the Requester to ensure that the Responder knows the private key associated with a certificate chain. The `CHALLENGE` request and `CHALLENGE_AUTH` response contain several fields of note:

- Both the request and response messages contain `Nonce` fields, to protect against replay and chosen message attacks.
- The response contains a `CertChainHash` field, which the Requester can use to refute the `DIGESTS` or `CERTIFICATE` response.
- The response might contain a `MeasurementSummaryHash` field, which is a measurement of the concatenation of all elements of the TCB for the Responder.
- The response might also contain a MeasurementSummaryHash field, which is a measurement of the concatenation of all measurement blocks for the Responder, which may include measurement manifests. This applies to the request where the Param2 value was set to the value 0xFF.
- The `OpaqueDataLength` and `OpaqueData` fields are intended to be defined by a binding specification. The specific location of these fields ensures that they are included in the `CHALLENGE_AUTH` signature.
- The `Signature` field is generated according to the signature-generation process in the `CHALLENGE_AUTH` signature generation clause of the SPDM specification. The goal of the signature is to show that the Responder is the entity that has been responding to the Requester for earlier message exchanges, and that the Responder knows the private key associated with the public key in the leaf certificate of the certificate chain.

252 Although the use of `Nonce` fields in both the `CHALLENGE` request and the `CHALLENGE_AUTH` response messages protects against replay attacks, an adversary with physical access to the component can leverage the fact that a component responds to any correctly formed `CHALLENGE` with a signed response to perform side channel analysis, chip-clip attacks, or similar approaches to extract the component's private key.

253 Mitigations to such concerns should be applied at the implementation level, for example through steps such as those that the Key protection clause discusses. The SPDM protocol can require that request messages are authenticated, that is signed, as an additional protection for this class of threats. However, this requirement results in a significantly more complex protocol overall, increases message overhead unnecessarily in cases where Requester authentication is not supported, such as feature-limited Responders, and, ultimately, does not prevent adversaries who can produce `CHALLENGE` messages signed by a certificate chain trusted by the Responder from pursuing such avenues of attack.

254 9.2.6.1 Unique MeasurementSummaryHash

255 To prevent a potential length extension attack, a Responder should ensure that each `MeasurementBlock` used in a `MeasurementSummaryHash` is unique from any other `MeasurementBlock` in the given `MeasurementSummaryHash`. This applies to all uses of `MeasurementSummaryHash`. The exposure to a potential length extension attack is only in cases where the Requester does not issue `GET_MEASUREMENTS` and instead relies on the `MeasurementSummaryHash` alone to determine the state of the Responder.

256 9.2.6.2 Components in TCB

257 Requester may request the combined hash of measurements returned in the `MeasurementSummaryHash` field of the `CHALLENGE_AUTH` response to be a combined hash of the measurements of all measurable components or only those considered to be in the TCB.

258 What measurable components are in the TCB of a Responder is not negotiated between Requester and Responder and thus is implementation specific.

259 The intent of the `MeasurementSummaryHash` for the Requester is to check this value against the expected summary hash provided through a reference manifest or other out-of-band information.

260 Examples of data that may be the subject of TCB measurements:

- Fuse settings
- Strap configuration
- Manufacturing defaults
- Any other data the Responder considers to be part of the TCB

261 The reference manifest is the golden source of measurements, namely, its summary hashes and other data that can be fetched from the Responder device. Measurements that are retrieved from the Responder are usually verified against data provided in the reference manifest. The reference manifest is provided to the Requester by trusted means that are out of scope for SPDM specifications.

262 9.2.7 GET_MEASUREMENTS and MEASUREMENTS exchange

263 The `MEASUREMENTS` exchange enables the Requester to query the measurements of the firmware, the software, or configuration of a Responder.

264 In the `GET_MEASUREMENTS` request, the signature is optional. In some cases, Responders might not be able to create signatures, but can still return measurements. A Requester might refuse to operate with a Responder that does not support signed measurements. When specified, the `MEASUREMENTS` response is signed, showing that the Responder originated all `MEASUREMENTS` responses and has knowledge of the private key that is associated with the public key in the leaf certificate of the specified certificate chain.

265 The `MEASUREMENTS` exchange is designed to work with measurements of static data, which is data that does not change except in response to a user action. The `MEASUREMENTS` exchange does not handle measurement of dynamic values that can change without user action, such as the speed of a fan.

266 If a Responder that does not support measurement (`MEAS_CAP=0`) receives `GET_MEASUREMENTS` , or, if the requested measurement index is invalid, then the Responder sends an `ERROR` response with error code `InvalidRequest` to the Requester. If the Responder supports measurement and the requested index is valid, but the Responder is in a state (such as during boot) that the requested measurement is not available, then the Responder sends an `ERROR` response with error code `Busy` to the Requester.

267 9.2.7.1 Summary measurements

268 The `MEASUREMENTS` exchange does not support a mechanism to request a summary measurement option, meaning that there is not a mechanism to request that a Responder hash together all of its measurements and return a single hash of those measurements. A Requester might want to implement a summary measurement mechanism on its own to periodically check for changes in the underlying measurements, such as firmware configuration changes that happen outside of the purview of Requester. Another use case for a summary measurement mechanism is to monitor a component for firmware updates that happen outside of the purview of the Requester, though a firmware update and component reset also causes the component to return `ErrorCode=RequestResynch` . Note, periodic polling for measurements and use of summary measurements are optional behaviors.

269 If a Requester requires a summary measurement capability, the Requester should assemble its own summary measurement from the `MEASUREMENTS` responses from a given Responder. The Requester can check the stored summary by issuing one or more `GET_MEASUREMENTS` requests, regenerating the summary measurement, and checking the new summary measurement against the previous summary measurement.

270 In addition, if a Requester already knows the expected summary of the Responder's TCB or all measurements, then the Requester can retrieve the summary through the `MeasurementSummaryHash` field in the `CHALLENGE_AUTH`, `KEY_EXCHANGE_RSP`, or `PSK_EXCHANGE_RSP` response. By doing so, the Requester can avoid sending the `GET_MEASUREMENTS` request.

271 If the Responder supports measurements but is in a state (such as during boot) when some measurements are not available, then the Responder sends an `ERROR` response with error code `Busy` to the Requester.

272 9.2.7.2 Firmware debug indication

273 The `MEASUREMENTS` response includes a mechanism to return a measurement of firmware configuration. If a component typically operates in a mode that restricts debug access, it is recommended that the component use at least one measurement to indicate whether debug restrictions are in place. In this case, the component should alter a firmware configuration measurement when it enters debug mode. This measurement should remain altered until the component is reset. If the user subsequently disables debug mode, the component should continue to report an altered firmware configuration measurement until reset to ensure that a Requester can detect a case where a debug capability has been enabled and disabled before the Requester can detect it. The measurement index and definition of any debug mode measurement is vendor specific.

274 Starting with version 1.2, the SPDM specification defines a measurement data structure that a component can use for debug and mode indication. The use of a standardized mechanism has additional benefits as it can be interpreted by a Requester without the use of vendor unique data structures. The standardized device mode indication is modeled on the above paragraph and is meant to convey the information described above.

275 9.2.7.3 MEASUREMENTS only components

276 Some components might only support the `MEASUREMENTS` capability, but not support the ability to sign the measurements. Such a component sets `CERT_CAP=0`, `CHAL_CAP=0`, and `MEAS_CAP=1` in the `CAPABILITIES` response message. This capabilities configuration is desirable in some cases, such as in a component with minimal processing capabilities. If a component like this exists, a Requester should carefully consider whether to trust the measurement that is returned by the Responder.

277 9.2.7.4 Use of RawBitStreamRequested

278 The `RawBitStreamRequested` bit in the `GET_MEASUREMENTS` request message requests that a Responder return a raw bit stream. The Responder might return a digest instead of a raw bit stream for a variety of reasons, including if the raw bit stream contains sensitive information, the raw bit stream is not available, or the raw bit stream exceeds the maximum measurement block size of 64 KiB.

9.2.8 Hash-extend measurement and measurement extension log

The Responder may maintain a measurement extension log (MEL) that records events, configurations, and other data of the Responder. The Responder may maintain a hash-extend measurement (HEM) that is the result of extending entries in the MEL. The Responder reports the HEM to the Requester in a MEASUREMENTS response message. The Responder reports the MEL to the Requester in a MEASUREMENT_EXTENSION_LOG response message.

The purpose of the HEM is for the Requester to verify the integrity of the MEL. The integrity of the MEL may also be protected by other means, such as sending the MEASUREMENT_EXTENSION_LOG message within a secure session or in a trusted environment. In these cases, the Responder does not have to support MEASUREMENTS in order to support MEL.

9.2.9 Encapsulated request flows

In certain use cases, such as mutual authentication, the Responder needs the ability to issue its own SPDM request messages to the Requester. Certain transports prohibit the Responder from asynchronously sending out data on that transport. Message encapsulation, which preserves the roles of Requester and Responder as far as the transport is concerned but enables the Responder to issue its own requests to the Requester, addresses cases like these.

The `GET_ENCAPSULATED_REQUEST` and `DELIVER_ENCAPSULATED_RESPONSE` request messages, (`ENCAPSULATED_REQUEST`) and `ENCAPSULATED_RESPONSE_ACK` response messages facilitate the encapsulated request flow.

The encapsulated requests flow is used in limited scenarios, such as mutual authentication, and cannot be used for general purpose SPDM message encapsulation. Only certain requests and their corresponding responses, including `ERROR` , can be encapsulated. For details, see DMTF DSP0274.

9.2.10 Session establishment

This section discusses the Session-Secrets Exchange process.

9.2.10.1 OpaqueData handling

The `OpaqueData` field is used to exchange information about the session being established. The `KEY_EXCHANGE` message is defined in DSP0274, while the contents of the `OpaqueData` field are defined in DSP0277. The contents of the `OpaqueData` field may grow over time. When a component is processing the contents of the `OpaqueData` field, it should ignore any field that it does not recognize, as it may come from a component that supports a newer version of DSP0277.

9.2.11 Secure session messages

A number of capabilities `Flags` are related to managing secure sessions, and many of the capabilities are used in conjunction with each other. The Secured Messages-related capabilities `Flags` are:

- `ENCRYPT_CAP`
- `MAC_CAP`
- `MUT_AUTH_CAP`
- `KEY_EX_CAP`
- `PSK_CAP`
- `ENCAP_CAP`
- `HBEAT_CAP`
- `KEY_UPD_CAP`
- `PUB_KEY_ID_CAP`

292 Many of the capabilities `Flags` have dependencies on each other, which are explained in the SPDM specification. One dependency relationship of note is that the use of `ENCRYPT_CAP` requires the use of `MAC_CAP`. SPDM Secured Messages that use encryption require the use of message authentication. Note that, while the SPDM specification allows for encryption-only sessions, the use of such messages can result in a receiver decrypting messages from an attacker and are not recommended for most use cases. SPDM Secured Messages avoids these issues by requiring `MAC_CAP` to be set when `ENCRYPT_CAP` is set.

293 This section ignores any potential use of Pre-Shared Keys.

294 Another dependency of note is between `HANDSHAKE_IN_THE_CLEAR_CAP`, `ENCRYPT_CAP`, and `MAC_CAP`. When any of these capabilities are set, `KEY_EX_CAP` must also be set. These capabilities affect the behavior of a secure session, during the session handshake phase as well as the application phase. The `MAC_CAP` flag is not the same as the `HMAC` in the `RequesterVerifyData` or `ResponderVerifyData` fields of Session-Secrets-Exchange and Session-Secrets-Finish messages. In the context of SPDM secured messages, `HMAC` provides key confirmation during the key exchange process. Message authentication is provided by the use of AEAD when a secured message is constructed.

295 Table 8 — Combinations of secure session capabilities summarizes the interaction between these capabilities and the secure session phases.

296 **Table 8 — Combinations of secure session capabilities**

Secure Session Phase	`HANDSHAKE_IN_THE_CLEAR_CAP`	`ENCRYPT_CAP`	`MAC_CAP`	Description
Session Handshake Phase	0	0	0	Invalid Combination for secure sessions. One of `ENCRYPT_CAP` or `MAC_CAP` must be set if `HANDSHAKE_IN_THE_CLEAR_CAP` is not set and `KEY_EX_CAP` is set. This is invalid since the requester is indicating that it requires handshake messages to be encrypted and/or authenticated but does not support either capability.

Secure Session Phase	HANDSHAKE_IN_THE_CLEAR_CAP	ENCRYPT_CAP	MAC_CAP	Description
	0	0	1	Valid Combination. Messages in the Session Handshake Phase such as FINISH, FINISH_RSP are authenticated but are not encrypted.
	0	1	0	Valid Combination in DMTF DSP0274. Messages in the Session Handshake Phase such as FINISH, FINISH_RSP are encrypted but are not authenticated. When using SPDM Secured Messages, as defined in DMTF DSP0277, this combination is not allowed.
	0	1	1	Valid Combination. Messages in the Session Handshake Phase such as FINISH, FINISH_RSP are both encrypted and authenticated.
	1	0	0	Invalid Combination. One of ENCRYPT_CAP or MAC_CAP must be set if KEY_EX_CAP is set. ENCRYPT_CAP and MAC_CAP have no effect during the session handshake phase, however these capabilities do affect the application phase, and at least one of them must be set.
	1	0	1	When HANDSHAKE_IN_THE_CLEAR_CAP is requested, ENCRYPT_CAP and MAC_CAP have no effect. Messages in the Session Handshake Phase are neither encrypted nor authenticated.

Secure Session Phase	HANDSHAKE_IN_THE_CLEAR_CAP	ENCRYPT_CAP	MAC_CAP	Description
	1	1	0	When HANDSHAKE_IN_THE_CLEAR_CAP is requested, ENCRYPT_CAP and MAC_CAP have no effect. Messages in the Session Handshake Phase are neither encrypted nor authenticated.
	1	1	1	When HANDSHAKE_IN_THE_CLEAR_CAP is requested, ENCRYPT_CAP and MAC_CAP have no effect. Messages in the Session Handshake Phase are neither encrypted nor authenticated.
Application Phase	Not Applicable	0	0	Invalid Combination. One of ENCRYPT_CAP or MAC_CAP must be set if KEY_EX_CAP is set.
	Not Applicable	0	1	Valid Combination. Messages in the Application Phase are authenticated but are not encrypted.
	Not Applicable	1	0	Valid Combination in DMTF DSP0274. Messages in the Application Phase are encrypted, but not authenticated. When using SPDM Secured Messages, as defined in DMTF DSP0277, this combination is not allowed.
	Not Applicable	1	1	Valid Combination. Messages in the Application Phase are both encrypted and authenticated.

297 #### 9.2.11.1 Handling of Heartbeat disabled

298 There might be cases where the Responder sets the `HeartbeatPeriod` to 0. The behavior in this case depends on the `HBEAT_CAP` field.

299 If the Requester and Responder both set the `HBEAT_CAP` field to 1 in the `CAPABILITIES` exchange and set the `HeartbeatPeriod` to 0, they are indicating that they support heartbeats, but do not use a defined period for the

heartbeat. This might occur when the Responder does not have sufficient resources (such as watchdog timers) to support a timed heartbeat. In such a case, if a Requester sends a `HEARTBEAT` request anyway, the Responder can send a `HEARTBEAT_ACK` response.

300 If the Requester and Responder do not set the `HBEAT_CAP` field to 1 in the `CAPABILITIES` exchange, then they are expected to set the `HeartbeatPeriod` to 0. This indicates that heartbeats are not supported for this session. In this case, if a Responder receives a `HEARTBEAT` message, then it can return an `ERROR` response with `ErrorCode=UnexpectedRequest`, or it can silently discard the request.

301 9.2.11.2 Session timeout

302 A Responder is likely to have limited resources to manage sessions. A Responder can impose policies on the management of session resources based on vendor defined policies. A vendor could consider the following conditions:

- A timeout policy for when the Heartbeat period expires.
- A policy for handling timeouts during the session handshake.

303 9.2.11.3 Mutual authentication required capabilities

304 Mutual authentication depends on support for `KEY_EXCHANGE` (`KEY_EX_CAP` = `1`) or `PSK_EXCHANGE` (`PSK_CAP` is non-zero), as mutual authentication is started during the processing of these commands. Mutual authentication is also only possible if the components support certificates or a provisioned public key. Therefore, if the components do not support at least one of these capabilities, mutual authentication will fail. In such a configuration, one component could raise an error on receiving the `CAPABILITIES` response. A component could also send an `ERROR` response to the `KEY_EXCHANGE` request, possibly with the `ErrorCode` = `InvalidPolicy`.

305 9.2.11.4 Session-Secrets-Exchange collisions

306 During the Session-Secrets-Exchange (`KEY_EXCHANGE`), there is a very small chance that both components will generate the same public key. If this condition occurs, a component can choose to restart the key exchange for the session with a different random input, based on the implementer's policy.

307 9.2.12 VENDOR_DEFINED_REQUEST and VENDOR_DEFINED_RESPONSE exchange

308 The `VENDOR_DEFINED_RESPONSE` exchange enables a Requester and Responder pair to exchange information that the SPDM specification does not otherwise cover. A component vendor or another standards body can define request and response messages. For more information on implementations by other standards bodies, see Partner implementations.

309 9.2.13 RESPOND_IF_READY sequence

310 The `RESPOND_IF_READY` sequence allows for situations when the Responder cannot respond in a reasonable time.

The time to a final response, which fulfills a `RESPOND_IF_READY` request, is still bound by the timing parameters that the SPDM specification defines.

311 The design intent of the `RESPOND_IF_READY` sequence is to enable components to cooperate with a larger system while performing long operations, such as signing. One reason to use `RESPOND_IF_READY` during a long operation is to release a shared bus to enable other components to use the bus during the operation.

312 9.2.14 Certificate provisioning commands

313 The SPDM specification defines two commands for certificate provisioning. The first command in the sequence is `GET_CSR`, which retrieves a certificate signing request (CSR) from the Responder. After the CSR is signed, forming a signed certificate, the resulting certificate chain is sent to the Responder using the `SET_CERTIFICATE` request.

314 Another possible flow is to generate a certificate chain in an external environment without using a `GET_CSR` exchange. In this case, the Requester would send a `SET_CERTIFICATE` without a preceding `GET_CSR` exchange.

315 9.2.14.1 GET_CSR exchange

316 The `GET_CSR` command requests a CSR from the Responder. A Requester can provide DER formatted information to be included in the CSR using the `RequesterInfo` field. `RequesterInfo` is intended to allow the Requester to insert information that might not be available to the Responder.

317 Referencing the example in Figure 4 — Example certificate chain, the CSR returned by `GET_CSR` is for the Device Certificate from the `DeviceCert` example, or the Device Certificate CA from the `AliasCert` example.

318 The Responder signs the produced CSR using the associated private key. In some cases, the required private key is not available during normal operation, so the Responder may require a reset or some other action to allow it to generate a CSR. If the Responder requires a reset to complete the `GET_CSR` request, then it responds with `ErrorCode=ResetRequired`. If the Responder requires any other action to complete the `GET_CSR` request, it signals this action with a vendor defined `ErrorCode` or other status.

319 9.2.14.1.1 GET_CSR after reset

320 The use of a reset during a `GET_CSR` exchange creates additional complexity. For instance, the Responder might need to persist the `RequesterInfo` data across the reset. The Responder might also need to generate the CSR during the initialization process and store it for later retrieval.

321 After the Responder has completed its reset, the Requester resubmits the same `GET_CSR` as it did before the reset, and the Responder now returns the completed `GET_CSR` response.

322 Note, the method for resetting a Responder is out of scope for the SPDM specification.

323 9.2.14.1.2 Overlapping GET_CSR requests

324 Since a Responder might require a reset to process a `GET_CSR` request, then a situation could arise where a new `GET_CSR` request arrives while the Responder still has a reset pending from an earlier `GET_CSR` request. When SPDM version 1.2 is in use, the SPDM specification states that the Responder overwrites the earlier request with the

new request. Beginning in SPDM version 1.3, the `Overwrite` bit (Bit 7 of `Param2`) must be set to `1b` for the Responder to overwrite the earlier request with the new request. Otherwise, the Responder could choose to overwrite the request or respond with an `ERROR` message. This behavior is required to maintain backward compatibility with SPDM Version 1.2.

9.2.14.2 SET_CERTIFICATE exchange

After a certificate chain has been built, the Requester uses the `SET_CERTIFICATE` request to send the certificate chain to the Responder. The Responder component may include any number of validation checks for the certificate chain, and might return an `ERROR` response if it detects a problem with the certificate chain.

When a certificate chain is imported, a Responder might require a reset to allow the component to generate new alias certificates for the newly imported certificate chain. The Responder indicates the need for a reset prior to use of the newly imported certificate chain by responding to the `SET_CERTIFICATE` request with `ErrorCode=ResetRequired`. Once a Responder returns `ErrorCode=ResetRequired` to a `SET_CERTIFICATE` request, a Requester can attempt to import additional certificate chains. The Responder is allowed to return `ErrorCode=ResetRequired` to prevent additional `SET_CERTIFICATE` requests, or the Responder can accept additional `SET_CERTIFICATE` requests prior to the reset.

9.2.14.2.1 Slot write behavior

A Responder can apply policies to writes to one or more slots. A Responder may implement vendor specific policies around when slot contents can be written, overwritten, or write protected. For instance, a Responder might send an error when it receives a `SET_CERTIFICATE` request for any slot that already has contents. A Responder might also require the use of mutual authentication prior to accepting a `SET_CERTIFICATE` request from a given Requester. These policies are vendor specific and are out of scope for the SPDM specification.

The Responder can apply authentication policies for the payload of a `SET_CERTIFICATE` request. For instance, the Responder might check the public key of the leaf certificate to ensure that it matches the component's public key or ensure that the signatures in the certificate chain are correct. Such policies are vendor specific and are out of scope for the SPDM specification.

The Responder should consider the case where it loses power while writing a certificate chain to persistent memory. If a Requester receives a successful response to a `SET_CERTIFICATE` request, then the Requester would expect that a read of the same slot would return the new certificate chain. If the write of the certificate chain is interrupted before the response, then the Responder should ensure that subsequent reads of the slot return old contents, new contents, or show the slot as empty.

9.2.14.2.2 Slot write authorization

The SPDM specification states requirements and recommendations around when a `SET_CERTIFICATE` request can be sent. For instance, the SPDM specification states that `SET_CERTIFICATE` requests for slot 0 must be issued in a trusted environment, and that `SET_CERTIFICATE` requests for slots 1-7 should be issued in a trusted environment or using a secure session with mutual authentication.

The requirement to use a trusted environment for `SET_CERTIFICATE` requests for slot 0 is intended to support

component manufacturing. Since a secure session cannot be established without a certificate chain present on the Responder, the specification provides this route to load an initial certificate to support manufacturing. For slots 1-7, there are multiple routes possible, and the goal of the SPDM specification is to support these routes. One possible route to provision additional certificate chains is to return the component to a trusted environment to provision the certificate. Another route discussed by the SPDM specification is to use a secure session with mutual authentication, since a Responder should not accept a certificate chain from an unknown Requester. A Responder can impose additional vendor defined requirements for authorization.

335 #### 9.2.14.2.3 Trusted environment

336 The SPDM specification makes several references to a trusted environment, but the SPDM specification does not make any requirements on the properties of a trusted environment. In other contexts, this may be referred to as a secure environment. In principle, a trusted environment is one where the operator ensures that a bad actor cannot interfere with operations, so a component can be placed in a less restrictive state. Properties of a trusted environment might include:

- A physically secured facility with personnel access restrictions.
- Logging of operations and periodic audits of those logs.
- Assurance that all software is authentic and unaltered.
- Restrictions on, or elimination of, external network connections and any unnecessary ports on computers.

337 #### 9.2.14.2.4 Overlapping SET_CERTIFICATE requests

338 For Responders that support the `SET_CERTIFICATE` request, the Responder might encounter a situation where the Responder receives a `SET_CERTIFICATE` request while it is busy with another task. If the Responder is not able to process the `SET_CERTIFICATE` request due to a temporary condition, including for the reason that the Responder is processing another `SET_CERTIFICATE` request, the Responder's non-volatile storage is write-protected, or another condition, the SPDM specification states that the Responder sends an `ErrorCode=Busy` response.

339 ## 9.2.15 Event notification

340 Version 1.3 of the SPDM Specification adds support for events. SPDM events use the following high-level flow:

1. An endpoint determines that another endpoint supports being an Event Notifier using `EVENT_CAP` in the `CAPABILITIES` exchange.
2. The Event Recipient uses the `SUPPORTED_EVENT_TYPES` exchange to request a list of events that the Event Notifier is able to generate.
3. The Event Recipient uses the `SUBSCRIBE_EVENT_TYPES` request to subscribe to one or more events from the Event Notifier.
4. The Event Notifier uses `SEND_EVENT` request to send one or more event instances to the Event Recipient.

341 Note that an Event Recipient can also use `SUBSCRIBE_EVENT_TYPES` to modify or cancel an existing event subscription.

342 #### 9.2.15.1 Event transfer considerations

343 An Event Notifier is allowed to send a number of events in one `SEND_EVENT` request, and can even use an SPDM large message transfer to send events. However, the Event Notifier is recommended to send small events to ensure timely delivery and to reduce the likelihood of losing an event. For most Event Recipients, it is more important to receive an event close to the time that the reported condition occurs, rather than waiting and consolidating a large number of events into one transfer.

344 Some transports used for SPDM might not support bidirectional transfers. In such cases, the Event Notifier cannot send a `SEND_EVENT` request. Instead, the Event Recipient must poll the Event Notifier using `GET_ENCAPSULATED_REQUEST`. The Event Recipient should be aware that its polling interval will determine the maximum latency for event notifications, and too long of a polling interval can result in a large number of events being received per poll or events being lost due to buffer exhaustion. If the Responder receives a `GET_ENCAPSULATED_REQUEST` request, but does not have any requests to send, it responds with an `ERROR` message of `ErrorCode=NoPendingRequests`. Note that if a Requester and Responder are using a transport that does not support bidirectional transfers, then the components can only support `EVENT_CAP` if they also support `ENCAP_CAP`.

345 #### 9.2.15.2 Event retries

346 The Event Notifier should implement error handling to ensure a certain degree of reliability and security robustness. A typical form of error handling is resending the request up to a certain number of tries. When resending unacknowledged events, the Event Notifier can append new events if they occur as a transfer optimization. The Event Notifier can discard the event if it remains unacknowledged. If the Event Notifier discards an event, the SPDM Specification requires that the Event Notifier send an event lost event.

347 There can be scenarios where a large number of events remain unacknowledged, including the event lost event itself. When all other error handling methods cannot result in a successful acknowledgement, especially the acknowledgement of an event lost event, the Event Notifier can terminate the session.

348 #### 9.2.15.3 DMTF defined event types

349 The SPDM specification defines several event types. The only one that is mandatory is the `EventLost` event, which is used to manage the event subsystem. However, there are additional events defined that are intended to improve operation between a Responder and Requester, and are recommended for implementation. The DMTF defined events other than `EventLost` allow an Event Notifier to inform an Event Recipient of a change in a more timely manner than the alternative of waiting for the Event Recipient to discover the change on its own.

350 ### 9.2.16 Large message transfers

351 In version 1.2, the SPDM specification introduced a large message transfer mechanism. This mechanism provides a transport agnostic mechanism to send SPDM message assemblies that are larger than the maximum message size of the underlying transport. The SPDM large message transfer mechanism uses the word *chunk* to describe each portion of the transfer to keep the terminology distinct from similar terms in MCTP. This mechanism is used in place of adding message fragmentation support to each message that is potentially larger than a single transfer size.

9.2.16.1 Large message transfer parameters

The use of large message transfers is controlled by a set of parameters in the `CAPABILITIES` exchange. Both the Requester and Responder make their parameters known to the other endpoint. When sending a chunk, the underlying transport might still break each chunk into smaller packets.

The `DataTransferSize` field indicates the component's maximum buffer size for receiving a single complete SPDM message. Messages that are larger than `DataTransferSize` must be broken into chunks.

The `MaxSPDMmsgSize` field indicates the component's maximum buffer size for receiving a complete, large SPDM message. Even with a message broken into transfer chunks, the recipient of the message is limited to processing a message assembly that is no larger than `MaxSPDMmsgSize`.

`MaxSPDMmsgSize` should be provisioned to a sufficiently large size to handle all situations required for a component. For instance, if the Requester has a certificate chain that is 20 KB in size but the Responder limits `MaxSPDMmsgSize` to 16 KB, then there is no way for the Requester to send the certificate chain using `SET_CERTIFICATE`.

9.2.16.2 Large message ordering

A Requester may receive a response of `ErrorCode=LargeResponse`, indicating that the response is larger than the Responder's `DataTransferSize`. In a typical case, the next request from the Requester would be `CHUNK_GET` to start the large message transfer. However, the SPDM specification does not require the Requester to send `CHUNK_GET` as the next request, and the Requester can send another request message as long as it follows other SPDM specification defined sequencing rules. The Requester should understand that if it sends a request other than `CHUNK_GET`, the Responder might discard the buffered response for the message that triggered the response of `ErrorCode=LargeResponse`.

9.2.16.3 Large message reassembly

The process for tracking and reassembling SPDM large messages depends on two numbers.

The first number is the Handle, which is transmitted in the `Param2` field. The Handle is used to uniquely identify the SPDM large message being transmitted. The SPDM specification states that the Handle should be monotonically increasing or decreasing until it wraps. The intent of this guidance is to ensure that endpoints can avoid confusion between different SPDM large messages. In this case, the SPDM specification does not require a specific implementation because a variety of implementations are acceptable so long as they meet the intent to avoid confusion between SPDM large messages.

The second number is the `ChunkSeqNo`, which is the sequence number. The sequence number identifies the order of a set of messages that use the same Handle. The requirements around the sequence number are more strict, and the SPDM specification requires that the sequence number starts at 0 and is monotonically increasing. These requirements ensure that the receiver of the message chunks can correctly order the chunks in order to reassemble the message, even if they are delivered out of order, and can correctly detect missing chunks.

9.2.17 GET_ENDPOINT_INFO and ENDPOINT_INFO exchange

The `ENDPOINT_INFO` exchange is used to retrieve general information from a component. The command uses a `SubCode` to specify the operation, and supports signing of responses.

9.2.17.1 ENDPOINT_INFO DeviceClassIdentifier

The `ENDPOINT_INFO` `DeviceClassIdentifier` `SubCode` is used to retrieve identifying information from a component. The term Device Class Identifier is meant to convey that the returned identifier represents a large group of components, such as a model identifier. While most transports have a method for returning a device identifier (FRU data, PCI configuration cycles, etc.), a method may not be available in all cases. The Device Class Identifier provides a consistent approach that only depends on SPDM. The returned Device Class Identifier can be used to retrieve stored information about a component, or for other implementation specific use cases.

Rather than creating a new vendor registry, SPDM leverages existing registries as the *Registry or standards body ID* table in the SPDM Specification defines. The Device Class Identifier element includes a `SubordinateID` field that is used to carry additional identifiers in the namespace of the vendor ID that is in the `SVH` field. For instance, if the `SVH` field references the PCI-SIG as the Registry or standards body ID, the `VendorID` field would contain the vendor's PCI-SIG assigned Vendor ID. In this case, the `SubordinateID` field might contain the Device ID, Subsystem Vendor ID, and Subsystem Device ID fields, as the PCIe Specification defines. The format of the `SubordinateID` is not intended to be machine processed, but should be consistent to assist in the cache lookup use case.

The `ENDPOINT_INFO` response contains a `NumIdentifiers` field that contains the number of identifiers. Using this field, a component can return identifiers using multiple registries, though all of the reported identifiers should be appropriate for the component in question.

9.3 Message exchanges

The SPDM specification specifies ordering rules for message exchanges and the transcript hash that is generated from those message exchanges. To reduce the complexity associated with message sequencing, the SPDM specification defines valid sequences including options for use cases that cache certain responses.

During the SPDM message exchanges, the Requester can drop communication with a Responder if the Responder violates a policy that the Requester holds, such as when the Responder negotiates too low of a version or the Responder returns too many errors.

The SPDM specification defines some messages as optional, such as `CHALLENGE`, which permits a variety of implementation permutations. Ultimately, the Requester implementation controls the policy that it wants to use and the SPDM specification grants the Requester some degree of implementation latitude. For instance, a security-sensitive Requester might reissue all requests on every reset while a more permissive Requester might cache certificate digests and skip the `CHALLENGE` on each reset. The Responder should make no assumptions about the security policy of the Requester.

9.3.1 Multiple Requesters

374 The tracking for message sequences is on the basis of a Requester and Responder pair, and a Responder can optionally support tracking more than one Requester and Responder pair. If a Responder receives requests from Requesters A and B, for instance, the Responder must track message payloads for the successful message exchanges with both Requester A and Requester B. A Responder has limited resources for tracking message exchanges, and might take steps to both limit the number of supported Requesters and reclaim resources that it has used to track exchanges with a given Requester. The exact mechanisms to do so are out of scope of the SPDM specification.

375 If a Responder supports communication with only a single Requester at a time, the Responder does not need to track the Requesters because communication with a new Requester starts with the `GET_VERSION` request and causes the Responder to discard any existing tracked messages. This type of implementation can cause problems in complex environments due to constantly restarting message sequences.

376 For implementations that use an MCTP transport, the `MCTP Endpoint ID` is the recommended method for tracking the Requester (see DMTF DSP0275). For other binding specifications, the binding specification should document the Requester tracking method.

9.3.2 Message timeouts and retries

378 The ***Timing specification for SPDM Messages*** table in the SPDM specification lists a number of interrelated timeout values. The `RTT` value is the worst-case value for a message round trip based on the transport. The `RTT` value might be less than the `CT` value. If so, the Responder must respond with `ErrorCode=ResponseNotReady` within the RTT-specified time.

379 This mechanism ensures that Responders release the bus in a timely manner. After a Responder returns `ErrorCode=ResponseNotReady`, the Requester can issue a request to another Responder or wait for the time specified by `RDTExponent` and issue `RESPOND_IF_READY`. During this time, the Requester should not issue any request to the Responder other than `RESPOND_IF_READY`.

380 The SPDM specification allows for retries of messages after a timeout has occurred. In a retry scenario, a Requester retries the same request as before. Specifically, a retry of a `CHALLENGE` or `GET_MEASUREMENTS` request reuses the same nonce as the request that timed out so that the transcript hash calculation is not disrupted. A security-sensitive Requester can choose not to retry a request and instead return to `GET_VERSION` and restart the message sequence.

381 When a message is retried, the endpoints must ensure that the transcript hash is not updated until successful transmission of the message. One possible implementation is to hold current and next versions of the transcript hash. In this case, the value held in the current transcript hash would be maintained until a new message is received, at which point the transcript would move to the next transcript hash value. If the Requester retries the message instead, then the value held in the current transcript hash can be used as part of the retry.

9.3.2.1 RDT and CT interactions

383 When a Responder is processing a message that requires cryptographic processing, the timeout values of `CT` and

RDT can both play a role. When a Responder receives a request that requires a cryptographic operation, the CT based timer T2 is already running on the Requester. If the Responder returns an ERROR response with ErrorCode=ResponseNotReady, the response includes a value for RDT. This value will usually not match the time remaining in T2. In such cases, the Requester is recommended to wait RDT time before issuing a RESPOND_IF_READY request, rather than waiting for the remainder of T2. Since RDT is a runtime generated value, rather than a worst case value like CT, RDT is likely more accurate at the time that it is returned.

384 If the sum of the time elapsed waiting for a response plus the returned value in RDT exceeds CT, the Requester might timeout the request. However, since a Responder should return an RDT that is an accurate reflection of the time remaining to process the request, it is recommended that a Requester continue to wait for the response for the remaining amount of time specified in RDT.

385 When a Requester issues RESPOND_IF_READY, the Responder might continue to respond with an ERROR response with ErrorCode=ResponseNotReady and an updated value for RDTExponent. This can happen for a variety of reasons, including when a Responder has internal tasks queued and needs more time to process the request or due to a back-off algorithm. Whether a Requester continues to wait for a Responder that is requesting time beyond CT, and how long it waits in total, is implementation specific.

386 9.3.2.2 Message resource management

387 Certain SPDM protocol interactions involve the exchange of multiple messages, during which state information is maintained. For example, multiple GET_MEASUREMENTS messages might be issued in a sequence requesting individual unsigned measurements, with the Responder maintaining a message transcript to be signed at the end of the sequence. While individual requests and responses might be issued within the permitted timeout parameters, a malicious or buggy Requester might consume resources at the Responder by starting but never completing such multi-message interactions. This issue might be accentuated if a Responder interacts with more than one Requester in parallel, maintaining a number of active states. It is advised that SPDM implementations implement protections against such resource exhaustion scenarios by maintaining session limits, timeouts or similar mechanisms to detect and reset a misbehaving session when necessary. In this context, a session denotes an ongoing exchange of SPDM messages between a Requester and Responder pair.

388 9.3.2.3 Secured Messages retries

389 The *Secured Messages using SPDM Specification 1.0.0 (DSP0277)* indicates that it is permissible for a component to include the sequence number in a message to help the receiver process a retry or out of order delivery if the transport protocol does not provide a mechanism to reconstruct the proper message order. SPDM Secured Messages are based on *IETF TLS DTLS13-43*, which indicates that including the sequence number is not considered a potential attack vector because section 3 of *IETF TLS DTLS13-43* adds the sequence number to the datagram record.

390 9.3.2.4 Transcript management

391 The SPDM specification provides rules for processing ERROR responses during the construction of the L1 and L2 transcripts during a MEASUREMENTS exchange. Specifically, the endpoints reinitialize L1 and L2 on any ERROR response with an ErrorCode set to anything but except certain specified values. Note, this applies to

`ErrorCode` = `Busy`, so a busy response causes the transcript hash to be reset to null. A Requester that cares about maintaining a transcript across multiple `MEASUREMENTS` responses should restart the process if it receives a busy response. Responder implementers should be aware that busy responses may cause this behavior and be judicious in the use of these responses.

392 ## 9.4 Cryptography Endianness

393 ### 9.4.1 Endianness of digital signatures

394 The SPDM specification version 1.2 and later define the endianness of digital signatures for RSA, ECDSA, SM2, and EdDSA.

- RSA: big endian for `s`.
- ECDSA and SM2: big endian for `r` and `s`.
- EdDSA: big endian for `R` and little endian for `S`.

395 The SPDM specification version 1.0 and version 1.1 did not specify the endianness of the RSA and ECDSA digital signatures. Implementing the endianness as defined in the SPDM specification version 1.2 is recommended. In order to maximize compatibility, it is recommend that an implementation interpret the endianness of the signature first as big-endian and, if verification fails, then attempt to interpret it as little-endian.

396 ### 9.4.2 Endianness of key exchange data

397 The SPDM specification version 1.1 and later defines the endianness of key exchange data for FFDHE, ECDHE, and SM2_P256.

- FFDHE: big endian for `Y`.
- ECDHE and SM2_P256: big endian for `X` and `Y`.

398 ### 9.4.3 Endianness of AEAD IV

399 The *Secured Messages using SPDM Specification (DSP0277)* versions 1.2 defines AEAD IV construction with the sequence number encoded as little endian in memory.

400 The *Secured Messages using SPDM Specification (DSP0277)* versions 1.0 and version 1.1 do not explicitly specify how the AEAD IV is formed. To align with the endianness definition in version 1.2 is recommended.

401 # 10 Component behavior

402 ## 10.1 Reset processing

403 The SPDM specification and this document make reference to a component reset. The SPDM specification defines a reset as “a Reset or restart of a device that runs the Requester or Responder code, that typically leads to loss of all volatile state on the device.” The authors of the SPDM specification generally understand that a reset implies the following:

- Volatile state of the component is typically lost.
- Component firmware typically restarts execution, and might change to a different version.
- The host system and operating system might or might not reboot.
- One or more measurement values might change.
- The contents of one or more certificates might change, including the associated key pair.
- The mechanism to cause a reset is out of scope for the SPDM specification.

404 While the above list is typical of the SPDM authors’ understanding of reset, the authors also understand that components can implement many strategies that deviate from these behaviors.

405 ### 10.1.1 Transport level Reset

406 Some transports, like PCIe and CXL, define transport level reset (typically named Function Level Reset, FLR) that do not cause Reset of the whole device running SPDM Requester or Responder.

407 Authors of both PCIe and CXL specifications state that in case of FLR all security state of corresponding device shall be cleared.

- Refer to section 9.5 “Function Level Reset” of CXL 3.0 v.1 Base Specification
- Refer to section 6.6.2 “Function Level Reset (FLR)” of PCI Express Base Specification Revision 6.0

408 This effectively means that, in case of transport-level FLR, while a device in general does not restart and its firmware does not go thru cold restart sequence, both PCIe and CXL specifications state that all SPDM Secure Sessions that run on top of the transport has been reset shall be cancelled and re-established. It is also recommended to re-authenticate device to ensure its trustworthy state.

409 ## 10.2 Effectively Negotiated Connection Parameters

410 As the result of VCA (VERSION-CAPABILITIES-ALGORITHMS) flow both sides of SPDM Connection have the following parameters negotiated:

- SPDM Protocol Version: while any SPDM version supported by both sides could be used, it is recommended to use the highest version supported by both Requester and Responder. Also during all SPDM communication

starting from the message next after receiving VERSION response, Requester uses the same SPDM Version.

- Timing parameters
  - Most of timing parameters are informative for other side in the meaning that they do not require negotiation.
  - The HeartbeatPeriod timing parameter is the only one that is provided by Responder only but is effective by both sides.
- Capabilities: in most cases each Capability is the result of AND operation between Requester Capability and Responder Capability. Means: Connection supports Capability only if both Requester and Responder support that Capability.
  - The exceptions are the CACHE_CAP, GET_KEY_PAIR_INFO_CAP, SET_KEY_PAIR_INFO_CAP that are supported on the Responder side only.
    - CACHE_CAP: It is up to Requester to use this Capability, or to request the whole VCA sequence after each Reset.
- Message and chunk sizes are not subject of negotiation, they just inform other side of SPDM Communication about abilities to receive message and chunk of message (if chunking is supported) of certain size.
  - Abilities to send message of certain size are not negotiated nor informed. If Responder is unable to send large response in one chunk, it responds with ERROR=LargeResponse, and Requester shall use Large SPDM Message Transfer mechanism to receive response.
- Algorithms negotiation: for each group of algorithms Requester present the list of algorithms it supports, and Responder chooses one of those that it supports too. This algorithm will be used for given purposes. It is up to Requester to decide if certain group do not have common Algorithms supported by both Requester and Responder: Requester may choose to use only those SPDM Exchange elements that do not require Algorithms supported by both sides, or just terminate the SPDM sequence.

411 Table 9 — SPDM Requests and Responses requirements for Capabilities and Algorithms provides information what SPDM Request and Response messages require what algorithms and capabilities.

412 **Table 9 — SPDM Requests and Responses requirements for Capabilities and Algorithms**

Request	Response	Capabilities flags	Algorithms
GET_VERSION	VERSION	*Mandatory support*	*None*
GET_CAPABILITIES	CAPABILITIES	*Mandatory support*	*None*
NEGOTIATE_ALGORITHMS	ALGORITHMS	*Mandatory support*	*None*
GET_DIGESTS	DIGESTS	CERT_CAP	*BaseHashAlgo* or *ExtHash*
GET_CERTIFICATE	CERTIFICATE	CERT_CAP	*None*
CHALLENGE	CHALLENGE_AUTH	CHAL_CAP MEAS_CAP (conditional) to request Measurement Summary Hash	*BaseHashAlgo* or *ExtHash* (conditional) to request Measurement Summary Hash

Request	Response	Capabilities flags	Algorithms
GET_MEASUREMENTS	MEASUREMENTS	MEAS_CAP MEAS_FRESH_CAP (conditional) - to return *Measurements* recalculated as a result of GET_MEASUREMENTS request.	*BaseAsymAlgo* or *ExtAsym* (conditional) to request Signature
-	ERROR	*Mandatory support*	*None*
RESPOND_IF_READY	-	Depends on the original Request	Depends on the original Request
VENDOR_DEFINED_REQUEST	VENDOR_DEFINED_RESPONSE	*Implementation dependent (no capability flag)*	*None*
KEY_EXCHANGE	KEY_EXCHANGE_RSP	ENCRYPT_CAP and/or MAC_CAP KEY_EX_CAP MEAS_CAP (conditional) to request Measurement Summary Hash	DHE AEAD KeySchedule *BaseHashAlgo* or *ExtHash* (conditional) to request Measurement Summary Hash
FINISH	FINISH_RSP	ENCRYPT_CAP and/or MAC_CAP KEY_EX_CAP CERT_CAP, MUT_AUTH_CAP (conditional) to provide signature for mutual authentication HANDSHAKE_IN_THE_CLEAR_CAP (conditional) to send this request in the clear	*BaseHashAlgo* or *ExtHash* *ReqBaseAsymAlg* (conditional) to provide signature for mutual authentication
PSK_EXCHANGE	PSK_EXCHANGE_RSP	ENCRYPT_CAP and/or MAC_CAP PSK_CAP MEAS_CAP (conditional) to request Measurement Summary Hash	AEAD KeySchedule *BaseHashAlgo* or *ExtHash* (conditional) to request Measurement Summary Hash
PSK_FINISH	PSK_FINISH_RSP	ENCRYPT_CAP and/or MAC_CAP PSK_CAP	*BaseHashAlgo* or *ExtHash*
HEARTBEAT	HEARTBEAT_ACK	ENCRYPT_CAP and/or MAC_CAP since HEARTBEAT flow is possible inside Secure Session only HBEAT_CAP	*None*
KEY_UPDATE	KEY_UPDATE_ACK	ENCRYPT_CAP and/or MAC_CAP since KEY_UPDATE flow is possible inside Secure Session only KEY_UPD_CAP	KeySchedule
GET_ENCAPSULATED_REQUEST	ENCAPSULATED_REQUEST	ENCAP_CAP	*None*

Request	Response	Capabilities flags	Algorithms
DELIVER_ENCAPSULATED_RESPONSE	ENCAPSULATED_RESPONSE_ACK	ENCAP_CAP	*None*
END_SESSION	END_SESSION_ACK	ENCRYPT_CAP and/or MAC_CAP	*None*
GET_CSR	CSR	CERT_CAP CSR_CAP MULTI_KEY_CAP (conditional) to request specific key pair in Param1	*BaseAsymAlgo* or *ExtAsym* *ResponderMultiKeyConn* (conditional) to request specific key pair in Param1
SET_CERTIFICATE	SET_CERTIFICATE_RSP	CERT_CAP SET_CERT_CAP ALIAS_CERT_CAP (conditional) to be able to request setting up Alias certificate in Param1 of Request CERT_INSTALL_RESET_CAP (conditional) for Responder to be able to respond with ERROR=ResetRequired	*ResponderMultiKeyConn* (conditional) to request specific key pair in Param2 of Request
CHUNK_SEND	CHUNK_SEND_ACK	CHUNK_CAP	DataTransferSize smaller than MaxSPDMmsgSize
CHUNK_GET	CHUNK_RESPONSE	CHUNK_CAP	DataTransferSize smaller than MaxSPDMmsgSize
GET_KEY_PAIR_INFO	KEY_PAIR_INFO	MULTI_KEY_CAP GET_KEY_PAIR_INFO_CAP	*BaseAsymAlgo* or *ExtAsym*
SET_KEY_PAIR_INFO	SET_KEY_PAIR_INFO_ACK	MULTI_KEY_CAP SET_KEY_PAIR_INFO_CAP	*BaseAsymAlgo* or *ExtAsym*
GET_SUPPORTED_EVENT_TYPES	SUPPORTED_EVENT_TYPES	EVENT_CAP	*None*
SUBSCRIBE_EVENT_TYPES	SUBSCRIBE_EVENT_TYPES_ACK	EVENT_CAP	*None*
SEND_EVENT	EVENT_ACK	EVENT_CAP	*None*
GET_ENDPOINT_INFO	ENDPOINT_INFO	EP_INFO_CAP=01b or 10b CERT_CAP and EP_INFO_CAP=10b (conditional) for signature generation	*BaseAsymAlgo* or *ExtAsym* (conditional) for signature generation
GET_MEASUREMENT_EXTENSION_LOG	MEASUREMENT_EXTENSION_LOG	MEAS_CAP MEL_CAP	

10.3 Cached VCA

VCA (VERSION, CAPABILITIES, ALGORITHMS) information could be stored in persistent storage (storage that remains during device reset) and used in establishing SPDM Communication after reset in case if Responder reports CACHE_CAP in CAPABILITIES response.

VCA information that should be stored includes:

- Version, Capabilities and Algorithms negotiated during Negotiation phase of SPDM communication.
- The most recent data from the following SPDM message pairs enough for subsequent Transcript Hash calculations:
  - GET_VERSION - VERSION
  - GET_CAPABILITIES - CAPABILITIES
  - NEGOTIATE_ALGORITHMS - ALGORITHMS

NOTE: it is up to implementation what data to store for Transcript Hash calculations. It could be full messages data as well as partial hash calculations result.

NOTE: due to the fact that SPDM messages are small enough it could be more optimal to store whole messages, not partial hash calculation result, as the later may include some additional data required to continue hash calculations.

So, in the case when Responder reports CACHE_CAP in CAPABILITIES Response, Requester may skip VCA exchange in the subsequent SPDM Communication establishment and start from Attestation stage (GET_DIGESTS).

In addition to persistent storing of data mentioned above, Requester is required to have abilities to distinguish between different Responders. Similar requirement exists for Responder that communicates with multiple Requesters. See Endpoint identification for details. Storing of VCA state data is performed on the per-SPDM Communication (means: distinct Requester-Responder pair) basis.

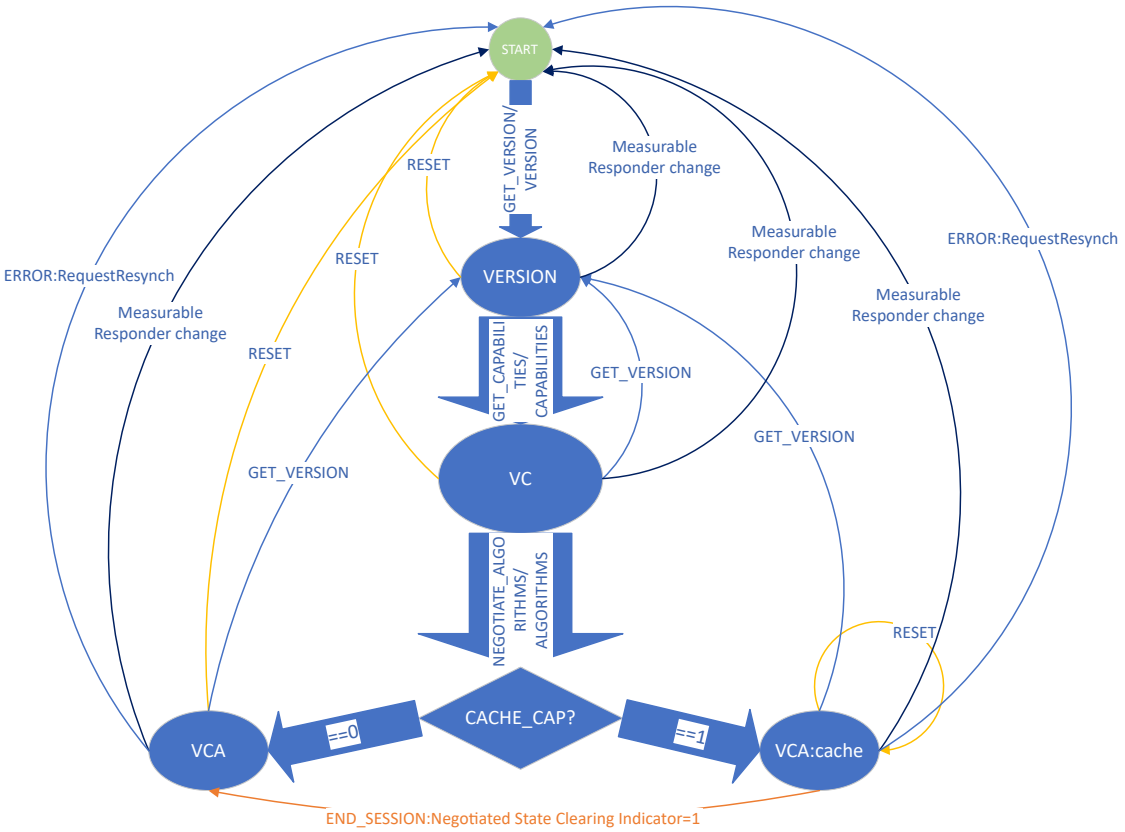
At any time, even if CACHE_CAP was reported by Responder, Responder may require re-negotiation of VCA from Requester by issuing ERROR response with code=RequestResynch.

Similarly Requester could send GET_VERSION request at any time despite of negotiated VCA state.

State of SPDM Communication is treated as negotiated only after ALGORITHMS Response is received by Requester. If reset (or any other kind of SPDM Communication break that may cause re-negotiation) happens after Requester receives CAPABILITIES Response with CACHE_CAP set but before Requester receives ALGORITHMS response, this state treated as not negotiated, and the next SPDM Connection establishment after reset starts from GET_VERSION Request.

Figure 7 — Cached Negotiated state flows

424



425 **NOTE** on the “Measurable Responder change” in the Figure 7 — “Cached Negotiated state flows”. This change means change in the Responder code or configuration. Measurable changes in the device running Responder code but not affecting Responder do not cause reset of Negotiated State.

426 # 11 Attestation and security policies

427 This clause provides guidelines on:

- Attestation policies that can be implemented using the SPDM specification.
- Security policies that can accompany such an implementation.

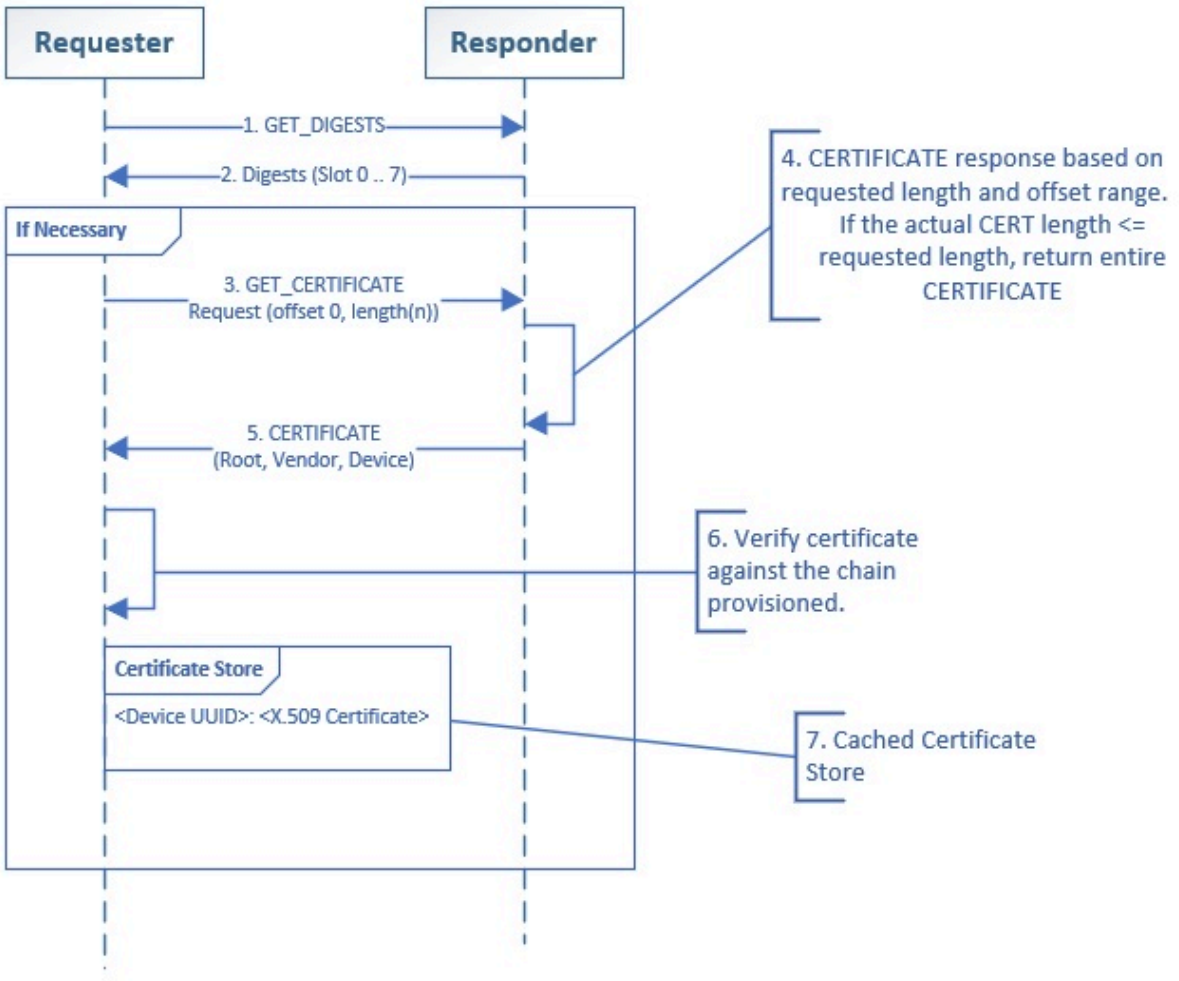
428 This clause is not exhaustive and should be considered informative. The details of any policy are vendor defined.

429 ## 11.1 Certificate authorization policy

430 Trusting the certificate chain and its security policy is confined to the authentication initiator's security policies. The SPDM authentication process involves retrieving the certificate chain digests first and comparing them with the cached digests, or the trust store database. If not found in the cached trust store database, the Requester sends the `GET_CERTIFICATE` request. The responder returns the certificate based on the requested length and offset, as Figure 8 — Example certificate authentication policy shows. It is recommended that the Requester perform certificate verification procedures before storing the corresponding digest to the trust store.

431 **Figure 8 — Example certificate authentication policy**

432



433 The following initiator security policies can verify certificate chains:

1. Generate warnings for components that do not support the SPDM.
2. Generate warnings for components that have certificate chains where root CA is not in the initiator's trust store database.
3. Quarantine components that have certificate chains where the root CA certificate is not in the trust store database.

434 11.2 Measurement

435 In addition to providing the hardware identity through a certificate, an authenticated endpoint can also be queried to provide the firmware identity. The firmware identity in this case refers to firmware code and configuration data. The value provided by the endpoint is a *measurement*. Using the `GET_MEASUREMENTS` command, the Requester can use a single command to ask for an individual measurement or all measurements. The returned values can be in the form

of a hash value or a bit stream, and the Requester can specify whether the measurements must be signed to verify that the measurements originated with the Responder endpoint.

436 The Requester can, in turn, compare the returned measurements to known values. The Requester can either verify the measurements locally or remotely. The mechanism to obtain reference measurement values is out of scope for the SPDM specification.

437 11.3 Secured Messages policy

438 The addition of Secured Messages enables Requesters and Responders to apply policies surrounding their use. For example, a Responder might not accept certain vendor defined messages that it deems to be potentially destructive unless it receives those commands in a Secured Message. Another example is that a Requester might not support communication with a Responder of a certain component class unless the component supports authenticated encrypted Secured Messages.

12 Secured Messages

Starting with version 1.1.0, the SPDM specification enables the use of Secured Messages.

12.1 Secured Message layering

This section discusses the layering of secured messages. The examples in the section are presented to illustrate the concepts used in secured message layering, but are not intended to prescribe an implementation.

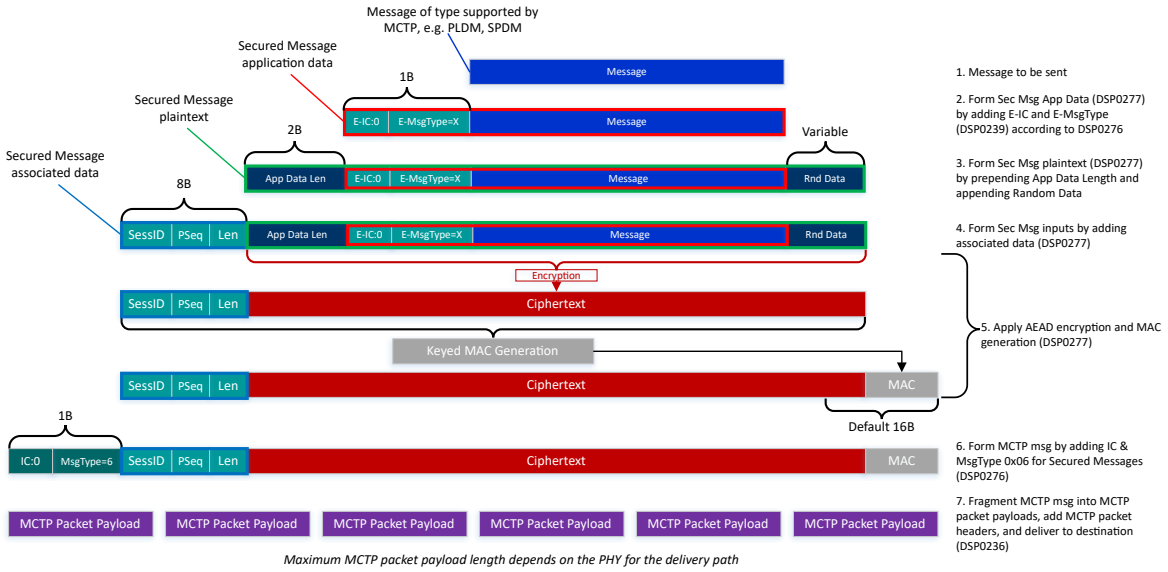
12.1.1 Secured Message send

Figure 9 — Secured Message send shows how layers are assembled when sending a Secured Message. The following describes the steps in the message assembly, moving from the top of the diagram to the bottom.

1. The component builds the message to be sent. This message can be any MCTP message type, as DMTF DSP0239 defines.
2. The component adds an MCTP header, setting the MCTP type and Integrity Check (IC) for the message. The result is the message to be encapsulated, which is the Application Data.
3. The component adds the `Application Data Length` and `Random Data` fields, as DMTF DSP0277 defines.
4. The component adds the Associated Data to the message, which comprises `Transport Version`, `Length` and `Session ID` as DMTF DSP0277 defines.
5. The component encrypts the message contents that were built over the previous steps, resulting in the ciphertext of the message. The component then generates a MAC over the message contents, including the ciphertext and the Associated Data. The encryption and MAC generation are typically handled by the AEAD algorithm.
6. The component appends the MAC to the message.
7. The component adds the MCTP header for the Secured Message, which is set to MCTP message type 6, as DMTF DSP0276 describes. This results in the Secured Message.
8. The component transmits the message as a sequence of one or more packets, as DMTF DSP0236 describes.

Figure 9 — Secured Message send

446



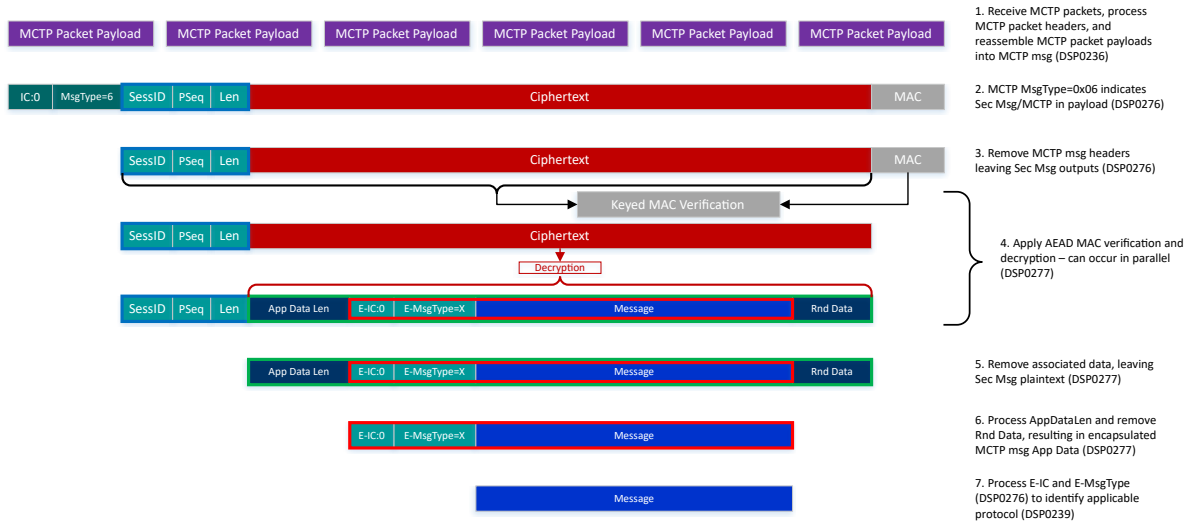
447 12.1.2 Secured Message receive

448 Figure 10 — Secured Message receive shows how layers are disassembled and authenticated when receiving a Secured Message. The following describes the steps in the message disassembly, moving from the top of the diagram to the bottom.

1. The component receives the message as a sequence of one or more packets, as DMTF DSP0236 describes, and reassembles the packets in an MCTP message.
2. The component reads the MCTP header to determine whether this is a Secured Message, which is indicated by MCTP message type 6 as DMTF DSP0276 describes. The MCTP header of the Secured Message is removed.
3. The component verifies the MAC by computing a MAC of the message and comparing it to the MAC field from the message.
4. The component removes the MAC field from the message.
5. The component uses the Associated Data from the message to decrypt the ciphertext. The decryption and MAC verification are typically handled by the AEAD algorithm.
6. The component removes the Associated Data from the message, leaving the plaintext of the Secured Message, as DMTF DSP0277 describes.
7. The component removes the `Application Data Length` and `Random Data` fields, as DMTF DSP0277 defines. The result is the encapsulated message.
8. The component processes the message.

449 **Figure 10 — Secured Message receive**

450



451

12.2 Secured Message error handling

452 If an error occurs during the Session Handshake phase or if an error happens during a secure session, the `Negotiated State` is preserved. The `Negotiated State` is preserved through errors unless the Requester sends an `END_SESSION` request with `Negotiated State Preservation Indicator=1` to terminate the session or sends a `GET_VERSION` request to reset the session.

453 If a timeout occurs during a secure session, the Requester can retry the message that failed. The retry is sent without modification. In this case, the Requester technically has more than one message outstanding to the same Responder but this is allowed because the second message is only a retry of the first message. Optionally, the Requester can send a `GET_VERSION` request to reset all sessions.

454

12.3 Secured Messages ordering

455 The *Key update allowances* section of *Secured Messages using SPDM Specification 1.0.0 (DSP0277)* discusses scenarios involving the use of `KEY_UPDATE` on transports that do not guarantee ordering. In this case, when a component receives a `KEY_UPDATE` command, it does not know if it has processed all of the outstanding messages that used the old key. Therefore, the `KEY_UPDATE` receiver should wait a period of time or for some other indicator to ensure that all messages encrypted with the old key have been received. *Secured Messages using SPDM over MCTP Binding Specification (DSP0276)* defines a time based process for waiting to discard the old key. Partner binding specifications should define a similar process (using time, a message count, or other event) if it is possible to receive messages out of order.

12.4 Random data

457 DSP0277 specifies that a component should set the `Random Data` field to a random length and fill it with random data. A component is allowed to set the length of `Random Data` to 0, or to fill the `Random Data` field with fixed values. However, there are benefits to using `Random Data` as DSP0277 suggests, including:

- Setting the length of `Random Data` to a random value can obfuscate the data being transmitted. An observer might gather information about the communication by observing the length of messages between two components and including data of random length hides the transmission from such an observation.
- Setting the contents of `Random Data` to random values ensures that all inputs to the encryption algorithm are unique. In the case of repeated encapsulated messages, the inclusion of random data values ensures that input plaintext to each encryption operation is unique.

12.5 Sequence number

459 The `Sequence Number` is an important part of Secured Message transfers. The requirements around the 8-byte `Sequence Number` are defined in DSP0277. However, the specification states that the transport binding specification specifies how much, if any, of the `Sequence Number` is transmitted. If the transport can ensure that SPDM messages arrive in order, the `Partial Sequence Number` might be absent from the transmitted packet (zero bytes in length). DSP0276 specifies a field called the `Partial Sequence Number`, which is 2 bytes in length. The DSP0276-defined `Partial Sequence Number` is specifically the lower 2 bytes of the DSP0277-defined `Sequence Number`.

13 Measurement extension log

Version 1.3 of the SPDM Specification adds support for measurement extension logs. Measurement extension logs provide a mechanism to verify logs that are constructed in an additive manner. The use of a measurement extension log results in a hash extended measurement (HEM), which is a single measurement that reflects one or more input values. The input values are captured in a measurement extension log (MEL). A Requester can validate an HEM by processing a MEL, to validate that the HEM is an accurate reflection of the associated MEL.

13.1 MEL construction

The MEL can be thought of as an ordered record of the steps that a component traversed to arrive at its current state. A generalized view is that a component would start a MEL from a reset or other initialization event. As the component initializes, it would add measurements or raw bit streams to the log as the associated data is consumed. For instance, a MEL might contain the following sequence:

1. Measurement of the ROM
2. Raw bit stream of the hardware configuration
3. Measurement of the second boot stage
4. Raw bit stream of the firmware version
5. Measurement of the firmware configuration

Each MEL entry in the MEL is associated with a measurement index, which is used to read the associated HEM. Each MEL entry also contains a MEL index, which is used to order the entries in the MEL. The MEL indices are sequentially increasing, which helps a Requester detect a missing entry.

The MEL offers informational entries. These entries are not part of the HEM construction, but are helpful for a user viewing or debugging the MEL.

13.1.1 MEL storage capacity

The MEL requires storage for the MEL. The size and nature of the storage is implementation specific. A component implementer should know how many events will be recorded during initialization, and can provision storage accordingly. Also, entries in a MEL might reflect real data, and may not need duplicate storage. For instance, if the hardware configuration raw bit stream is part of the MEL, the component might just return the bit stream from its primary location, instead of double buffering the data.

13.1.2 Use of MEL to record requests

The MEL may be used to record SPDM requests or responses. In theory, the Requester may send the two requests, `GET_MEASUREMENTS` and `GET_MEASUREMENT_EXTENSION_LOG`, in any order. `GET_MEASUREMENTS` should be sent before `GET_MEASUREMENT_EXTENSION_LOG` for the following reason. If the MEL is updated with new entries between the two

requests, this order would still allow the Requester to verify the MEL (except the new entries) against the hash-extended measurement. To do this, if the Requester finds that the hash-extended measurement resulted from extending the MEL entries does not match the value in `MEASUREMENTS`, the Responder may restart the extend operations with entries of the MEL and compare every intermediate hash-extended measurement with the value in `MEASUREMENTS`. If an intermediate hash-extended measurement matches the value in `MEASUREMENTS`, then it indicates that the MEL entries after this extend operation were appended to MEL after Responder sent the `MEASUREMENTS` message. The integrity of these MEL entries is not covered by the hash-extended measurement. The Requester may issue `GET_MEASUREMENTS` again and acquire the latest hash-extended measurement that covers these MEL entries.

470

13.2 HEM construction

471 An HEM is constructed using a hash operation. The HEM starts by initializing the HEM to all zeros. At each step, the existing HEM is concatenated with the next MEL entry and hashed. The hash operation is finalized on each step of MEL processing.

472 There may be more than one MEL presented by a component. Each MEL generates an independent HEM, which must be verified independently.

473 From the view of a component implementer, the storage for the HEM and the hash-extend operation must be protected, but the MEL does not require protected storage. When the Requester verifies the MEL, it will detect any discrepancies between the MEL and the HEM, so long as the process for creating the HEM is trustworthy.

474

13.2.1 HEM verification

475 The Requester or another entity can validate the authenticity of a MEL. In this process, the Requester reads the MEL using `GET_MEASUREMENT_EXTENSION_LOG`, walks through the entries to generate an HEM, and then compares its computed measurement to the HEM returned by the device.

476 # 14 Manifest support

477 A manifest is a general term that refers to structured data that describes the contents of measurements and might also contain measurement values. A manifest is typically in a machine readable format. A measurement manifest is a manifest that describes the current measurement values for a component. In contrast, a reference manifest is a manifest that describes the expected measurement values for a component. In practice, a Requester or an associated Verifier can use a reference manifest to evaluate the contents of a measurement manifest and any associated measurements. The processes for evaluating and comparing manifests are out of scope for the SPDM specification and this document; however, the SPDM specification does describe mechanisms to retrieve a measurement manifest from a device. A Requester can also track changes in a manifest using a digest of the manifest and the use of hash-extend measurements and a measurement extension log, if these are supported.

478 A measurement manifest can take the form of a data structure that contains measurement values and metadata that describes the measurements, or it can contain only the metadata that describes the measurements along with the index values used to retrieve the measurement values. The Responder selects the approach that it uses, and the selection of such an approach is out of scope for the SPDM specification.

479 Version 1.2 of the SPDM specification introduced an assigned Measurement Index (0xFD) for retrieval of a measurement manifest.

480 ## 14.1 Freeform measurement manifest

481 Version 1.1 of the SPDM specification introduced a `DMTFSpecMeasurementValueType` of `0x04` (“Measurement manifest”) to indicate that a `MEASUREMENTS` response is a measurement manifest. The format of the measurement manifest is not specified by the SPDM specification. Any measurement index is allowed to return a measurement manifest, and a component is allowed to use more than one index for measurement manifests. In version 1.3 of the SPDM specification, the value of `0x04` for the `DMTFSpecMeasurementValueType` field was renamed to “Freeform measurement manifest”, and the specification introduced a new `DMTFSpecMeasurementValueType` value of 0x0A for Structured measurement manifest.

482 ### 14.1.1 Structured measurement manifest

483 Version 1.3 of the SPDM specification introduced a `DMTFSpecMeasurementValueType` value of `0x0A` to indicate that a `MEASUREMENTS` response is a structured measurement manifest. A structured measurement manifest is different from a freeform manifest because it is defined as starting with an `SVH` header, which defines the standards body or vendor that defined the manifest data format. In conjunction with this change, a registry or standards body ID of `0x0A` was introduced to represent the IANA CBOR registry in an `SVH` header. This structure can be used to describe a manifest structure from any organization defined in the registry or standards body ID table, but were introduced to enable the use of manifest formats defined by TCG, a DMTF alliance partner organization.

15 Root of Trust

A Root of Trust provides the basis for trust in one or more security related functions. All Root of Trust functions that the following clauses list can be implemented in one or multiple entities. An implementer should consider the following roots of trust when implementing an SPDM solution.

15.1 Root of Trust for detection

The foundation of component trust relies on the internal security of the component. During the component-boot process, the component performs a signature verification of each firmware stage to ensure that the firmware is authentic and no unauthenticated code has been injected into the firmware image. Examples of how to accomplish this task include using a static Root of Trust for detection that can authenticate subsequent stages of the boot process. If the signature verification fails during the boot process, the component can halt, boot to a recovery partition, or follow another recovery path for the platform that also conforms to the security policy. For more details on a Root of Trust for firmware authentication, see NIST SP800-193. As NIST SP800-193 indicates, the

> …central tenet to the firmware protection guidelines is ensuring that only authentic and authorized firmware update images may be applied to platform components.

15.2 Root of Trust for measurement

The SPDM implementation relies on the integrity of reported measurements. The Root of Trust for measurement is responsible for measurement of the elements, such as firmware images, that the `MEASUREMENTS` response reports, and for storing these measurements in a secure fashion.

15.3 Root of Trust for reporting

A Root of Trust for reporting ensures that values reported in SPDM responses accurately reflect the reported underlying state or condition. The Root of Trust for reporting ensures that other software in the system or an unauthorized user does not alter reported values.

493 # 16 Partner implementations

494 DMTF partners with other standards bodies to enable those bodies to use SPDM on other interfaces and protocols.

495 ## 16.1 Available partner specifications

496 Many partner organizations provide specifications that use or extend SPDM, or place additional requirements on implementations.

- The PCI-SIG publishes implementation guidance as Component Measurement and Authentication (CMA), and the Data Object Exchange (DOE) binding specification that includes support for SPDM.
  - As part of CMA, the PCI-SIG guides to a subset of the algorithms that SPDM allows.
- The PCI-SIG uses SPDM to manage connections and keys as part of its Integrity and Data Encryption (IDE) capability.
- The CXL Consortium uses SPDM as part of its Integrity and Data Encryption (IDE) capability.
- The Open Compute Project (OCP) publishes requirements around the SPDM specification.
- The Trusted Computing Group (TCG) publishes a binding specification for the use of Device Identifier Composition Engine (DICE) artifacts over an SPDM connection.
  - TCG and the Internet Engineering Task Force (IETF) publish a manifest format that can be used with an SPDM implementation.
- HDBaseT and the MIPI Alliance reference SPDM.

497 ## 16.2 Partner binding specifications

498 DMTF enables partner standards bodies to create SPDM bindings for their specifications. Other binding specifications should provide the following guidance:

- Alterations to the `Subject Alternative Name` and `Common Name` fields in the certificate.
- Guidance on the vendor identification in the certificate.
- Bus timing and timeout requirements, including RTT.
- Use of `OpaqueData` fields in `CHALLENGE_AUTH` and `MEASUREMENTS` responses.
- Method to track messages from multiple Requesters, as Multiple caching Requesters describes.
- Method to uniquely identify endpoints, as described in Public key provisioning details.
- Process to clear out of order messages during a key update, as Secured Messages ordering describes.
- Rules for the transmission of part or all of the `Sequence Number` from DSP0277.

16.3 Endpoint identification

The use of caching (`CACHE_CAP=1`) introduces the need for endpoints to be able to uniquely identify each other so that the endpoint can locate the `VCA` data that it has cached for the endpoint. The stability of the identification method influences the stability of the caching of `VCA` , with less stable identifiers making a cache miss more likely. For reference, MCTP provides the `Get Endpoint UUID` command, which can be used for this purpose.

16.4 Enabling partner implementations

The SPDM specification has several mechanisms to enable partner implementations.

16.4.1 OpaqueData

Many messages include fields for `OpaqueData` and `OpaqueDataLength` . These fields are for partner standards bodies to use to meet their requirements to include additional data in the SPDM messages. By including these fields in the messages, the contents of the fields are also covered by message transcripts and signatures.

`OpaqueData` can be used to convey data that is out of scope for the SPDM specification. Examples include vendor defined additional data and random numbers. If a standards body requires the use of the `OpaqueData` fields, then the standards body in question is responsible for documenting the proper use of the `OpaqueData` fields.

16.4.1.1 Interpretation of opaque data

Secured messages, as described in DSP0277, can include an `OpaqueData` field. This field follows a defined format. However, this format can change between versions of the DSP0277 specification, and it is important that both endpoints use the same format when communicating. When SPDM version 1.1 or greater is in use, the endpoints use the opaque data format that is described in DSP0277. When SPDM version 1.2 or higher is in use, the endpoints use the opaque data format that is selected in the `ALGORITHMS` response, as described in DSP0274.

16.4.2 Registry or standards body ID

Several message exchanges include a field for `Registry ID` or `StandardID` , which allows the use of enumerations and field definitions that are defined by partner standards bodies. If a standards body requires an additional Registry or standards body definition, the standards body should work with DMTF to define a new `Registry or standards body ID` in the SPDM specification.

16.4.3 Vendor-defined commands

The SPDM specification has an allowance for vendor defined commands, using the `VENDOR_DEFINED_REQUEST` and `VENDOR_DEFINED_RESPONSE` messages. These messages include fields to provide a vendor ID for the vendor that defined the command, and to accommodate vendor IDs that are defined by multiple standards bodies.

512 In addition to the use of vendor-defined commands by component vendors, a standards body itself can define vendor-defined commands, in which case the standards body assigns itself a vendor ID of the type of its vendor ID.

513 If a standards body is not listed in the ***Registry or standards body ID*** table in the SPDM specification and there is a requirement to add a command using an ID from that standards body, then the standards body should work with DMTF to allocate an ID to the table to avoid potential conflicts.

514 16.4.4 Certificates with partner information

515 The SPDM specification defines information that is stored in a certificate, and all such information is identified using a unique OID. Partner standards bodies and component vendors can also define information to be stored in a certificate.

516 The following certificate gives an example of such a certificate. This certificate contains SPDM specification defined information in the `Subject Alternative Name` `otherName` identified by the OID 1.3.6.1.4.1.412.274.1. The partner organization information is found in a second `Subject Alternative Name` `otherName` field, and identified by the OID 1.2.3.4.5.6.7.1. Requesters that process certificates can read the OID for each `Subject Alternative Name` `otherName` to help Requester correctly interpret the associated data.

```
          Certificate:
              Data:
                  Version: 3 (0x2)
                  Serial Number: 4098 (0x1002)
                  Signature Algorithm: ecdsa-with-SHA256
                  Issuer: C = US, ST = NC, L = City, O = ACME, OU = ACME Devices, CN = CA
                  Validity
                      Not Before: Jan  1 00:00:00 1970 GMT
                      Not After : Dec 31 11:59:59 9999 GMT
                  Subject: C = US, ST = NC, L = City, O = ACME Widget Manufacturing, OU = ACME Widget
Manufacturing Unit, CN = x0123456789
                  Subject Public Key Info:
                      Public Key Algorithm: rsaEncryption
                          Public-Key: (2048 bit)
                          Modulus:
                              00:c7:d6:81:6b:16:fa:c9:a9:de:60:8a:3b:3e:c6:
                              11:a2:fd:48:d2:e9:e8:d2:f5:d4:10:08:06:ad:ee:
                              14:76:b7:41:15:88:c9:c1:d0:5a:58:08:b7:f0:04:
                              bb:85:31:43:2f:3a:c9:53:67:99:9e:fc:b6:af:70:
                              bb:1d:ef:b1:6d:69:fb:38:57:c7:71:da:fe:2b:fd:
                              bf:18:81:15:c6:e1:cb:1c:65:54:5f:de:04:f7:f6:
                              a1:f9:b3:8b:40:12:69:05:23:7c:15:41:27:ac:65:
                              6c:d9:66:f4:eb:3c:b8:4f:f6:5a:4d:7a:26:ad:2f:
                              66:2b:cd:28:7c:d6:a6:ae:71:70:c8:0e:a8:3e:a3:
                              a1:96:d4:65:41:e2:01:a8:34:15:ef:50:ce:99:3f:
                              1d:38:ba:5c:53:37:d2:f3:46:94:08:ee:22:87:e2:
                              90:7b:25:cf:6e:b0:cd:05:f1:e3:b7:5a:ee:f7:4f:
                              9d:70:74:81:86:8d:5e:14:af:37:24:d0:39:71:3c:
                              05:c2:a5:1c:a3:a1:5e:6b:f7:9e:5d:cf:c2:67:b9:
                              a3:f2:e6:62:c9:96:97:e3:5e:83:c6:14:dd:4c:8b:
```

```
                            53:87:7e:43:a2:81:28:4d:41:d1:48:b2:c9:c8:b2:
                            53:ff:ce:82:d8:f9:ed:48:5a:87:fd:85:19:dc:ea:
                            07:e5
                        Exponent: 65537 (0x10001)
                X509v3 extensions:
                    X509v3 Basic Constraints:
                        CA:FALSE
                    X509v3 Authority Key Identifier:
                        CB:0C:55:D9:4F:18:EE:B9:54:25:3D:08:1A:4C:02:24:80:BF:CF:FE
                    X509v3 Key Usage: critical
                        Digital Signature
                    X509v3 Subject Alternative Name:
                        otherName: 1.3.6.1.4.1.412.274.1::ACME:WIDGET:0123456789, otherName:
1.2.3.4.5.6.7.1::Vendor=ACME:Device=WIDGET:SN=0123456789
            Signature Algorithm: ecdsa-with-SHA256
            Signature Value:
                30:46:02:21:00:f1:a5:9b:1f:6e:ac:9d:11:24:d5:da:6f:2c:
                ea:c1:93:e8:0c:58:38:c9:66:38:5c:96:20:75:a7:77:5d:20:
                c5:02:21:00:88:30:e4:f0:2e:82:e4:45:93:84:e5:23:58:2d:
                90:c3:32:51:6f:a0:35:c8:7f:a4:6b:21:01:0a:13:db:26:92
```

16.5 Partner event definitions

The SPDM event mechanism allows partner organizations to define events. The event group definitions include an identifier for the standards body or vendor that defined the event. This provides unique namespaces for organizations to extend event definitions unilaterally. Note that DMTF defined events follow this pattern and identify DMTF as the defining organization.

519 # 17 ANNEX A (informative) change log

520 ## 17.1 Version 1.0.0 (2020-05-13)

- Initial Release

521 ## 17.2 Version 1.1.0 (2022-01-04)

- Update content and diagrams to match *Security Protocol and Data Model (SPDM) Specification 1.1.1 (DSP0274)*
- Restructure several sections to improve readability, including:
  - Certificates.
  - Partner implementations.
- Update references to latest versions.
- Removed statement about possible re-provisioning of the certificate chain in slot 0.
- New:
  - Add Authenticated Encryption with Associated Data (AEAD) to the introduction.
  - Add discussion of new Message details and CAPABILITIES Flags.
  - Add discussion of Pre-Shared Key.
  - Add discussion of details for use of the CACHE_CAP flag.
  - Add discussion of complexities around Certificate chain algorithms and implementation considerations.
  - Discuss validation of certificate chains in Certificate requirements.
  - Clarify use of `MeasurementSummaryHash` versus Summary measurements.
  - Clarify that SPDM code is in the SPDM Trusted Computing Base.
  - Add Figure 3 — SPDM security stack.
  - Add Secured Message layering example.
  - Add an example of Certificates with partner information.
  - Add discussion of Secured Messages and Secured Messages policy.
  - Add a section for Alternatives to certificate chains.
  - Add discussion of Vendor defined commands.
  - Added Table 8 — Combinations of secure session capabilities

522 ## 17.3 Version 1.2.0 (2022-09-26)

- Update content and diagrams to match *Security Protocol and Data Model (SPDM) Specification 1.2.1 (DSP0274)*

- Clean up Figure 9 — Secured Message send and Figure 10 — Secured Message receive.
- Discuss the case where Certificate retrieval changes between slots without reading the entire certificate chain.
- Change use of secure environment to trusted environment to match the SPDM specification.
- Correct the use case for `BaseAsymAlgo` in Certificate chain algorithms.
- Modify the guidance for Device key pair lifespans.
- New:
  - Introduce the AliasCert model in Figure 4 — Example certificate chain and Certificate chain models.
  - Added a description of the new, standardized Firmware debug indication.
  - Added a section on the Interpretation of opaque data.
  - Added a discussion of the Handling of Heartbeat disabled case.
  - Added a discussion of Session timeout handling.
  - Added a discussion of the implications of message retry on the transcript hash. See Message timeouts and retries.
  - Described differences in the Use of BaseAsymAlgo and ReqBaseAsymAlg.
  - Added discussion of Certificate provisioning commands, `GET_CSR` and `SET_CERTIFICATE`.
  - Added discussion of Overlapping GET_CSR requests.
  - Added discussion of Overlapping SET_CERTIFICATE requests.
  - Added discussion of Slot write behavior.
  - Added a discussion of Embedded certificate authority protection.
  - Added Table 6 — Comparison of X.509 identity certificate fields and renumbered tables.
  - Added Public key provisioning details, and added a pointer to this discussion for Partner implementations.
  - Clarified certificate provisioning details.
  - Added a description of Large message transfers.
  - Added a section to discuss Component behavior and Reset processing.
  - Added a discussion of GET_CERTIFICATE and GET_DIGESTS in a session.
  - Added a discussion of Responder measurement error handling in GET_MEASUREMENTS and MEASUREMENTS exchange.
  - Added clarification about certificate chain validation.

17.4 Version 1.3.0 (2024-06-05)

- Update content and diagrams to match *Security Protocol and Data Model (SPDM) Specification 1.3.0 (DSP0274)*.
- Update the CAPABILITIES flags with new capabilities for SPDM 1.3.
- State that a Device key pair should stay consistent to reflect a consistent hardware identity.
- Clarify in Large message transfer parameters that `MaxSPDMmsgSize` must be sufficiently sized for all scenarios.
- Add more details on errata versions in Compatibility between versions.
- Clarify Transcript management handling with busy responses.

- Added details of how some partner organizations are using SPDM.
- Update Message timeouts and retries to discuss when a Responder continues to request more time with `ErrorCode=ResponseNotReady`.
- New:
  - Added explanation of measurements of components in TCB.
  - Add a discussion of the Supported algorithms block.
  - Add a section for Mutual authentication required capabilities and discuss the dependencies for mutual authentication.
  - Added discussion of Manifest support, including the new structure measurement manifest format.
  - Add section for Measurement extension log.
  - Add a link to the SPDM Forum.
  - Added a discussion on the relationship between the `Sequence Number` and `Partial Sequence Number`.
  - Added a discussion of Hardware identity.
  - Added Slot erase and overwrite and Key and slot management.
  - Added a discussion of the GET_ENDPOINT_INFO and ENDPOINT_INFO exchange.
  - Added discussion of Event notification and Partner event definitions.
  - Added a section to discuss Secured Messages ordering.
  - Added a discussion of OpaqueData handling.
  - Added paragraph to discuss Endpoint identification for Partner implementations.
  - Added a section for Session-Secrets-Exchange collisions.
  - Added a section for Certificate slot management to discuss the `SlotSizeRequested` attribute and show a certificate chain slot state machine.
  - Add timing details specific to RDT and CT interactions.
  - Added discussion of Certificate model use cases and add `GenericCert`s.
  - Added a section “Hash-extend measurement and measurement extension log”.
  - Clarified behavior of Overlapping GET_CSR requests in SPDM 1.3.
  - Clarified behavior of Cached Negotiated State
  - Added Effectively Negotiated Connection Parameters
  - Added Transport Level Reset subsection
  - Added a section “Cryptography Endianness”.

18 Bibliography

DMTF DSP4014, *DMTF Process for Working Bodies 2.6.1*.