Internet Engineering Task Force (IETF)
Request for Comments: 6010
Category: Standards Track
ISSN: 2070-1721

R. Housley
Vigil Security, LLC
S. Ashmore
National Security Agency
C. Wallace
Cygnacom Solutions

September 2010

Cryptographic Message Syntax (CMS) Content Constraints Extension

Abstract

This document specifies the syntax and semantics for the Cryptographic Message Syntax (CMS) content constraints extension. This extension is used to determine whether a public key is appropriate to use in the processing of a protected content. In particular, the CMS content constraints extension is one part of the authorization decision; it is used when validating a digital signature on a CMS SignedData content or validating a message authentication code (MAC) on a CMS AuthenticatedData content or CMS AuthEnvelopedData content. The signed or authenticated content type is identified by an ASN.1 object identifier, and this extension indicates the content types that the public key is authorized to validate. If the authorization check is successful, the CMS content constraints extension also provides default values for absent attributes.

Status of This Memo

This is an Internet Standards Track document.

This document is a product of the Internet Engineering Task Force (IETF). It represents the consensus of the IETF community. It has received public review and has been approved for publication by the Internet Engineering Steering Group (IESG). Further information on Internet Standards is available in Section 2 of RFC 5741.

Information about the current status of this document, any errata, and how to provide feedback on it may be obtained at http://www.rfc-editor.org/info/rfc6010.

Table of Contents

1. Introduction

The Cryptographic Message Syntax (CMS) SignedData [RFC5652] construct is used to sign many things, including cryptographic module firmware packages [RFC4108] and certificate management messages [RFC5272]. Similarly, the CMS AuthenticatedData and CMS AuthEnvelopedData constructs provide authentication, which can be affiliated with an originator's static public key. CMS Content Constraints (CCC) information is conveyed via an extension in a certificate or trust anchor object that contains the originator's or signer's public key.

This document assumes a particular authorization model, where each originator is associated with one or more authorized content types. A CMS SignedData, AuthenticatedData, or AuthEnvelopedData will be considered valid only if the signature or message authentication code (MAC) verification process is successful and the originator is authorized for the encapsulated content type. For example, one originator might be acceptable for verifying signatures on firmware packages, but that same originator may be unacceptable for verifying signatures on certificate management messages.

An originator's constraints are derived from the certification path used to validate the originator's public key. Constraints are associated with trust anchors [RFC5914], and constraints are optionally included in public key certificates [RFC5280]. Using the CMS Content Constraints (CCC) extension, a trust anchor lists the content types for which it may be used. A trust anchor may also include further constraints associated with each of the content types. Certificates in a certification path may contain a CCC extension that further constrains the authorization for subordinate certificates in the certification path.

Delegation of authorizations is accomplished using the CCC certificate extension. An entity may delegate none, some, or all of its authorizations to another entity by issuing it a certificate with an appropriate CCC extension. Absence of a CCC certificate extension in a certificate means that the subject is not authorized for any content type. If the entity is an end entity, it may perform CCC delegation, i.e., through the use of proxy certificates. However, usage of proxy certificates is not described in this specification.

While processing the certification path, relying parties MUST ensure that authorizations of a subject of a certificate are constrained by the authorizations of the issuer of that certificate. In other words, when a content signature or MAC is validated, checks MUST be performed to ensure that the encapsulated content type is within the permitted set for the trust anchor (TA) and each certificate in the

path and that the constraints associated with the specific content type, if any, are satisfied by the TA and each certificate in the path.

Additionally, this document provides subordination rules for processing CCC extensions within the Trust Anchor Management Protocol (TAMP) and relies on vocabulary from that document [RFC5934].

1.1. CMS Data Structures

CMS encapsulation can be used to compose structures of arbitrary breadth and depth. This is achieved using a variety of content types that achieve different compositional goals. A content type is an arbitrary structure that is identified using an object identifier. This document defines two categories of content types: intermediate content types and leaf content types. Intermediate content types are those designed specifically to encapsulate one or more additional content types with the addition of some service (such as a signature). Leaf content types are those designed to carry specific information. (Leaf content types may contain other content types.) CCC is not used to constrain MIME encapsulated data, i.e., CCC processing stops when a MIME encapsulation layer is encountered. SignedData [RFC5652] and ContentCollection [RFC4073] are examples of intermediate content types. FirmwarePkgData [RFC4108] and TSTInfo [RFC3161] are examples of leaf content types. Protocol designers may provide an indication regarding the classification of content types within the protocol. Four documents define the primary intermediate content types:

RFC 5652 [RFC5652]: Cryptographic Message Syntax (CMS)

- SignedData
- EnvelopedData
- EncryptedData
- DigestedData
- AuthenticatedData

RFC 5083 [RFC5083]: The Cryptographic Message Syntax (CMS) AuthEnvelopedData Content Type

- AuthEnvelopedData

RFC 4073 [RFC4073]: Protecting Multiple Contents with the Cryptographic Message Syntax (CMS)

- ContentCollection

- ContentWithAttributes

RFC 3274 [RFC3274]: Compressed Data Content Type for Cryptographic Message Syntax (CMS)

- CompressedData

Some intermediate nodes can also function as leaf nodes in some situations. EncryptedData, EnvelopedData, and AuthEnvelopedData nodes will function as intermediate nodes for recipients that can decrypt the content and as encrypted leaf nodes for recipients who cannot decrypt the content.

When using CMS, the outermost structure is always ContentInfo. ContentInfo consists of an object identifier and an associated content. The object identifier describes the structure of the content. Object identifiers are used throughout the CMS family of specifications to identify structures.

Using the content types listed above, ignoring for the moment ContentCollection, encapsulation can be used to create structures of arbitrary depth. Two examples based on [RFC4108] are shown in Figure 1 and Figure 2.

When ContentCollection is used in conjunction with the other content types, tree-like structures can be defined, as shown in Figure 3.

The examples in Figures 1, 2, and 3 can each be represented as a tree: the root node is the outermost ContentInfo, and the leaf nodes are the encapsulated contents. The trees are shown in Figure 4.

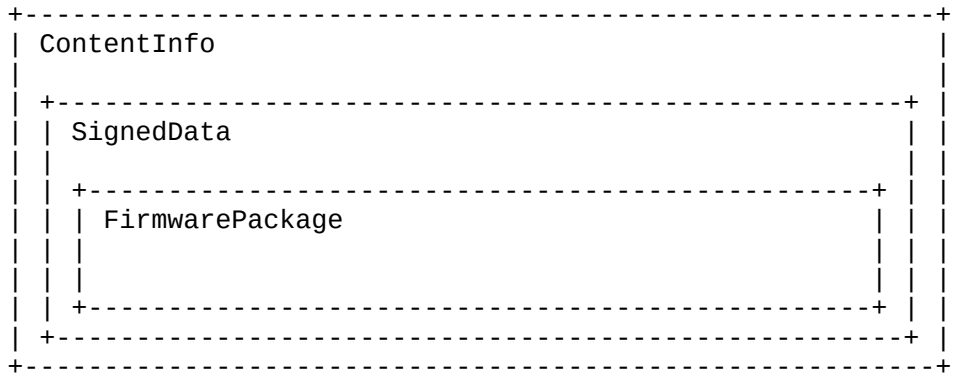

Figure 1. Example of a Signed Firmware Package

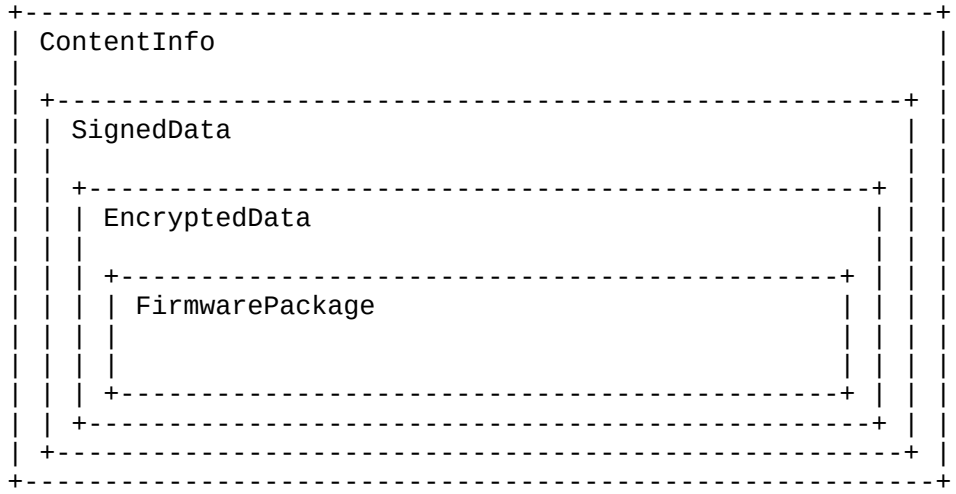

Figure 2. Example of a Signed and Encrypted Firmware Package

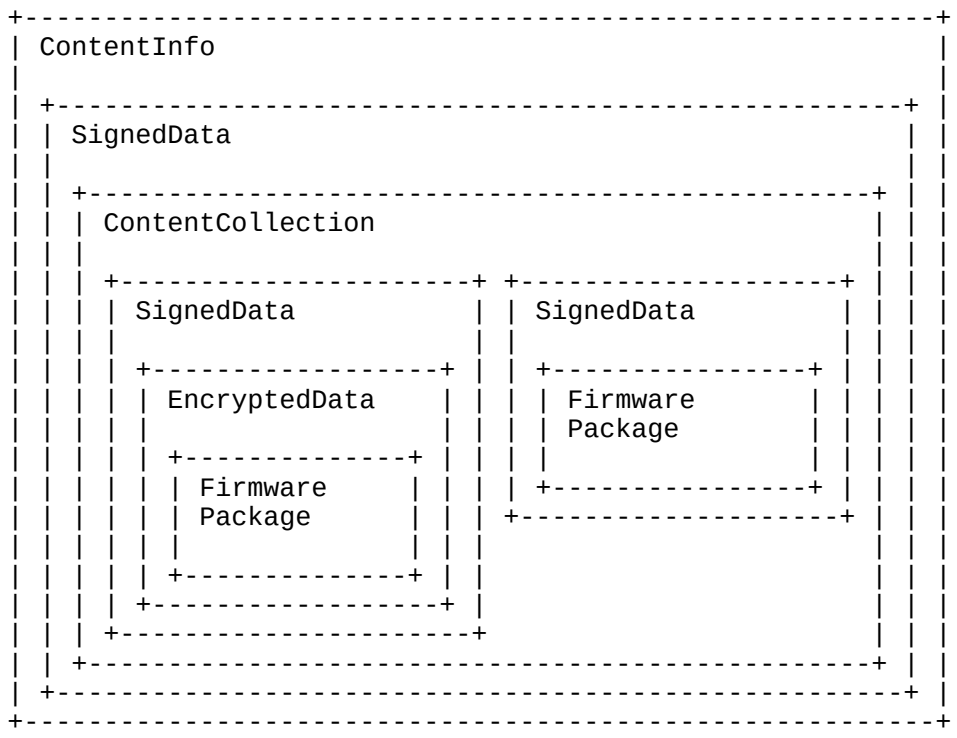

Figure 3. Example of Two Firmware Packages in a Collection

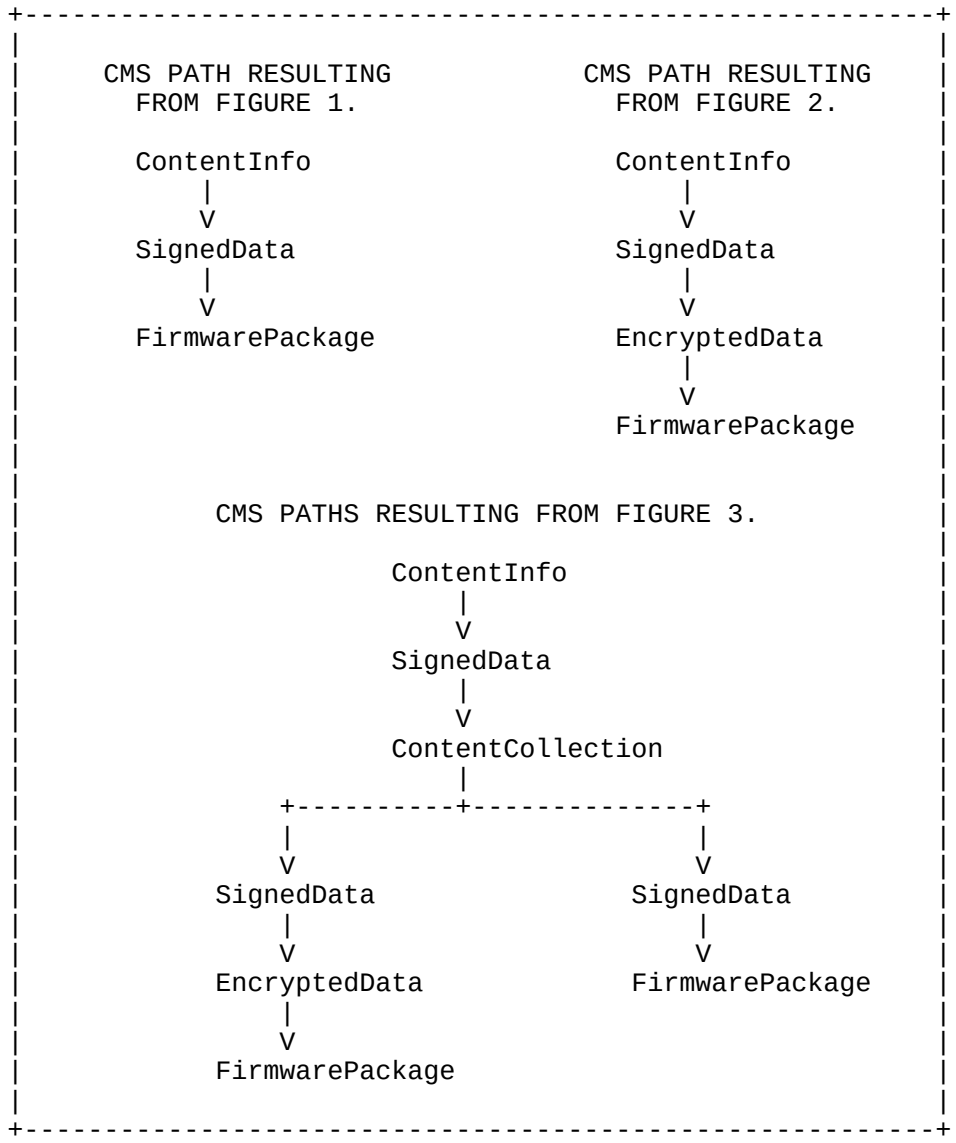

Figure 4. Example CMS Path Structures

These examples do not illustrate all of the details of CMS structures; most CMS protecting content types, and some leaf-node content types, contain attributes. Attributes from intermediate nodes can influence processing and handling of the CMS protecting content type or the encapsulated content type. Attributes from leaf nodes may be checked independent of the CCC processing, but such

processing is not addressed in this document. Throughout this document, paths through the tree structure from a root node to a leaf node in a CMS-protected message are referred to as CMS paths.

1.2. CMS Content Constraints Model

The CCC extension is used to restrict the types of content for which a particular public key can be used to verify a signature or MAC. Trust in a public key is established by building and validating a certification path from a trust anchor to the subject public key. Section 6 of [RFC5280] describes the algorithm for certification path validation, and the basic path validation algorithm is augmented, as described in Section 3 of this document, to include processing required to determine the CMS content constraints that have been delegated to the subject public key. If the subject public key is explicitly trusted (the public key belongs to a trust anchor), then any CMS content constraints associated with the trust anchor are used directly. If the subject public key is not explicitly trusted, then the CMS content constraints are determined by calculating the intersection of the CMS content constraints included in all the certificates in a valid certification path from the trust anchor to the subject public key, including those associated with the trust anchor.

CMS enables the use of multiple nested signatures or MACs. Each signature or MAC can protect and associate attributes with an encapsulated data object. The CMS content constraints extension is associated with a public key, and that public key is used to verify a signature or a MAC.

The CMS content constraints mechanism can be used to place limits on the use of the subject public key used for authentication or signature verification for one or more specific content types. Furthermore, within each permitted content type, a permitted set of values can be expressed for one or more specific attribute types.

When a leaf content type is encapsulated by multiple intermediate authentication layers, the signer or originator closest to a leaf node must be authorized to serve as a source for the leaf content type; outer signers or originators need not be authorized to serve as a source, but must be authorized for the leaf content type. All signers or originators must be authorized for the attributes that appear in a CMS path.

A signer or originator may be constrained to use a specific set of attribute values for some attribute types when producing a particular content type. If a signer or originator is constrained for a particular attribute that does not appear in a protected content of

the type for which the constraint is defined, the constraint serves as a default attribute, i.e., the payload should be processed as if an attribute equal to the constraint appeared in the protected content. However, in some cases, the processing rules for a particular content type may disallow the usage of default values for some attribute types and require a signer to explicitly assert the attribute to satisfy the constraint. Signer constraints are output for use in leaf node processing or other processing not addressed by this specification.

Three models for processing attributes were considered:

- Each signer or originator must be authorized for attributes it asserts.
- Each signer or originator must be authorized for attributes it asserts and attributes contained in the content it authenticates.
- Each signer or originator must be authorized for attributes it asserts, attributes contained in the content it authenticates, and attributes contained in content that authenticates it, i.e., all signers or originators must be authorized for all attributes appearing in the CMS path.

The third model is used in this specification.

1.3. Attribute Processing

This specification defines a mechanism for enforcing constraints on content types and attributes. Where content types are straightforward to process because there is precisely one content type of interest for a given CMS path, attributes are more challenging. Attributes can be asserted at many different points in a CMS path. Some attributes may, by their nature, be applicable to a specific node of a CMS path; for example, ContentType and MessageDigest attributes apply to a specific SignerInfo object. Other attributes may apply to a less well-defined target; for example, a ContentCollection may appear as the payload within a ContentWithAttributes object.

Since there is no automated means of determining what an arbitrary attribute applies to or how the attribute should be used, CCC processing simply collects attributes and makes them available for applications to use during leaf node processing. Implementations SHOULD refrain from collecting attributes that are known to be inapplicable to leaf node processing, for example, ContentType and MessageDigest attributes.

Some attributes contain multiple values. Attribute constraints expressed in a CCC extension may contain multiple values. Attributes expressed in a constraint that do not appear in a CMS path are returned as default attributes. Default attributes may have multiple values. Attributes are returned to an application via two output variables: cms_effective_attributes and cms_default_attributes. An attribute may be absent, present with one value, or present with multiple values in a CMS path and/or in CMS content constraints. A summary of the resulting nine possible combinations is below.

Attribute absent in CMS path; absent in cms_constraints: no action.

Attribute absent in CMS path; single value in cms_constraints: the value from cms_constraints is added to cms_default_attributes.

Attribute absent in CMS path; multiple values in cms_constraints: the values from cms_constraints are added to cms_default_attributes.

Attribute is present with a single value in CMS path; absent in cms_constraints: the value from CMS path is returned in cms_effective_attributes.

Attribute is present with a single value in CMS path; single value in cms_constraints: the value from CMS path must match the value from cms_constraints. If successful match, the value is returned in cms_effective_attribute. If no match, constraints processing fails.

Attribute is present with a single value in CMS path; multiple values in cms_constraints: the value from CMS path must match a value from cms_constraints. If successful match, the value from the CMS path is returned in cms_effective_attribute. If no match, constraints processing fails.

Attribute is present with multiple values in CMS path; absent in cms_constraints: the values from CMS path are returned in cms_effective_attributes.

Attribute is present with multiple values; single value in cms_constraints: the values from CMS path must match the value from cms_constraints (i.e., all values must be identical). If successful match, the values from the CMS path are returned in cms_effective_attribute. If no match, constraints processing fails.

Attribute is present with multiple values; multiple values in cms_constraints: each value from CMS path must match a value from cms_constraints. If each comparison is successful, the values from the CMS path are returned in cms_effective_attribute. If a comparison fails, constraints processing fails.

1.4. Abstract Syntax Notation

All X.509 certificate [RFC5280] extensions are defined using ASN.1 [X.680][X.690].

CMS content types [RFC5652] are also defined using ASN.1.

CMS uses the Attribute type. The syntax of Attribute is compatible with X.501 [X.501].

1.5. Terminology

The key words "MUST", "MUST NOT", "REQUIRED", "SHALL", "SHALL NOT", "SHOULD", "SHOULD NOT", "RECOMMENDED", "MAY", and "OPTIONAL" in this document are to be interpreted as described in RFC 2119 [RFC2119].

2. CMS Content Constraints Extension

The CMS content constraints extension provides a mechanism to constrain authorization during delegation. If the CMS content constraints extension is not present, then the subject of the trust anchor or certificate is not authorized for any content type, with an exception for apex trust anchors, which are implicitly authorized for all content types. A certificate issuer may use the CMS content constraints extension for one or more of the following purposes:

- Limit the certificate subject to a subset of the content types for which the certificate issuer is authorized.
- Add constraints to a previously unconstrained content type.
- Add additional constraints to a previously constrained content type.

The CMS content constraints extension MAY be critical, and it MUST appear at most one time in a trust anchor or certificate. The CMS content constraints extension is identified by the id-pe-cmsContentConstraints object identifier:

```
id-pe-cmsContentConstraints OBJECT IDENTIFIER ::=
    { iso(1) identified-organization(3) dod(6) internet(1)
      security(5) mechanisms(5) pkix(7) pe(1) 18 }
```

The syntax for the CMS content constraints extension is:

```
CMSContentConstraints ::= SEQUENCE SIZE (1..MAX) OF
  ContentTypeConstraint

ContentTypeGeneration ::= ENUMERATED {
    canSource(0),
    cannotSource(1)}

ContentTypeConstraint ::= SEQUENCE {
  contentType           OBJECT IDENTIFIER,
  canSource             ContentTypeGeneration DEFAULT canSource,
  attrConstraints       AttrConstraintList OPTIONAL }

AttrConstraintList ::= SEQUENCE SIZE (1..MAX) OF AttrConstraint

AttrConstraint ::= SEQUENCE {
  attrType               AttributeType,
  attrValues             SET SIZE (1..MAX) OF AttributeValue }

id-ct-anyContentType OBJECT IDENTIFIER ::= { iso(1) member-body(2)
       us(840) rsadsi(113549) pkcs(1) pkcs-9(9) smime(16)
       ct(1) 0 }
```

The CMSContentConstraints is a list of permitted content types and associated constraints. A particular content type MUST NOT appear more than once in a CMSContentConstraints. When the extension is present, the certificate subject is being authorized by the certificate issuer to sign or authenticate the content types in the permitted list as long as the provided constraints, if any, are met. The relying party MUST ensure that the certificate issuer is authorized to delegate the privileges. When the extension is absent, the certificate subject is not authorized for any content type.

The special id-ct-anyContentType value indicates the certificate subject is being authorized for any content type without any constraints. Where id-ct-anyContentType appears alongside a specific content type, the specific content type is authoritative. The id-ct-anyContentType object identifier can be used in trust anchors when the trust anchor is unconstrained. Where id-ct-anyContentType is asserted in the contentType field, the canSource field MUST be equal to the canSource enumerated value and attrConstraints MUST be absent, indicating that the trust anchor can serve as a source for any content type without any constraints.

The fields of the ContentTypeConstraint type have the following meanings:

contentType is an object identifier that specifies a permitted content type. When the extension appears in an end entity certificate, it indicates that a content of this type can be verified using the public key in the certificate. When the extension appears in a certification authority (CA) certificate, it indicates that a content of this type can be verified using the public key in the CA certificate or the public key in an appropriately authorized subordinate certificate. For example, this field contains id-ct-firmwarePackage when the public key can be used to verify digital signatures on firmware packages defined in [RFC4108]. A particular content type MUST NOT appear more than once in the list. Intermediate content types MUST NOT be included in the list of permitted content types. Since the content type of intermediate nodes is not subject to CMS Constraint Processing, originators need not be authorized for intermediate node content types. The intermediate content types are:

id-signedData,

id-envelopedData,

id-digestedData,

id-encryptedData,

id-ct-authEnvelopedData,

id-ct-authData,

id-ct-compressedData,

id-ct-contentCollection, and

id-ct-contentWithAttrs.

canSource is an enumerated value. If the canSource field is equal to canSource, then the subject can be the innermost authenticator of the specified content type. For a subject to be authorized to source a content type, the issuer of the subject certificate MUST also be authorized to source the content type. Regardless of the flag value, a subject can sign or authenticate a content that is already authenticated (when SignedData, AuthenticatedData, or AuthEnvelopedData is already present).

attrConstraints is an optional field that contains constraints that are specific to the content type. If the attrConstraints field is absent, the public key can be used to verify the specified content type without further checking. If the attrConstraints field is present, then the public key can only be used to verify the specified content type if all of the constraints are satisfied. A particular constraint type, i.e., attrValues structure for a particular attribute type, MUST NOT appear more than once in the attrConstraints for a specified content type. Constraints are checked by matching the values in the constraint against the corresponding attribute value(s) in the CMS path. Constraints processing fails if the attribute is present and the value is not one of the values provided in the constraint. Constraint checking is described fully in Section 4.

The fields of the AttrConstraint type have the following meanings:

attrType is an AttributeType, which is an object identifier that names an attribute. For a content encapsulated in a CMS SignedData, AuthenticatedData, or AuthEnvelopedData to satisfy the constraint, if the attributes that are covered by the signature or MAC include an attribute of the same type, then the attribute value MUST be equal to one of the values supplied in the attrValues field. Attributes that are not covered by the signature or MAC are not checked against constraints. Attribute types that do not appear as an AttrConstraint are unconstrained, i.e., the signer or originator is free to assert any value.

attrValues is a set of AttributeValue. The structure of each of the values in attrValues is determined by attrType. Constraint checking is described fully in Section 4.

3. Certification Path Processing

When CMS content constraints are used for authorization, the processing described in this section SHOULD be included in the certification path validation. The processing is presented as an augmentation to the certification path validation algorithm described in Section 6 of [RFC5280], as shown in the figure below. Alternative implementations are allowed but MUST yield the same results as described below.

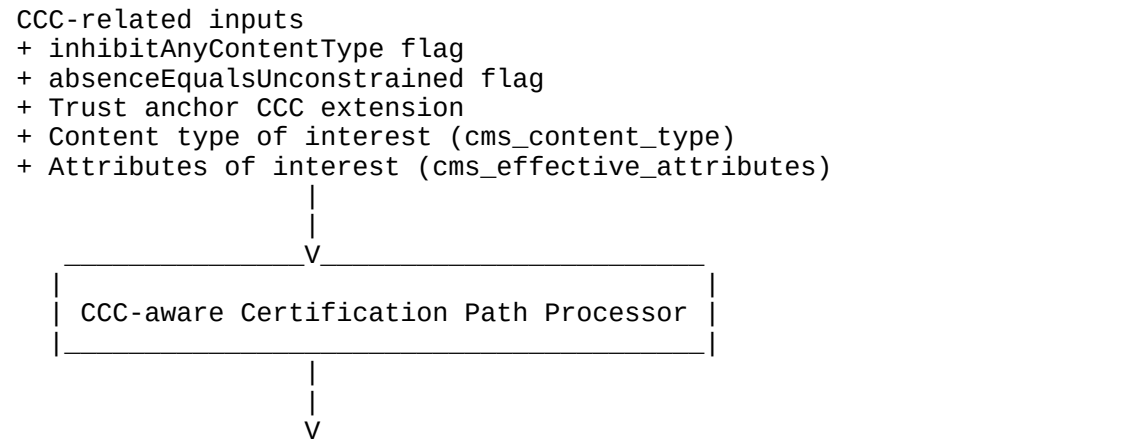

Figure 5. Certification Path Processing Inputs and Outputs

Certification path processing validates the binding between the subject and subject public key. If a valid certification path cannot be found, then the corresponding CMS path MUST be rejected.

3.1. Inputs

Two boolean values are provided as input: inhibitAnyContentType and absenceEqualsUnconstrained.

The inhibitAnyContentType flag is used to govern processing of the special id-ct-anyContentType value. When inhibitAnyContentType is true, id-ct-anyContentType is not considered to match a content type. When inhibitAnyContentType is false, id-ct-anyContentType is considered to match any content type.

The absenceEqualsUnconstrained flag is used to govern the meaning of CCC absence. When absenceEqualsUnconstrained is true, a trust anchor without a CCC extension is considered to be unconstrained and a certificate without a CCC extension is considered to have the same CCC privileges as its issuer. When absenceEqualsUnconstrained is false, a trust anchor or certificate without a CCC extension is not authorized for any content types.

Neither of these flags has any bearing on an apex trust anchor, which is always unconstrained by definition.

If a trust anchor used for path validation is authorized, then the trust anchor MAY include a CCC extension. A trust anchor may be constrained or unconstrained. If unconstrained, the trust anchor MUST either include a CMS Content Constraints extension containing the special id-ct-anyContentType value and inhibitAnyContentType is false or the trust anchor MUST have no CCC extension and absenceEqualsUnconstrained is true. If the trust anchor does not contain a CMS Content Constraints structure and absenceEqualsUnconstrained is false, the CMS content constraints processing fails. If the trust anchor contains a CCC extension with a single entry containing id-ct-anyContentType and inhibitAnyContentType is true, the CMS content constraints processing fails.

The content type of the protected content being verified can be provided as input along with the set of attributes collected from the CMS path in order to determine if the certification path is valid for a given context. Alternatively, the id-ct-anyContentType value can be provided as the content type input, along with an empty set of attributes, to determine the full set of constraints associated with a public key in the end entity certificate in the certification path being validated.

Trust anchors may produce CMS-protected contents. When validating messages originated by a trust anchor, certification path validation as described in Section 6 of [RFC5280] is not necessary, but constraints processing MUST still be performed for the trust anchor. In such cases, the initialization and wrap-up steps described below can be performed to determine if the public key in the trust anchor is appropriate to use in the processing of a protected content.

3.2. Initialization

Create an input variable named cms_content_type and set it equal to the content type provided as input.

Create an input variable named cms_effective_attributes and set it equal to the set of attributes provided as input.

Create a state variable named working_permitted_content_types. The initial value of working_permitted_content_types is the permitted content type list from the trust anchor, including any associated constraints.

Create a state variable named excluded_content_types. The initial value of excluded_content_types is empty.

Create a state variable of type SEQUENCE OF AttrConstraint named subject_default_attributes and initialize it to empty.

Create a state variable of type SEQUENCE OF ContentTypeConstraint named subject_constraints and initialize it to empty.

3.3. Basic Certificate Processing

If the CCC extension is not present in the certificate, check the value of absenceEqualsUnconstrained. If false, set working_permitted_content_types to empty. If true, working_permitted_content_types is unchanged. In either case, no further CCC processing is required for the certificate.

If inhibitAnyContenType is true, discard any entries in the CCC extension with a content type value equal to id-ct-anyContentType.

For each entry in the permitted content type list sequence in the CMS content constraints extension, the following steps are performed:

- If the entry contains the special id-ct-anyContentType value, skip to the next entry.

- If the entry contains a content type that is present in excluded_content_types, skip to the next entry.

- If the entry includes a content type that is not present in working_permitted_content_types, determine if working_permitted_content_types contains an entry equal to the special id-ct-anyContentType value. If no, no action is taken and working_permitted_content_types is unchanged. If yes, add the entry to working_permitted_content_types.

- If the entry includes a content type that is already present in working_permitted_content_types, then the constraints in the entry can further reduce the authorization by adding constraints to previously unconstrained attributes or by removing attribute values from the attrValues set of a constrained attribute. The canSource flag is set to cannotSource unless it is canSource in the working_permitted_content_types entry and in the entry. The processing actions to be performed for each constraint in the AttrConstraintList follow:

  -- If the constraint includes an attribute type that is not present in the corresponding working_permitted_content_types entry, add the attribute type and the associated set of attribute values to working_permitted_content_types entry.

```
      -- If the constraint includes an attribute type that is already
         present in the corresponding working_permitted_content_types
         entry, then compute the intersection of the set of attribute
         values from the working_permitted_content_types entry and the
         constraint.  If the intersection contains at least one
         attribute value, then the set of attribute values in
         working_permitted_content_types entry is assigned the
         intersection.  If the intersection is empty, then the entry is
         removed from working_permitted_content_types and the content
         type from the entry is added to excluded_content_types.
```

Remove each entry in working_permitted_content_types that includes a content type that is not present in the CMS content constraints extension. For values other than id-ct-anyContentType, add the removed content type to excluded_content_types.

3.4. Preparation for Certificate i+1

No additional action associated with the CMS content constraints extension is taken during this phase of certification path validation as described in Section 6 of [RFC5280].

3.5. Wrap-Up Procedure

If cms_content_type equals the special value anyContentType, the CCC processing portion of path validation succeeds. Set subject_constraints equal to working_permitted_content_types. If cms_content_type is not equal to the special value anyContentType, perform the following steps:

- If cms_content_type is present in excluded_content_types, the CCC processing portion of path validation fails.

- If working_permitted_content_types is equal to the special value anyContentType, set subject_constraints equal to working_permitted_content_types; the CCC processing portion of path validation succeeds.

- If cms_content_type does not equal the content type of an entry in working_permitted_content_types, constraints processing fails and path validation fails.

- If cms_content_type equals the content type of an entry in working_permitted_content_types, add the entry from working_permitted_content_types to subject_constraints. If the corresponding entry in working_permitted_content_types contains the special value anyContentType, set subject_constraints equal to cms_content_type; the CCC processing portion of path validation succeeds.

- If the attrConstraints field of the corresponding entry in working_permitted_content_types is absent; the CCC processing portion of path validation succeeds.

- If the attrConstraints field of the corresponding entry in working_permitted_content_types is present, then the constraints MUST be checked. For each attrType in the attrConstraints, the constraint is satisfied if either the attribute type is absent from cms_effective_attributes or each attribute value in the attrValues field of the corresponding entry in cms_effective_attributes is equal to one of the values for this attribute type in the attrConstraints field. If cms_effective_attributes does not contain an attribute of that type, then the entry from attrConstraints is added to the subject_default_attributes for use in processing the payload.

3.6. Outputs

If certification path validation processing succeeds, return the value of the subject_constraints, subject_default_attributes, and excluded_content_types variables.

4. CMS Content Constraints Processing

CMS contents constraints processing is performed on a per-CMS-path basis. The processing consists of traditional CMS processing augmented by collection of information required to perform content type and constraint checking. Content type and constraint checking uses the collected information to build and validate a certification path to each public key used to authenticate nodes in the CMS path per the certification path processing steps described above.

4.1. CMS Processing and CCC Information Collection

Traditional CMS content processing is augmented by the following three steps to support enforcement of CMS content constraints:

Collection of signer or originator keys

Collection of attributes

Leaf node classification

CMS processing and CCC information collection takes a CMS path as input and returns a collection of public keys used to authenticate protected content, a collection of authenticated attributes, and the leaf node, as shown in the figure below.

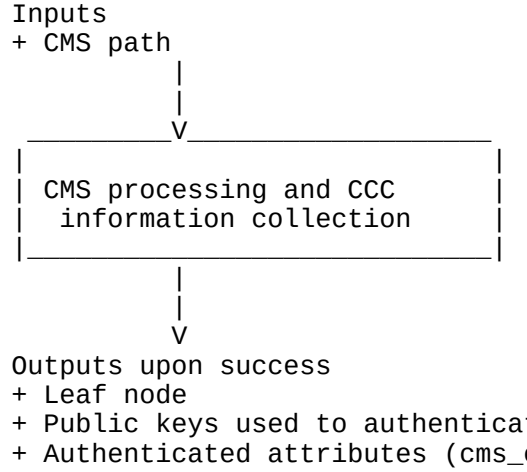

Figure 6. CMS Processing and CCC Information Collection

Processing is performed for each CMS path from the root node of a CMS-protected content to a leaf node, proceeding from the root node to the leaf node. Each path is processed independently of the other paths. Thus, it is possible that some leaf nodes in a content collection may be acceptable while other nodes are not acceptable. The processing described in this section applies to CMS paths that contain at least one SignedData, AuthEnvelopedData, or AuthenticatedData node. Since countersignatures are defined as not having a content, CMS content constraints are not used with countersignatures.

Signer or originator public keys are collected when verifying signatures or message authentication codes (MACs). These keys will be used to determine the constraints of each signer or originator by building and validating a certification path to the public key. Public key values, public key certificates, or public key identifiers are accumulated in a state variable named cms_public_keys, which is either initialized to empty or to an application-provided set of keys when processing begins. The variable will be updated each time a SignedData, AuthEnvelopedData, or AuthenticatedData node is encountered in the CMS path.

All authenticated attributes appearing in a CMS path are collected, beginning with the attributes protected by the outermost SignedData, AuthEnvelopedData, or AuthenticatedData and proceeding to the leaf node. During processing, attributes collected from the nodes in the CMS path are maintained in a state variable named cms_effective_attributes, and default attributes derived from message originator authorizations are collected in a state variable named cms_default_attributes. A default attribute value comes from a constraint that does not correspond to an attribute contained in the CMS path and may be used during payload processing in lieu of an explicitly included attribute. This prevents an originator from avoiding a constraint through omission. When processing begins, cms_effective_attributes and cms_default_attributes are initialized to empty. Alternatively, cms_effective_attributes may be initialized to an application-provided sequence of attributes. The cms_effective_attributes value will be updated each time an attribute set is encountered in a SignedData, AuthEnvelopedData, AuthenticatedData, or (authenticated) ContentWithAttributes node while processing a CMS path.

The output of content type and constraint checking always includes a set of attributes collected from the various nodes in a CMS path. When processing terminates at an encrypted node, the set of signer or originator public keys is also returned. When processing terminates at a leaf node, a set of default attribute values is also returned along with a set of constraints that apply to the CMS-protected content.

The output from CMS Content Constraints processing will depend on the type of the leaf node that terminates the CMS path. Four different output variables are possible. The conditions under which each is returned is described in the following sections. The variables are:

cms_public_keys is a list of public key values, public key certificates, or public key identifiers. Information maintained in cms_public_keys will be used to perform the certification path operations required to determine if a particular signer or originator is authorized to produce a specific object.

cms_effective_attributes contains the attributes collected from the nodes in a CMS path. cms_effective_attributes is a SEQUENCE OF Attribute, which is the same as the AttrConstraintList structure except that it may have zero entries in the sequence. An attribute can occur multiple times in the cms_effective_attribute set, potentially with different values.

cms_default_attributes contains default attributes derived from message signer or originator authorizations. A default attribute value is taken from a constraint that does not correspond to an attribute contained in the CMS path. cms_default_attributes is a SEQUENCE OF Attribute, which is the same as the AttrConstraintList structure except that it may have zero entries in the sequence.

cms_constraints contains the constraints associated with the message signer or originator for the content type of the leaf node. cms_constraints is a SEQUENCE OF Attribute, which is the same as the AttrConstraintList structure except that it may have zero entries in the sequence.

4.1.1. Collection of Signer or Originator Information

Signer or originator constraints are identified using the public keys to verify each SignedData, AuthEnvelopedData, or AuthenticatedData layer encountered in a CMS path. The public key value, public key certificate, or public key identifier of each signer or originator are collected in a state variable named cms_public_keys. Constraints are determined by building and validating a certification path for each public key after the content type and attributes of the CMS-protected object have been identified. If the CMS path has no SignedData, AuthEnvelopedData, or AuthenticatedData nodes, CCC processing succeeds and all output variables are set to empty.

The signature or MAC generated by the originator MUST be verified. If signature or MAC verification fails, then the CMS path containing the signature or MAC MUST be rejected. Signature and MAC verification procedures are defined in [RFC5652] [RFC5083]. The public key or public key certificate used to verify each signature or MAC in a CMS path is added to the cms_public_keys state variable for use in content type and constraint checking. Additional checks may be performed during this step, such as timestamp verification [RFC3161] and ESSCertId [RFC5035] processing.

4.1.1.1. Handling Multiple SignerInfo Elements

CMS content constraints MAY be applied to CMS-protected contents featuring multiple parallel signers, i.e., SignedData contents containing more than one SignerInfo. When multiple SignerInfo elements are present, each may represent a distinct entity or each may represent the same entity via different keys or certificates, e.g., in the event of key rollover or when the entity has been issued certificates from multiple organizations. For simplicity, signers represented by multiple SignerInfos within a single SignedData are not considered to be collaborating with regard to a particular content, unlike signers represented in distinct SignedData contents. Thus, for the purposes of CCC processing, each SignerInfo is treated as if it were the only SignerInfo. A content is considered valid if there is at least one valid CMS path employing one SignerInfo within each SignedData content. Where collaboration is desired, usage of multiple SignedData contents is RECOMMENDED.

Though not required by this specification, some applications may require successful processing of all or multiple SignerInfo elements within a single SignedData content. There are a number of potential ways of treating the evaluation process, including the following two possibilities:

- All signatures are meant to be collaborative: In this case, the public keys associated with each SignerInfo are added to the cms_public_keys variable, the attributes from each SignerInfo are added to the cms_effective_attributes variable, and normal processing is performed.

- All signatures are meant to be completely independent: In this case, each of the SignerInfos is processed as if it were a fork in the CMS path construction process. The application may require more than one CMS path to be valid in order to accept a content.

The exact processing will be a matter of application and local policy. See [RFC5752] for an example of an attribute that requires processing multiple SignerInfo elements within a SignedData content.

4.1.2. Collection of Attributes

Attributes are collected from all authenticated nodes in a CMS path. That is, attributes are not collected from content types that are unauthenticated, i.e., those that are not covered by a SignedData, AuthEnvelopedData, or AuthenticatedData layer. Additionally, an application MAY specify a set of attributes that it has authenticated, perhaps from processing one or more content types that encapsulate a CMS-protected content. Leaf node attributes MAY be

checked independent of the CCC processing, but such processing is not addressed in this document. Applications are free to perform further processing using all or some of the attributes returned from CCC processing.

4.1.3. Leaf Node Classification

The type of leaf node that terminates a CMS path determines the types of information that are returned and the type of processing that is performed. There are two types of leaf nodes: encrypted leaf nodes and payload leaf nodes.

A node in a CMS path is a leaf node if the content type of the node is not one of the following content types:

id-signedData (SignedData),

id-digestedData (DigestedData),

id-ct-authData (AuthenticatedData),

id-ct-compressedData (CompressedData),

id-ct-contentCollection (ContentCollection), or

id-ct-contentWithAttrs (ContentWithAttributes).

A leaf node is an encrypted leaf node if the content type of the node is one of the following content types:

id-encryptedData (EncryptedData),

id-envelopedData (EnvelopedData), or

id-ct-authEnvelopedData (AuthEnvelopedData).

All other leaf nodes are payload leaf nodes, since no further CMS encapsulation can occur beyond that node. However, specifications may define content types that provide protection similar to the CMS content types, may augment the lists of possible leaf and encrypted leaf nodes, or may define some encrypted types as payload leaf nodes.

When an encrypted leaf node is encountered, processing terminates and returns information that may be used as input when processing the decrypted contents. Content type and constraints checking are only performed for payload leaf nodes. When an encrypted leaf node terminates a CMS path, the attributes collected in cms_effective_attributes are returned along with the public key

information collected in cms_public_keys. When a payload leaf node terminates a CMS path, content type and constraint checking MUST be performed, as described in the next section.

4.2. Content Type and Constraint Checking

Content type and constraint checking is performed when a payload leaf node is encountered. This section does not apply to CMS paths that are terminated by an encrypted leaf node nor to CMS paths that have no SignedData, AuthEnvelopedData, or AuthenticatedData nodes.

4.2.1. Inputs

The inputs to content type and constraint checking are the values collected in cms_public_keys and cms_effective_attributes from a CMS path, along with the payload leaf node that terminates the CMS path, as shown in the figure below.

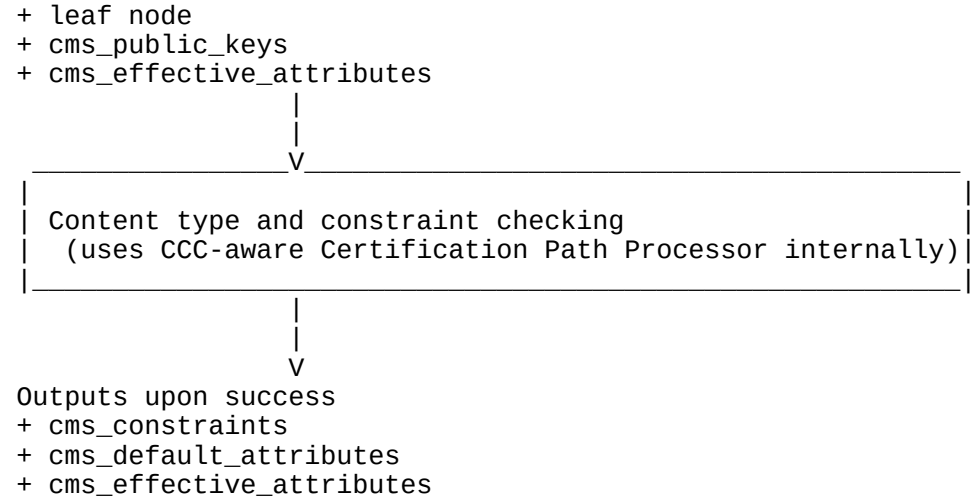

Figure 7. Content Type and Constraint Checking

4.2.2. Processing

When a payload leaf node is encountered in a CMS path and a signed or authenticated content type is present in the CMS path, content type and constraint checking MUST be performed. Content type and constraint checking need not be performed for CMS paths that do not contain at least one SignedData, AuthEnvelopedData, or AuthenticatedData content type. The cms_effective_attributes and cms_public_keys variables are used to perform constraint checking.

Two additional state variables are used during the processing: cms_constraints and cms_default_attributes, both of which are initialized to empty. The steps required to perform content type and constraint checking are below.

For each public key in cms_public_keys, build and validate a certification path from a trust anchor to the public key, providing the content type of the payload leaf node and cms_effective_attributes as input. Observe any limitations imposed by intermediate layers. For example, if the SigningCertificateV2 [RFC5035] attribute is used, the certificate identified by the attribute is required to serve as the target certificate.

- If path validation is successful, add the contents of subject_default_attributes to cms_default_attributes. The subject_constraints variable returned from certification path validation will contain a single entry. If the subject_constraints entry is equal to the special value anyContentType, content type and constraints checking succeeds. If the subject_constraints entry is not equal to the special value anyContentType, for each entry in the attrConstraints field of the entry in subject_constraints,
  - If there is an entry in cms_constraints with the same attrType value, add the value from the attrValues entry to the entry in cms_constraints if that value does not already appear.
  - If there is no entry in cms_constraints with the same attrType value, add a new entry to cms_constraints equal to the entry from the attrConstraints field.
- If the value of the canSource field of the entry in the subject_constraints variable for the public key used to verify the signature or MAC closest to the payload leaf node is set to cannotSource, constraints checking fails and the CMS path MUST be rejected.

If no valid certification path can be found, constraints checking fails and the CMS path MUST be rejected.

4.2.3. Outputs

When a payload leaf node is encountered and content type and constraint checking succeeds, return cms_constraints, cms_default_attributes, and cms_effective_attributes for use in leaf node payload processing.

When an encrypted leaf node is encountered and constraint checking is not performed, return cms_public_keys and cms_effective_attributes for use in continued processing (as described in Section 4.2.1).

The cms_effective_attributes list may contain multiple instances of the same attribute type. An instance of an attribute may contain multiple values. Leaf node processing, which might take advantage of these effective attributes, needs to describe the proper handling of this situation. Leaf node processing is described in other documents, and it is expected to be specific to a particular content type.

The cms_default_attributes list may contain attributes with multiple values. Payload processing, which might take advantage of these default attributes, needs to describe the proper handling of this situation. Payload processing is described in other documents, and it is expected to be specific to a particular content type.

5. Subordination Processing in TAMP

TAMP [RFC5934] does not define an authorization mechanism. CCC can be used to authorize TAMP message signers and to delegate TAMP message-signing authority. TAMP requires trust anchors managed by a TAMP message signer to be subordinate to the signer. This section describes subordination processing for CCC extensions of trust anchors contained in a TrustAnchorUpdate message where CCC is used to authorize TAMP messages.

For a Trust Anchor Update message that is not signed with the apex trust anchor operational public key to be valid, the digital signature MUST be validated using a management trust anchor associated with the id-ct-TAMP-update content type, either directly or via an X.509 certification path originating with an authorized trust anchor. The following subordination checks MUST also be performed as part of validation.

Each Trust Anchor Update message contains one or more individual updates, each of which is used to add, modify, or remove a trust anchor. For each individual update, the constraints of the TAMP message signer MUST be greater than or equal to the constraints of the trust anchor in the update. The constraints of the TAMP message signer and the to-be-updated trust anchor are determined based on the applicable CMS Content Constraints. Specifically, the constraints of the TAMP message signer are determined as described in Section 3, passing the special value id-ct-anyContentType and an empty set of attributes as input; the constraints of the to-be-updated trust anchor are determined as described below. If the constraints of a trust anchor in an update exceed the constraints of the signer, that

update MUST be rejected. Each update is considered and accepted or rejected individually without regard to other updates in the TAMP message. The constraints of the to-be-updated trust anchors are determined as follows:

- If the to-be-updated trust anchor is the subject of an add operation, the constraints are read from the CMSContentConstraints extension of the corresponding trust anchor in the update.

- If the to-be-updated trust anchor is the subject of a remove operation, the trust anchor is located in the message recipient's trust anchor store using the public key included in the update.

- If the to-be-updated trust anchor is the subject of a change operation, the trust anchor has two distinct sets of constraints that MUST be checked. The trust anchor's pre-change constraints are determined by locating the trust anchor in the message recipient's trust anchor store using the public key included in the update and reading the constraints from the CMSContentConstraints extension in the trust anchor. The trust anchor's post-change constraints are read from the CMSContentConstraints extension of the corresponding TBSCertificateChangeInfo or the TrustAnchorChangeInfo in the update. If the CMSContentConstraints extension is not present, then the trust anchor's post-change constraints are equivalent to the trust anchor's pre-change constraints.

The following steps can be used to determine if a Trust Anchor Update message signer is authorized to manage each to-be-updated trust anchor contained in a Trust Anchor Update message.

- The TAMP message signer's CMS Content Constraints are determined as described in Section 3, passing the special value id-ct-anyContentType and an empty set of attributes as input. The message signer MUST be authorized for the Trust Anchor Update message. This can be confirmed using the steps described in Section 4.

- The constraints of each to-be-updated trust anchor in the TAMP message MUST be checked against the message signer's constraints (represented in the message signer's subject_constraints computed above) using the following steps. For change operations, the following steps MUST be performed for the trust anchor's pre-change constraints and the trust anchor's post-change constraints.

  * If the to-be-updated trust anchor is unconstrained, the message signer MUST also be unconstrained, i.e., the message signer's subject_constraints MUST be set to the special value

anyContentType. If the to-be-updated trust anchor is unconstrained and the message signer is not, then the message signer is not authorized to manage the trust anchor and the update MUST be rejected.

* The message signer's authorization for each permitted content type MUST be checked using the state variables and procedures similar to those described in Sections 3.2 and 3.3. For each permitted content type in the to-be-updated trust anchor's constraints,

  + Set cms_effective_attributes equal to the value of the attrConstraints field from the permitted content type.

  + If the content type does not match an entry in the message signer's subject_constraints, the message signer is not authorized to manage the trust anchor and the update MUST be rejected. Note, the special value id-ct-anyContentType produces a match for all content types that have the resulting matching entry containing the content type, canSource set to canSource, and attrConstraints absent.

  + If the content type matches an entry in the message signer's subject_constraints, the canSource field of the entry is cannotSource, and the canSource field in the to-be-updated trust anchor's privilege is canSource, the message signer is not authorized to manage the trust anchor and the update MUST be rejected.

  + If the content type matches an entry in the message signer's subject_constraints and the entry's attrConstraints field is present, then constraints MUST be checked. For each attrType in the entry's attrConstraints, a corresponding attribute MUST be present in cms_effective_attributes containing values from the entry's attrConstraints. If values appear in the corresponding attribute that are not in the entry's attrConstraints or if there is no corresponding attribute, the message signer is not authorized to manage the trust anchor and the update MUST be rejected.

Once these steps are completed, if the update has not been rejected, then the message signer is authorized to manage the to-be-updated trust anchor.

Note that a management trust anchor that has only the id-ct-TAMP-update permitted content type is useful only for managing identity trust anchors. It can sign a Trust Anchor Update message, but it cannot impact a management trust anchor that is associated with any other content type.

6. Security Considerations

For any given certificate, multiple certification paths may exist, and each one can yield different results for CMS content constraints processing. For example, default attributes can change when multiple certification paths exist, as each path can potentially have different attribute requirements or default values.

Compromise of a trust anchor private key permits unauthorized parties to generate signed messages that will be acceptable to all applications that use a trust anchor store containing the corresponding management trust anchor. For example, if the trust anchor is authorized to sign firmware packages, then the unauthorized private key holder can generate firmware that may be successfully installed and used by applications that trust the management trust anchor.

For implementations that support validation of TAMP messages using X.509 certificates, it is possible for the TAMP message signer to have more than one possible certification path that will authorize it to sign Trust Anchor Update messages, with each certification path resulting in different CMS Content Constraints. The update is authorized if the processing below succeeds for any one certification path of the TAMP message signer. The resulting subject_constraints variable is used to check each to-be-updated trust anchor contained in the update message.

CMS does not provide a mechanism for indicating that an attribute applies to a particular content within a ContentCollection or a set CMS layers. For the sake of simplicity, this specification collects all attributes that appear in a CMS path. These attributes are processed as part of CCC processing and are made available for use in processing leaf node contents. This can result in a collection of attributes that have no relationship with the leaf node contents.

CMS does not provide a means for indicating what element within a CMS message an attribute applies to. For example, a MessageDigest attribute included in a SignedData signedAttributes collection applies to a specific signature, but a Firmware Package Identifier attribute appearing in the same list of attributes describes the encapsulated content. As such, CCC treats all attributes as applying to the encapsulated content type. Care should be taken to avoid

provisioning trust anchors or certificates that include constraints on attribute types that are never used to describe a leaf content type, such as a MessageDigest attribute.

The CMS Constraint Processing algorithm is designed to collect signer information for processing when all information for a CMS path is available. In cases where the certification path discovered during SignedData layer processing is not acceptable, an alternative certification path may be discovered that is acceptable. These alternatives may include an alternative signer certificate. When the ESSCertId attribute is used, alternative signer certificates are not permitted. The certificate referenced by ESSCertId must be used, possibly resulting in failure where alternative certificates would yield success.

7. Acknowledgments

Thanks to Jim Schaad for thorough review and many suggestions.

8. References

8.1. Normative References

[RFC2119] Bradner, S., "Key words for use in RFCs to Indicate Requirement Levels", BCP 14, RFC 2119, March 1997.

[RFC3274] Gutmann, P., "Compressed Data Content Type for Cryptographic Message Syntax (CMS)", RFC 3274, June 2002.

[RFC4073] Housley, R., "Protecting Multiple Contents with the Cryptographic Message Syntax (CMS)", RFC 4073, May 2005.

[RFC5083] Housley, R., "Cryptographic Message Syntax (CMS) Authenticated-Enveloped-Data Content Type", RFC 5083, November 2007.

[RFC5280] Cooper, D., Santesson, S., Farrell, S., Boeyen, S., Housley, R., and W. Polk, "Internet X.509 Public Key Infrastructure Certificate and Certificate Revocation List (CRL) Profile", RFC 5280, May 2008.

[RFC5652] Housley, R., "Cryptographic Message Syntax (CMS)", STD 70, RFC 5652, September 2009.

[RFC5911] Hoffman, P. and J. Schaad, "New ASN.1 Modules for Cryptographic Message Syntax (CMS) and S/MIME", RFC 5911, June 2010.

[RFC5912] Hoffman, P. and J. Schaad, "New ASN.1 Modules for the Public Key Infrastructure Using X.509 (PKIX)", RFC 5912, June 2010.

[X.208] "ITU-T Recommendation X.208 - Specification of Abstract Syntax Notation One (ASN.1)", 1988.

[X.501] ITU-T Recommendation X.501, "Information technology - Open Systems Interconnection - The Directory: Models", ISO/ IEC 9594-2:2005, 2005.

[X.680] "ITU-T Recommendation X.680: Information Technology - Abstract Syntax Notation One", 2002.

[X.690] "ITU-T Recommendation X.690 Information Technology - ASN.1 encoding rules: Specification of Basic Encoding Rules (BER), Canonical Encoding Rules (CER) and Distinguished Encoding Rules (DER)", 2002.

8.2. Informative References

[RFC3161] Adams, C., Cain, P., Pinkas, D., and R. Zuccherato, "Internet X.509 Public Key Infrastructure Time-Stamp Protocol (TSP)", RFC 3161, August 2001.

[RFC4108] Housley, R., "Using Cryptographic Message Syntax (CMS) to Protect Firmware Packages", RFC 4108, August 2005.

[RFC5035] Schaad, J., "Enhanced Security Services (ESS) Update: Adding CertID Algorithm Agility", RFC 5035, August 2007.

[RFC5272] Schaad, J. and M. Myers, "Certificate Management over CMS (CMC)", RFC 5272, June 2008.

[RFC5752] Schaad, J. and S. Turner, "Multiple Signatures in Cryptographic Message Syntax (CMS)", December 2009.

[RFC5914] Housley, R., Ashmore, S., and C. Wallace, "Trust Anchor Format", RFC 5914, June 2010.

[RFC5934] Housley, R., Ashmore, S., and C. Wallace, "Trust Anchor Management Protocol (TAMP)", RFC 5934, August 2010.

Appendix A. ASN.1 Modules

Appendix A.1 provides the normative ASN.1 definitions for the structures described in this specification using ASN.1 as defined in [X.680]. Appendix A.2 provides a module using ASN.1 as defined in [X.208]. The module in A.2 removes usage of newer ASN.1 features that provide support for limiting the types of elements that may appear in certain SEQUENCE and SET constructions. Otherwise, the modules are compatible in terms of encoded representation, i.e., the modules are bits-on-the-wire compatible aside from the limitations on SEQUENCE and SET constituents. A.2 is included as a courtesy to developers using ASN.1 compilers that do not support current ASN.1. A.1 references an ASN.1 module from [RFC5912] and [RFC5911].

A.1. ASN.1 Module Using 1993 Syntax

```
CMSContentConstraintsCertExtn
  { iso(1) identified-organization(3) dod(6) internet(1) security(5)
    mechanisms(5) pkix(7) id-mod(0) cmsContentConstr-93(42) }

DEFINITIONS IMPLICIT TAGS ::= BEGIN

IMPORTS
    EXTENSION, ATTRIBUTE
      FROM  -- from [RFC5912]
        PKIX-CommonTypes-2009
            {iso(1) identified-organization(3) dod(6) internet(1)
            security(5) mechanisms(5) pkix(7) id-mod(0)
            id-mod-pkixCommon-02(57)}

    CONTENT-TYPE, ContentSet, SignedAttributesSet, ContentType
    FROM  -- from [RFC5911]
        CryptographicMessageSyntax-2009
            { iso(1) member-body(2) us(840) rsadsi(113549)
            pkcs(1) pkcs-9(9) smime(16) modules(0)
            id-mod-cms-2004-02(41) }
    ;

id-ct-anyContentType ContentType ::=
    { iso(1) member-body(2)
      us(840) rsadsi(113549) pkcs(1) pkcs-9(9) smime(16)
      ct(1) 0 }

ct-Any CONTENT-TYPE ::= {NULL IDENTIFIED BY id-ct-anyContentType }
```

```
   --
   --  Add this to CertExtensions in PKIX1Implicit-2009
   --

   ext-cmsContentConstraints EXTENSION ::= {
       SYNTAX         CMSContentConstraints
       IDENTIFIED BY  id-pe-cmsContentConstraints }

   id-pe-cmsContentConstraints OBJECT IDENTIFIER ::=
       { iso(1) identified-organization(3) dod(6) internet(1)
         security(5) mechanisms(5) pkix(7) pe(1) 18 }

   CMSContentConstraints ::= SEQUENCE SIZE (1..MAX) OF
                             ContentTypeConstraint

   ContentTypeGeneration ::= ENUMERATED  {
       canSource(0),
       cannotSource(1)}

   ContentTypeConstraint ::= SEQUENCE {
       contentType           CONTENT-TYPE.&id ({ContentSet|ct-Any,...}),
       canSource             ContentTypeGeneration DEFAULT canSource,
       attrConstraints       AttrConstraintList OPTIONAL }


   Constraint { ATTRIBUTE:ConstraintList } ::= SEQUENCE {
       attrType           ATTRIBUTE.
               &id({ConstraintList}),
       attrValues         SET SIZE (1..MAX) OF ATTRIBUTE.
               &Type({ConstraintList}{@attrType})  }

   SupportedConstraints ATTRIBUTE ::= {SignedAttributesSet, ... }

   AttrConstraintList ::=
       SEQUENCE SIZE (1..MAX) OF Constraint {{ SupportedConstraints }}

   END
```

A.2. ASN.1 Module Using 1988 Syntax

```
CMSContentConstraintsCertExtn-88
  { iso(1) identified-organization(3) dod(6) internet(1) security(5)
    mechanisms(5) pkix(7) id-mod(0) cmsContentConstr-88(41) }

DEFINITIONS IMPLICIT TAGS ::=
BEGIN

IMPORTS
    AttributeType, AttributeValue
      FROM PKIX1Explicit88 -- from [RFC5280]
        { iso(1) identified-organization(3) dod(6) internet(1)
          security(5) mechanisms(5) pkix(7) id-mod(0)
          id-pkix1-explicit(18) } ;

id-ct-anyContentType OBJECT IDENTIFIER ::=
    { iso(1) member-body(2)
      us(840) rsadsi(113549) pkcs(1) pkcs-9(9) smime(16)
      ct(1) 0}

-- Extension object identifier

id-pe-cmsContentConstraints OBJECT IDENTIFIER ::=
    { iso(1) identified-organization(3) dod(6) internet(1)
      security(5) mechanisms(5) pkix(7) pe(1) 18 }

-- CMS Content Constraints Extension

CMSContentConstraints ::= SEQUENCE SIZE (1..MAX) OF
                          ContentTypeConstraint

ContentTypeGeneration ::= ENUMERATED  {
    canSource(0),
    cannotSource(1)}

ContentTypeConstraint ::= SEQUENCE {
    contentType           OBJECT IDENTIFIER,
    canSource             ContentTypeGeneration DEFAULT canSource,
    attrConstraints       AttrConstraintList OPTIONAL }

AttrConstraintList ::= SEQUENCE SIZE (1..MAX) OF AttrConstraint

AttrConstraint ::= SEQUENCE {
    attrType               AttributeType,
    attrValues             SET SIZE (1..MAX) OF AttributeValue }

END
```

Authors' Addresses

Russ Housley
Vigil Security, LLC
918 Spring Knoll Drive
Herndon, VA 20170

EMail: housley@vigilsec.com

Sam Ashmore
National Security Agency
Suite 6751
9800 Savage Road
Fort Meade, MD 20755

EMail: srashmo@radium.ncsc.mil

Carl Wallace
Cygnacom Solutions
Suite 5400
7925 Jones Branch Drive
McLean, VA 22102

EMail: cwallace@cygnacom.com