Internet Draft
draft-ietf-smime-ess-12.txt
March 29, 1999
Expires in six months

Editor: Paul Hoffman
Internet Mail Consortium

Enhanced Security Services for S/MIME

Status of this memo

Internet-Drafts are working documents of the Internet Engineering Task Force (IETF), its areas, and its working groups. Note that other groups may also distribute working documents as Internet-Drafts.

Internet-Drafts are draft documents valid for a maximum of six months and may be updated, replaced, or obsoleted by other documents at any time. It is inappropriate to use Internet- Drafts as reference material or to cite them other than as "work in progress."

To view the list Internet-Draft Shadow Directories, see http://www.ietf.org/shadow.html.

This document is an Internet-Draft and is in full conformance with all provisions of Section 10 of RFC2026.

1. Introduction

This document describes four optional security service extensions for S/MIME. The services are:

- signed receipts
- security labels
- secure mailing lists
- signing certificates

The first three of these services provide functionality that is similar to the Message Security Protocol [MSP4], but are useful in many other environments, particularly business and finance. Signing certificates are useful in any environment where certificates might be transmitted with signed messages.

The services described here are extensions to S/MIME version 3 ([MSG] and [CERT]), and some of them can also be added to S/MIME version 2 [SMIME2]. The extensions described here will not cause an S/MIME version 3 recipient to be unable to read messages from an S/MIME version 2 sender. However, some of the extensions will cause messages created by an S/MIME version 3 sender to be unreadable by an S/MIME version 2 recipient.

This document describes both the procedures and the attributes needed for the four services. Note that some of the attributes described in this document are quite useful in other contexts and should be

considered when extending S/MIME or other CMS applications.

The format of the messages are described in ASN.1:1988 [ASN1-1988].

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 [MUSTSHOULD].

This document is being discussed on the "ietf-smime" mailing list. To subscribe, send a message to:

```
     ietf-smime-request@imc.org
```

with the single word

```
     subscribe
```

in the body of the message. There is a Web site for the mailing list at <http://www.imc.org/ietf-smime/>.

1.1 Triple Wrapping

Some of the features of each service use the concept of a "triple wrapped" message. A triple wrapped message is one that has been signed, then encrypted, then signed again. The signers of the inner and outer signatures may be different entities or the same entity. Note that the S/MIME specification does not limit the number of nested encapsulations, so there may be more than three wrappings.

1.1.1 Purpose of Triple Wrapping

Not all messages need to be triple wrapped. Triple wrapping is used when a message must be signed, then encrypted, and then have signed attributes bound to the encrypted body. Outer attributes may be added or removed by the message originator or intermediate agents, and may be signed by intermediate agents or the final recipient.

The inside signature is used for content integrity, non-repudiation with proof of origin, and binding attributes (such as a security label) to the original content. These attributes go from the originator to the recipient, regardless of the number of intermediate entities such as mail list agents that process the message. The signed attributes can be used for access control to the inner body. Requests for signed receipts by the originator are carried in the inside signature as well.

The encrypted body provides confidentiality, including confidentiality of the attributes that are carried in the inside signature.

The outside signature provides authentication and integrity for information that is processed hop-by-hop, where each hop is an intermediate entity such as a mail list agent. The outer signature binds attributes (such as a security label) to the encrypted body. These attributes can be used for access control and routing decisions.

1.1.2 Steps for Triple Wrapping

The steps to create a triple wrapped message are:

1. Start with a message body, called the "original content".

2. Encapsulate the original content with the appropriate MIME Content-type headers, such as "Content-type: text/plain". An exception to this MIME encapsulation rule is that a signed receipt is not put in MIME headers.

3. Sign the result of step 2 (the inner MIME headers and the original content). The SignedData encapContentInfo eContentType object identifier MUST be id-data. If the structure you create in step 4 is multipart/signed, then the SignedData encapContentInfo eContent MUST be absent. If the structure you create in step 4 is application/pkcs7-mime, then the SignedData encapContentInfo eContent MUST contain the result of step 2 above. The SignedData structure is encapsulated by a ContentInfo SEQUENCE with a contentType of id-signedData.

4. Add an appropriate MIME construct to the signed message from step 3 as defined in [MSG]. The resulting message is called the "inside signature".

- If you are signing using multipart/signed, the MIME construct added consists of a Content-type of multipart/signed with parameters, the boundary, the result of step 2 above, the boundary, a Content-type of application/pkcs7-signature, optional MIME headers (such as Content-transfer-encoding and Content-disposition), and a body part that is the result of step 3 above.

- If you are instead signing using application/pkcs7-mime, the MIME construct added consists of a Content-type of application/pkcs7-mime with parameters, optional MIME headers (such as Content-transfer-encoding and Content-disposition), and the result of step 3 above.

5. Encrypt the result of step 4 as a single block, turning it into an application/pkcs7-mime object. The EnvelopedData encryptedContentInfo contentType MUST be id-data. The EnvelopedData structure is encapsulated by a ContentInfo SEQUENCE with a contentType of id-envelopedData. This is called the "encrypted body".

6. Add the appropriate MIME headers: a Content-type of application/pkcs7-mime with parameters, and optional MIME headers such as Content-transfer-encoding and Content-disposition.

7. Using the same logic as in step 3 above, sign the result of step 6 (the MIME headers and the encrypted body) as a single block

8. Using the same logic as in step 4 above, add an appropriate MIME

construct to the signed message from step 7. The resulting message is called the "outside signature", and is also the triple wrapped message.

1.2 Format of a Triple Wrapped Message

A triple wrapped message has many layers of encapsulation. The structure differs based on the choice of format for the signed portions of the message. Because of the way that MIME encapsulates data, the layers do not appear in order, and the notion of "layers" becomes vague.

There is no need to use the multipart/signed format in an inner signature because it is known that the recipient is able to process S/MIME messages (because they decrypted the middle wrapper). A sending agent might choose to use the multipart/signed format in the outer layer so that a non-S/MIME agent could see that the next inner layer is encrypted; however, this is not of great value, since all it shows the recipient is that the rest of the message is unreadable. Because many sending agents always use multipart/signed structures, all receiving agents MUST be able to interpret either multipart/signed or application/pkcs7-mime signature structures.

The format of a triple wrapped message that uses multipart/signed for both signatures is:

```
[step 8] Content-type: multipart/signed;
[step 8]    protocol="application/pkcs7-signature";
[step 8]    boundary=outerboundary
[step 8]
[step 8] --outerboundary
[step 6] Content-type: application/pkcs7-mime;              )
[step 6]    smime-type=enveloped-data                       )
[step 6]                                                    )
[step 4] Content-type: multipart/signed;                  | )
[step 4]    protocol="application/pkcs7-signature";       | )
[step 4]    boundary=innerboundary                        | )
[step 4]                                                  | )
[step 4] --innerboundary                                  | )
[step 2] Content-type: text/plain                       % | )
[step 2]                                                % | )
[step 1] Original content                               % | )
[step 4]                                                  | )
[step 4] --innerboundary                                  | )
[step 4] Content-type: application/pkcs7-signature        | )
[step 4]                                                  | )
[step 3] inner SignedData block (eContent is missing)     | )
[step 4]                                                  | )
[step 4] --innerboundary--                                | )
[step 8]
[step 8] --outerboundary
[step 8] Content-type: application/pkcs7-signature
```

```
[step 8]
[step 7] outer SignedData block (eContent is missing)
[step 8]
[step 8] --outerboundary--
```

% = These lines are what the inner signature is computed over.
| = These lines are what is encrypted in step 5. This encrypted result
    is opaque and is a part of an EnvelopedData block.
) = These lines are what the outer signature is computed over.

The format of a triple wrapped message that uses application/pkcs7-mime for the both signatures is:

```
[step 8] Content-type: application/pkcs7-mime;
[step 8]    smime-type=signed-data
[step 8]
[step 7] outer SignedData block (eContent is present)        O
[step 6] Content-type: application/pkcs7-mime;             ) O
[step 6]    smime-type=enveloped-data;                     ) O
[step 6]                                                   ) O
[step 4] Content-type: application/pkcs7-mime;           | ) O
[step 4]    smime-type=signed-data                       | ) O
[step 4]                                                 | ) O
[step 3] inner SignedData block (eContent is present)  I | ) O
[step 2] Content-type: text/plain                      I | ) O
[step 2]                                               I | ) O
[step 1] Original content                              I | ) O
```

I = These lines are the inner SignedData block, which is opaque and
    contains the ASN.1 encoded result of step 2 as well as control
    information.
| = These lines are what is encrypted in step 5. This encrypted result
    is opaque and is a part of an EnvelopedData block.
) = These lines are what the outer signature is computed over.
O = These lines are the outer SignedData block, which is opaque and
    contains the ASN.1 encoded result of step 6 as well as control
    information.

1.3 Security Services and Triple Wrapping

The first three security services described in this document are used with triple wrapped messages in different ways. This section briefly describes the relationship of each service with triple wrapping; the other sections of the document go into greater detail.

1.3.1 Signed Receipts and Triple Wrapping

A signed receipt may be requested in any SignedData object. However, if a signed receipt is requested for a triple wrapped message, the receipt request MUST be in the inside signature, not in the outside signature. A secure mailing list agent may change the receipt policy in the

outside signature of a triple wrapped message when that message is processed by the mailing list.

Note: the signed receipts and receipt requests described in this draft differ from those described in the work done by the IETF Receipt Notification Working Group. The output of that Working Group, when finished, is not expected to work well with triple wrapped messages as described in this document.

1.3.2 Security Labels and Triple Wrapping

A security label may be included in the signed attributes of any SignedData object. A security label attribute may be included in either the inner signature, outer signature, or both.

The inner security label is used for access control decisions related to the plaintext original content. The inner signature provides authentication and cryptographically protects the integrity of the original signer's security label that is in the inside body. This strategy facilitates the forwarding of messages because the original signer's security label is included in the SignedData block which can be forwarded to a third party that can verify the inner signature which will cover the inner security label. The confidentiality security service can be applied to the inner security label by encrypting the entire inner SignedData block within an EnvelopedData block.

A security label may also be included in the signed attributes of the outer SignedData block which will include the sensitivities of the encrypted message. The outer security label is used for access control and routing decisions related to the encrypted message. Note that a security label attribute can only be used in a signedAttributes block. An eSSSecurityLabel attribute MUST NOT be used in an EnvelopedData or unsigned attributes.

1.3.3 Secure Mailing Lists and Triple Wrapping

Secure mail list message processing depends on the structure of S/MIME layers present in the message sent to the mail list agent. The mail list agent never changes the data that was hashed to form the inner signature, if such a signature is present. If an outer signature is present, then the agent will modify the data that was hashed to form that outer signature. In all cases, the agent adds or updates an mlExpansionHistory attribute to document the agent's processing, and ultimately adds or replaces the outer signature on the message to be distributed.

1.3.4 Placement of Attributes

Certain attributes should be placed in the inner or outer SignedData message; some attributes can be in either. Further, some attributes must be signed, while signing is optional for others, and some

attributes must not be signed. ESS defines several types of attributes. ContentHints and ContentIdentifier MAY appear in any list of attributes. contentReference, equivalentLabel, eSSSecurityLabel and mlExpansionHistory MUST be carried in a SignedAttributes or AuthAttributes type, and MUST NOT be carried in a UnsignedAttributes, UnauthAttributes or UnprotectedAttributes type. msgSigDigest, receiptRequest and signingCertificate MUST be carried in a SignedAttributes, and MUST NOT be carried in a AuthAttributes, UnsignedAttributes, UnauthAttributes or UnprotectedAttributes type.

The following table summarizes the recommendation of this profile. In the OID column, [ESS] indicates that the attribute is defined in this document.

Attribute	OID	Inner or outer	Signed
contentHints	id-aa-contentHint [ESS]	either	MAY
contentIdentifier	id-aa-contentIdentifier [ESS]	either	MAY
contentReference	id-aa-contentReference [ESS]	either	MUST
contentType	id-contentType [CMS]	either	MUST
counterSignature	id-countersignature [CMS]	either	MUST NOT
equivalentLabel	id-aa-equivalentLabels [ESS]	either	MUST
eSSSecurityLabel	id-aa-securityLabel [ESS]	either	MUST
messageDigest	id-messageDigest [CMS]	either	MUST
msgSigDigest	id-aa-msgSigDigest [ESS]	inner only	MUST
mlExpansionHistory	id-aa-mlExpandHistory [ESS]	outer only	MUST
receiptRequest	id-aa-receiptRequest [ESS]	inner only	MUST
signingCertificate	id-aa-signingCertificate [ESS]	either	MUST
signingTime	id-signingTime [CMS]	either	MUST
smimeCapabilities	sMIMECapabilities [MSG]	either	MUST
sMIMEEncryption-KeyPreference	id-aa-encrypKeyPref [MSG]	either	MUST

CMS defines signedAttrs as a SET OF Attribute and defines unsignedAttrs as a SET OF Attribute. ESS defines the contentHints, contentIdentifier, eSSecurityLabel, msgSigDigest, mlExpansionHistory, receiptRequest, contentReference, equivalentLabels and signingCertificate attribute types. A signerInfo MUST NOT include multiple instances of any of the attribute types defined in ESS. Later sections of ESS specify further restrictions that apply to the receiptRequest, mlExpansionHistory and eSSecurityLabel attribute types.

CMS defines the syntax for the signed and unsigned attributes as "attrValues SET OF AttributeValue". For all of the attribute types defined in ESS, if the attribute type is present in a signerInfo, then it MUST only include a single instance of AttributeValue. In other words, there MUST NOT be zero, or multiple, instances of AttributeValue present in the attrValues SET OF AttributeValue.

If a counterSignature attribute is present, then it MUST be included in

the unsigned attributes. It MUST NOT be included in the signed attributes. The only attributes that are allowed in a counterSignature attribute are counterSignature, messageDigest, signingTime, and signingCertificate.

Note that the inner and outer signatures are usually those of different senders. Because of this, the same attribute in the two signatures could lead to very different consequences.

ContentIdentifier is an attribute (OCTET STRING) used to carry a unique identifier assigned to the message.

1.4 Required and Optional Attributes

Some security gateways sign messages that pass through them. If the message is any type other than a signedData type, the gateway has only one way to sign the message: by wrapping it with a signedData block and MIME headers. If the message to be signed by the gateway is a signedData message already, the gateway can sign the message by inserting a signerInfo into the signedData block.

The main advantage of a gateway adding a signerInfo instead of wrapping the message in a new signature is that the message doesn't grow as much as if the gateway wrapped the message. The main disadvantage is that the gateway must check for the presence of certain attributes in the other signerInfos and either omit or copy those attributes.

If a gateway or other processor adds a signerInfo to an existing signedData block, it MUST copy the mlExpansionHistory and eSSSecurityLabel attributes from other signerInfos. This helps ensure that the recipient will process those attributes in a signerInfo that it can verify.

Note that someone may in the future define an attribute that must be present in each signerInfo of a signedData block in order for the signature to be processed. If that happens, a gateway that inserts signerInfos and doesn't copy that attribute will cause every message with that attribute to fail when processed by the recipient. For this reason, it is safer to wrap messages with new signatures than to insert signerInfos.

1.5 Object Identifiers

The object identifiers for many of the objects described in this draft are found in [CMS], [MSG], and [CERT]. Other object identifiers used in S/MIME can be found in the registry kept at <http://www.imc.org/ietf-smime/oids.html>. When this draft moves to standards track within the IETF, it is intended that the IANA will maintain this registry.

2. Signed Receipts

Returning a signed receipt provides to the originator proof of delivery of a message, and allows the originator to demonstrate to a third party that the recipient was able to verify the signature of the original message. This receipt is bound to the original message through the signature; consequently, this service may be requested only if a message is signed. The receipt sender may optionally also encrypt a receipt to provide confidentiality between the receipt sender and the receipt recipient.

2.1 Signed Receipt Concepts

The originator of a message may request a signed receipt from the message's recipients. The request is indicated by adding a receiptRequest attribute to the signedAttributes field of the SignerInfo object for which the receipt is requested. The receiving user agent software SHOULD automatically create a signed receipt when requested to do so, and return the receipt in accordance with mailing list expansion options, local security policies, and configuration options.

Because receipts involve the interaction of two parties, the terminology can sometimes be confusing. In this section, the "sender" is the agent that sent the original message that included a request for a receipt. The "receiver" is the party that received that message and generated the receipt.

The steps in a typical transaction are:

1. Sender creates a signed message including a receipt request attribute (Section 2.2).

2. Sender transmits the resulting message to the recipient or recipients.

3. Recipient receives message and determines if there is a valid signature and receipt request in the message (Section 2.3).

4. Recipient creates a signed receipt (Section 2.4).

5. Recipient transmits the resulting signed receipt message to the sender (Section 2.5).

6. Sender receives the message and validates that it contains a signed receipt for the original message (Section 2.6). This validation relies on the sender having retained either a copy of the original message or information extracted from the original message.

The ASN.1 syntax for the receipt request is given in Section 2.7; the ASN.1 syntax for the receipt is given in Section 2.8.

Note that a sending agent SHOULD remember when it has sent a receipt so that it can avoid re-sending a receipt each time it processes the message.

A receipt request can indicate that receipts be sent to many places, not just to the sender (in fact, the receipt request might indicate that the receipts should not even go to the sender). In order to verify a receipt, the recipient of the receipt must be the originator or a recipient of the original message. Thus, the sender SHOULD NOT request that receipts be sent to anyone who does not have an exact copy of the message.

2.2 Receipt Request Creation

Multi-layer S/MIME messages may contain multiple SignedData layers. However, receipts may be requested only for the innermost SignedData layer in a multi-layer S/MIME message, such as a triple wrapped message. Only one receiptRequest attribute can be included in the signedAttributes of a SignerInfo.

A ReceiptRequest attribute MUST NOT be included in the attributes of a SignerInfo in a SignedData object that encapsulates a Receipt content. In other words, the receiving agent MUST NOT request a signed receipt for a signed receipt.

A sender requests receipts by placing a receiptRequest attribute in the signed attributes of a signerInfo as follows:

1. A receiptRequest data structure is created.

2. A signed content identifier for the message is created and assigned to the signedContentIdentifier field. The signedContentIdentifier is used to associate the signed receipt with the message requesting the signed receipt.

3. The entities requested to return a signed receipt are noted in the receiptsFrom field.

4. The message originator MUST populate the receiptsTo field with a GeneralNames for each entity to whom the recipient should send the signed receipt. If the message originator wants the recipient to send the signed receipt to the originator, then the originator MUST include a GeneralNames for itself in the receiptsTo field. GeneralNames is a SEQUENCE OF GeneralName. receiptsTo is a SEQUENCE OF GeneralNames in which each GeneralNames represents an entity. There may be multiple GeneralName instances in each GeneralNames. At a minimum, the message originator MUST populate each entity's GeneralNames with the address to which the signed receipt should be sent. Optionally, the message originator MAY also populate each entity's GeneralNames with other GeneralName instances (such as directoryName).

5. The completed receiptRequest attribute is placed in the signedAttributes field of the SignerInfo object.

2.2.1 Multiple Receipt Requests

There can be multiple SignerInfos within a SignedData object, and each SignerInfo may include signedAttributes. Therefore, a single SignedData object may include multiple SignerInfos, each SignerInfo having a receiptRequest attribute. For example, an originator can send a signed message with two SignerInfos, one containing a DSS signature, the other containing an RSA signature.

Each recipient SHOULD return only one signed receipt.

Not all of the SignerInfos need to include receipt requests, but in all of the SignerInfos that do contain receipt requests, the receipt requests MUST be identical.

2.2.2 Information Needed to Validate Signed Receipts

The sending agent MUST retain one or both of the following items to support the validation of signed receipts returned by the recipients.

- the original signedData object requesting the signed receipt

- the message signature digest value used to generate the original signedData signerInfo signature value and the digest value of the Receipt content containing values included in the original signedData object. If signed receipts are requested from multiple recipients, then retaining these digest values is a performance enhancement because the sending agent can reuse the saved values when verifying each returned signed receipt.

2.3 Receipt Request Processing

A receiptRequest is associated only with the SignerInfo object to which the receipt request attribute is directly attached. Receiving software SHOULD examine the signedAttributes field of each of the SignerInfos for which it verifies a signature in the innermost signedData object to determine if a receipt is requested. This may result in the receiving agent processing multiple receiptRequest attributes included in a single SignedData object, such as requests made from different people who signed the object in parallel.

Before processing a receiptRequest signedAttribute, the receiving agent MUST verify the signature of the SignerInfo which covers the receiptRequest attribute. A recipient MUST NOT process a receiptRequest attribute that has not been verified. Because all receiptRequest attributes in a SignedData object must be identical, the receiving application fully processes (as described in the following paragraphs)

the first receiptRequest attribute that it encounters in a SignerInfo that it verifies, and it then ensures that all other receiptRequest attributes in signerInfos that it verifies are identical to the first one encountered. If there are verified ReceiptRequest attributes which are not the same, then the processing software MUST NOT return any signed receipt. A signed receipt SHOULD be returned if any signerInfo containing a receiptRequest attribute can be validated, even if other signerInfos containing the same receiptRequest attribute cannot be validated because they are signed using an algorithm not supported by the receiving agent.

If a receiptRequest attribute is absent from the signed attributes, then a signed receipt has not been requested from any of the message recipients and MUST NOT be created. If a receiptRequest attribute is present in the signed attributes, then a signed receipt has been requested from some or all of the message recipients. Note that in some cases, a receiving agent might receive two almost-identical messages, one with a receipt request and the other without one. In this case, the receiving agent SHOULD send a signed receipt for the message that requests a signed receipt.

If a receiptRequest attribute is present in the signed attributes, the following process SHOULD be used to determine if a message recipient has been requested to return a signed receipt.

1. If an mlExpansionHistory attribute is present in the outermost signedData block, do one of the following two steps, based on the absence or presence of mlReceiptPolicy:

    1.1. If an mlReceiptPolicy value is absent from the last MLData element, a Mail List receipt policy has not been specified and the processing software SHOULD examine the receiptRequest attribute value to determine if a receipt should be created and returned.

    1.2. If an mlReceiptPolicy value is present in the last MLData element, do one of the following two steps, based on the value of mlReceiptPolicy:

        1.2.1. If the mlReceiptPolicy value is none, then the receipt policy of the Mail List supersedes the originator's request for a signed receipt and a signed receipt MUST NOT be created.

        1.2.2. If the mlReceiptPolicy value is insteadOf or inAdditionTo, the processing software SHOULD examine the receiptsFrom value from the receiptRequest attribute to determine if a receipt should be created and returned. If a receipt is created, the insteadOf and inAdditionTo fields identify entities that SHOULD be sent the receipt instead of or in addition to the originator.

2. If the receiptsFrom value of the receiptRequest attribute is

allOrFirstTier, do one of the following two steps based on the value of allOrFirstTier.

2.1. If the value of allOrFirstTier is allReceipts, then a signed receipt SHOULD be created.

2.2. If the value of allOrFirstTier is firstTierRecipients, do one of the following two steps based on the presence of an mlExpansionHistory attribute in an outer signedData block:

2.2.1. If an mlExpansionHistory attribute is present, then this recipient is not a first tier recipient and a signed receipt MUST NOT be created.

2.2.2. If an mlExpansionHistory attribute is not present, then a signed receipt SHOULD be created.

3. If the receiptsFrom value of the receiptRequest attribute is a receiptList:

3.1. If receiptList contains one of the GeneralNames of the recipient, then a signed receipt SHOULD be created.

3.2. If receiptList does not contain one of the GeneralNames of the recipient, then a signed receipt MUST NOT be created.

A flow chart for the above steps to be executed for each signerInfo for which the receiving agent verifies the signature would be:

```
0. Receipt Request attribute present?
       YES -> 1.
       NO  -> STOP
1. Has mlExpansionHistory in outer signedData?
       YES -> 1.1.
       NO  -> 2.
1.1. mlReceiptPolicy absent?
       YES -> 2.
       NO  -> 1.2.
1.2. Pick based on value of mlReceiptPolicy.
       none -> 1.2.1.
       insteadOf or inAdditionTo -> 1.2.2.
1.2.1. STOP.
1.2.2. Examine receiptsFrom to determine if a receipt should be created,
    create it if required, send it to recipients designated by
    mlReceiptPolicy, then -> STOP.
2. Is value of receiptsFrom allOrFirstTier?
       YES -> Pick based on value of allOrFirstTier.
             allReceipts -> 2.1.
             firstTierRecipients -> 2.2.
       NO  -> 3.
2.1. Create a receipt, then -> STOP.
```

2.2. Has mlExpansionHistory in the outer signedData block?
       YES -> 2.2.1.
       NO -> 2.2.2.
2.2.1. STOP.
2.2.2. Create a receipt, then -> STOP.
3. Is receiptsFrom value of receiptRequest a receiptList?
       YES -> 3.1.
       NO -> STOP.
3.1. Does receiptList contain the recipient?
       YES -> Create a receipt, then -> STOP.
       NO -> 3.2.
3.2. STOP.

2.4 Signed Receipt Creation

A signed receipt is a signedData object encapsulating a Receipt content (also called a "signedData/Receipt"). Signed receipts are created as follows:

1. The signature of the original signedData signerInfo that includes the receiptRequest signed attribute MUST be successfully verified before creating the signedData/Receipt.

    1.1. The content of the original signedData object is digested as described in [CMS]. The resulting digest value is then compared with the value of the messageDigest attribute included in the signedAttributes of the original signedData signerInfo. If these digest values are different, then the signature verification process fails and the signedData/Receipt MUST NOT be created.

    1.2. The ASN.1 DER encoded signedAttributes (including messageDigest, receiptRequest and, possibly, other signed attributes) in the original signedData signerInfo are digested as described in [CMS]. The resulting digest value, called msgSigDigest, is then used to verify the signature of the original signedData signerInfo. If the signature verification fails, then the signedData/Receipt MUST NOT be created.

2. A Receipt structure is created.

    2.1. The value of the Receipt version field is set to 1.

    2.2. The object identifier from the contentType attribute included in the original signedData signerInfo that includes the receiptRequest attribute is copied into the Receipt contentType.

    2.3. The original signedData signerInfo receiptRequest signedContentIdentifier is copied into the Receipt signedContentIdentifier.

    2.4. The signature value from the original signedData signerInfo

that includes the receiptRequest attribute is copied into the Receipt originatorSignatureValue.

3. The Receipt structure is ASN.1 DER encoded to produce a data stream, D1.

4. D1 is digested. The resulting digest value is included as the messageDigest attribute in the signedAttributes of the signerInfo which will eventually contain the signedData/Receipt signature value.

5. The digest value (msgSigDigest) calculated in Step 1 to verify the signature of the original signedData signerInfo is included as the msgSigDigest attribute in the signedAttributes of the signerInfo which will eventually contain the signedData/Receipt signature value.

6. A contentType attribute including the id-ct-receipt object identifier MUST be created and added to the signed attributes of the signerInfo which will eventually contain the signedData/Receipt signature value.

7. A signingTime attribute indicating the time that the signedData/Receipt is signed SHOULD be created and added to the signed attributes of the signerInfo which will eventually contain the signedData/Receipt signature value. Other attributes (except receiptRequest) may be added to the signedAttributes of the signerInfo.

8. The signedAttributes (messageDigest, msgSigDigest, contentType and, possibly, others) of the signerInfo are ASN.1 DER encoded and digested as described in [CMS]. The resulting digest value is used to calculate the signature value which is then included in the signedData/Receipt signerInfo.

9. The ASN.1 DER encoded Receipt content MUST be directly encoded within the signedData encapContentInfo eContent OCTET STRING defined in [CMS]. The id-ct-receipt object identifier MUST be included in the signedData encapContentInfo eContentType. This results in a single ASN.1 encoded object composed of a signedData including the Receipt content. The Data content type MUST NOT be used. The Receipt content MUST NOT be encapsulated in a MIME header or any other header prior to being encoded as part of the signedData object.

10. The signedData/Receipt is then put in an application/pkcs7-mime MIME wrapper with the smime-type parameter set to "signed-receipt". This will allow for identification of signed receipts without having to crack the ASN.1 body. The smime-type parameter would still be set as normal in any layer wrapped around this message.

11. If the signedData/Receipt is to be encrypted within an envelopedData object, then an outer signedData object MUST be created that encapsulates the envelopedData object, and a contentHints attribute with contentType set to the id-ct-receipt object identifier

MUST be included in the outer signedData SignerInfo signedAttributes. When a receiving agent processes the outer signedData object, the presence of the id-ct-receipt OID in the contentHints contentType indicates that a signedData/Receipt is encrypted within the envelopedData object encapsulated by the outer signedData.

All sending agents that support the generation of ESS signed receipts MUST provide the ability to send encrypted signed receipts (that is, a signedData/Receipt encapsulated within an envelopedData). The sending agent MAY send an encrypted signed receipt in response to an envelopedData-encapsulated signedData requesting a signed receipt. It is a matter of local policy regarding whether or not the signed receipt should be encrypted. The ESS signed receipt includes the message digest value calculated for the original signedData object that requested the signed receipt. If the original signedData object was sent encrypted within an envelopedData object and the ESS signed receipt is sent unencrypted, then the message digest value calculated for the original encrypted signedData object is sent unencrypted. The responder should consider this when deciding whether or not to encrypt the ESS signed receipt.

2.4.1 MLExpansionHistory Attributes and Receipts

An MLExpansionHistory attribute MUST NOT be included in the attributes of a SignerInfo in a SignedData object that encapsulates a Receipt content. This is true because when a SignedData/Receipt is sent to an MLA for distribution, then the MLA must always encapsulate the received SignedData/Receipt in an outer SignedData in which the MLA will include the MLExpansionHistory attribute. The MLA cannot change the signedAttributes of the received SignedData/Receipt object, so it can't add the MLExpansionHistory to the SignedData/Receipt.

2.5 Determining the Recipients of the Signed Receipt

If a signed receipt was created by the process described in the sections above, then the software MUST use the following process to determine to whom the signed receipt should be sent.

1. The receiptsTo field must be present in the receiptRequest attribute. The software initiates the sequence of recipients with the value(s) of receiptsTo.

2. If the MlExpansionHistory attribute is present in the outer SignedData block, and the last MLData contains an MLReceiptPolicy value of insteadOf, then the software replaces the sequence of recipients with the value(s) of insteadOf.

3. If the MlExpansionHistory attribute is present in the outer SignedData block and the last MLData contains an MLReceiptPolicy value of inAdditionTo, then the software adds the value(s) of inAdditionTo to the sequence of recipients.

2.6. Signed Receipt Validation

A signed receipt is communicated as a single ASN.1 encoded object composed of a signedData object directly including a Receipt content. It is identified by the presence of the id-ct-receipt object identifier in the encapContentInfo eContentType value of the signedData object including the Receipt content.

Although recipients are not supposed to send more than one signed receipt, receiving agents SHOULD be able to accept multiple signed receipts from a recipient.

A signedData/Receipt is validated as follows:

1. ASN.1 decode the signedData object including the Receipt content.

2. Extract the contentType, signedContentIdentifier, and originatorSignatureValue from the decoded Receipt structure to identify the original signedData signerInfo that requested the signedData/Receipt.

3. Acquire the message signature digest value calculated by the sender to generate the signature value included in the original signedData signerInfo that requested the signedData/Receipt.

    3.1. If the sender-calculated message signature digest value has been saved locally by the sender, it must be located and retrieved.

    3.2. If it has not been saved, then it must be re-calculated based on the original signedData content and signedAttributes as described in [CMS].

4. The message signature digest value calculated by the sender is then compared with the value of the msgSigDigest signedAttribute included in the signedData/Receipt signerInfo. If these digest values are identical, then that proves that the message signature digest value calculated by the recipient based on the received original signedData object is the same as that calculated by the sender. This proves that the recipient received exactly the same original signedData content and signedAttributes as sent by the sender because that is the only way that the recipient could have calculated the same message signature digest value as calculated by the sender. If the digest values are different, then the signedData/Receipt signature verification process fails.

5. Acquire the digest value calculated by the sender for the Receipt content constructed by the sender (including the contentType, signedContentIdentifier, and signature value that were included in the original signedData signerInfo that requested the signedData/Receipt).

5.1. If the sender-calculated Receipt content digest value has been saved locally by the sender, it must be located and retrieved.

5.2. If it has not been saved, then it must be re-calculated. As described in section 2.4 above, step 2, create a Receipt structure including the contentType, signedContentIdentifier and signature value that were included in the original signedData signerInfo that requested the signed receipt. The Receipt structure is then ASN.1 DER encoded to produce a data stream which is then digested to produce the Receipt content digest value.

6. The Receipt content digest value calculated by the sender is then compared with the value of the messageDigest signedAttribute included in the signedData/Receipt signerInfo. If these digest values are identical, then that proves that the values included in the Receipt content by the recipient are identical to those that were included in the original signedData signerInfo that requested the signedData/Receipt. This proves that the recipient received the original signedData signed by the sender, because that is the only way that the recipient could have obtained the original signedData signerInfo signature value for inclusion in the Receipt content. If the digest values are different, then the signedData/Receipt signature verification process fails.

7. The ASN.1 DER encoded signedAttributes of the signedData/Receipt signerInfo are digested as described in [CMS].

8. The resulting digest value is then used to verify the signature value included in the signedData/Receipt signerInfo. If the signature verification is successful, then that proves the integrity of the signedData/receipt signerInfo signedAttributes and authenticates the identity of the signer of the signedData/Receipt signerInfo. Note that the signedAttributes include the recipient-calculated Receipt content digest value (messageDigest attribute) and recipient-calculated message signature digest value (msgSigDigest attribute). Therefore, the aforementioned comparison of the sender-generated and recipient-generated digest values combined with the successful signedData/Receipt signature verification proves that the recipient received the exact original signedData content and signedAttributes (proven by msgSigDigest attribute) that were signed by the sender of the original signedData object (proven by messageDigest attribute). If the signature verification fails, then the signedData/Receipt signature verification process fails.

The signature verification process for each signature algorithm that is used in conjunction with the CMS protocol is specific to the algorithm. These processes are described in documents specific to the algorithms.

2.7 Receipt Request Syntax

A receiptRequest attribute value has ASN.1 type ReceiptRequest. Use the

receiptRequest attribute only within the signed attributes associated with a signed message.

```
ReceiptRequest ::= SEQUENCE {
  signedContentIdentifier ContentIdentifier,
  receiptsFrom ReceiptsFrom,
  receiptsTo SEQUENCE SIZE (1..ub-receiptsTo)) OF GeneralNames }

ub-receiptsTo INTEGER ::= 16

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

ContentIdentifier ::= OCTET STRING

id-aa-contentIdentifier OBJECT IDENTIFIER ::= { iso(1) member-body(2)
    us(840) rsadsi(113549) pkcs(1) pkcs-9(9) smime(16) id-aa(2) 7}
```

A signedContentIdentifier MUST be created by the message originator when creating a receipt request. To ensure global uniqueness, the minimal signedContentIdentifier SHOULD contain a concatenation of user-specific identification information (such as a user name or public keying material identification information), a GeneralizedTime string, and a random number.

The receiptsFrom field is used by the originator to specify the recipients requested to return a signed receipt. A CHOICE is provided to allow specification of:

- receipts from all recipients are requested
- receipts from first tier (recipients that did not receive the message as members of a mailing list) recipients are requested
- receipts from a specific list of recipients are requested

```
ReceiptsFrom ::= CHOICE {
  allOrFirstTier [0] AllOrFirstTier,
  -- formerly "allOrNone [0]AllOrNone"
  receiptList [1] SEQUENCE OF GeneralNames }

AllOrFirstTier ::= INTEGER { -- Formerly AllOrNone
  allReceipts (0),
  firstTierRecipients (1) }
```

The receiptsTo field is used by the originator to identify the user(s) to whom the identified recipient should send signed receipts. The message originator MUST populate the receiptsTo field with a GeneralNames for each entity to whom the recipient should send the signed receipt. If the message originator wants the recipient to send the signed receipt to the originator, then the originator MUST include a GeneralNames for itself in the receiptsTo field.

2.8 Receipt Syntax

Receipts are represented using a new content type, Receipt. The Receipt content type shall have ASN.1 type Receipt. Receipts must be encapsulated within a SignedData message.

```
Receipt ::= SEQUENCE {
  version ESSVersion,
  contentType ContentType,
  signedContentIdentifier ContentIdentifier,
  originatorSignatureValue OCTET STRING }

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

ESSVersion ::= INTEGER  { v1(1) }
```

The version field defines the syntax version number, which is 1 for this version of the standard.

2.9 Content Hints

Many applications find it useful to have information that describes the innermost signed content of a multi-layer message available on the outermost signature layer. The contentHints attribute provides such information.

Content-hints attribute values have ASN.1 type contentHints.

```
ContentHints ::= SEQUENCE {
  contentDescription UTF8String (SIZE (1..MAX)) OPTIONAL,
  contentType ContentType }

id-aa-contentHint OBJECT IDENTIFIER ::= { iso(1) member-body(2) us(840)
    rsadsi(113549) pkcs(1) pkcs-9(9) smime(16) id-aa(2) 4}
```

The contentDescription field may be used to provide information that the recipient may use to select protected messages for processing, such as a message subject. If this field is set, then the attribute is expected to appear on the signedData object enclosing an envelopedData object and not on the inner signedData object. The (SIZE (1..MAX)) construct constrains the sequence to have at least one entry. MAX indicates the upper bound is unspecified. Implementations are free to choose an upper bound that suits their environment.

Messages which contain a signedData object wrapped around an envelopedData object, thus masking the inner content type of the message, SHOULD include a contentHints attribute, except for the case of the data content type. Specific message content types may either force or preclude the inclusion of the contentHints attribute. For example, when a signedData/Receipt is encrypted within an envelopedData object, an outer signedData object MUST be created that encapsulates

the envelopedData object and a contentHints attribute with contentType set to the id-ct-receipt object identifier MUST be included in the outer signedData SignerInfo signedAttributes.

2.10 Message Signature Digest Attribute

The msgSigDigest attribute can only be used in the signed attributes of a signed receipt. It contains the digest of the ASN.1 DER encoded signedAttributes included in the original signedData that requested the signed receipt. Only one msgSigDigest attribute can appear in a signed attributes set. It is defined as follows:

```
msgSigDigest ::= OCTET STRING

id-aa-msgSigDigest OBJECT IDENTIFIER ::= { iso(1) member-body(2)
    us(840) rsadsi(113549) pkcs(1) pkcs-9(9) smime(16) id-aa(2) 5}
```

2.11 Signed Content Reference Attribute

The contentReference attribute is a link from one SignedData to another. It may be used to link a reply to the original message to which it refers, or to incorporate by reference one SignedData into another. The first SignedData MUST include a contentIdentifier signed attribute, which SHOULD be constructed as specified in section 2.7. The second SignedData links to the first by including a ContentReference signed attribute containing the content type, content identifier, and signature value from the first SignedData.

```
ContentReference ::= SEQUENCE {
  contentType ContentType,
  signedContentIdentifier ContentIdentifier,
  originatorSignatureValue OCTET STRING }

id-aa-contentReference   OBJECT IDENTIFIER ::= { iso(1) member-body(2)
    us(840) rsadsi(113549) pkcs(1) pkcs-9(9) smime(16) id-aa(2) 10 }
```

3. Security Labels

This section describes the syntax to be used for security labels that can optionally be associated with S/MIME encapsulated data. A security label is a set of security information regarding the sensitivity of the content that is protected by S/MIME encapsulation.

"Authorization" is the act of granting rights and/or privileges to users permitting them access to an object. "Access control" is a means of enforcing these authorizations. The sensitivity information in a security label can be compared with a user's authorizations to determine if the user is allowed to access the content that is protected by S/MIME encapsulation.

Security labels may be used for other purposes such as a source of routing information. The labels often describe ranked levels ("secret", "confidential", "restricted", and so on) or are role-based, describing which kind of people can see the information ("patient's health-care team", "medical billing agents", "unrestricted", and so on).

3.1 Security Label Processing Rules

A sending agent may include a security label attribute in the signed attributes of a signedData object. A receiving agent examines the security label on a received message and determines whether or not the recipient is allowed to see the contents of the message.

3.1.1 Adding Security Labels

A sending agent that is using security labels MUST put the security label attribute in the signedAttributes field of a SignerInfo block. The security label attribute MUST NOT be included in the unsigned attributes. Integrity and authentication security services MUST be applied to the security label, therefore it MUST be included as a signed attribute, if used. This causes the security label attribute to be part of the data that is hashed to form the SignerInfo signature value. A SignerInfo block MUST NOT have more than one security label signed attribute.

When there are multiple SignedData blocks applied to a message, a security label attribute may be included in either the inner signature, outer signature, or both. A security label signed attribute may be included in a signedAttributes field within the inner SignedData block. The inner security label will include the sensitivities of the original content and will be used for access control decisions related to the plaintext encapsulated content. The inner signature provides authentication of the inner security label and cryptographically protects the original signer's inner security label of the original content.

When the originator signs the plaintext content and signed attributes, the inner security label is bound to the plaintext content. An intermediate entity cannot change the inner security label without invalidating the inner signature. The confidentiality security service can be applied to the inner security label by encrypting the entire inner signedData object within an EnvelopedData block.

A security label signed attribute may also be included in a signedAttributes field within the outer SignedData block. The outer security label will include the sensitivities of the encrypted message and will be used for access control decisions related to the encrypted message and for routing decisions. The outer signature provides authentication of the outer security label (as well as for the encapsulated content which may include nested S/MIME messages).

There can be multiple SignerInfos within a SignedData object, and each SignerInfo may include signedAttributes. Therefore, a single SignedData object may include multiple eSSSecurityLabels, each SignerInfo having an eSSSecurityLabel attribute. For example, an originator can send a signed message with two SignerInfos, one containing a DSS signature, the other containing an RSA signature. If any of the SignerInfos included in a SignedData object include an eSSSecurityLabel attribute, then all of the SignerInfos in that SignedData object MUST include an eSSSecurityLabel attribute and the value of each MUST be identical.

3.1.2 Processing Security Labels

Before processing an eSSSecurityLabel signedAttribute, the receiving agent MUST verify the signature of the SignerInfo which covers the eSSSecurityLabel attribute. A recipient MUST NOT process an eSSSecurityLabel attribute that has not been verified.

A receiving agent MUST process the eSSSecurityLabel attribute, if present, in each SignerInfo in the SignedData object for which it verifies the signature. This may result in the receiving agent processing multiple eSSSecurityLabels included in a single SignedData object. Because all eSSSecurityLabels in a SignedData object must be identical, the receiving agent processes (such as performing access control) on the first eSSSecurityLabel that it encounters in a SignerInfo that it verifies, and then ensures that all other eSSSecurityLabels in signerInfos that it verifies are identical to the first one encountered. If the eSSSecurityLabels in the signerInfos that it verifies are not all identical, then the receiving agent MUST warn the user of this condition.

Receiving agents SHOULD have a local policy regarding whether or not to show the inner content of a signedData object that includes an eSSSecurityLabel security-policy-identifier that the processing software does not recognize. If the receiving agent does not recognize the eSSSecurityLabel security-policy-identifier value, then it SHOULD stop processing the message and indicate an error.

3.2 Syntax of eSSSecurityLabel

The eSSSecurityLabel syntax is derived directly from [MTSABS] ASN.1 module. (The MTSAbstractService module begins with "DEFINITIONS IMPLICIT TAGS ::=".) Further, the eSSSecurityLabel syntax is compatible with that used in [MSP4].

```
ESSSecurityLabel ::= SET {
  security-policy-identifier SecurityPolicyIdentifier,
  security-classification SecurityClassification OPTIONAL,
  privacy-mark ESSPrivacyMark OPTIONAL,
  security-categories SecurityCategories OPTIONAL }

id-aa-securityLabel OBJECT IDENTIFIER ::= { iso(1) member-body(2)
```

```
    us(840) rsadsi(113549) pkcs(1) pkcs-9(9) smime(16) id-aa(2) 2}

SecurityPolicyIdentifier ::= OBJECT IDENTIFIER

SecurityClassification ::= INTEGER {
  unmarked (0),
  unclassified (1),
  restricted (2),
  confidential (3),
  secret (4),
  top-secret (5) } (0..ub-integer-options)

ub-integer-options INTEGER ::= 256

ESSPrivacyMark ::= CHOICE {
    pString      PrintableString (SIZE (1..ub-privacy-mark-length)),
    utf8String   UTF8String (SIZE (1..MAX))
}

ub-privacy-mark-length INTEGER ::= 128

SecurityCategories ::= SET SIZE (1..ub-security-categories) OF
        SecurityCategory

ub-security-categories INTEGER ::= 64

SecurityCategory ::= SEQUENCE {
  type  [0] OBJECT IDENTIFIER,
  value [1] ANY DEFINED BY type -- defined by type
}

--Note: The aforementioned SecurityCategory syntax produces identical
--hex encodings as the following SecurityCategory syntax that is
--documented in the X.411 specification:
--
--SecurityCategory ::= SEQUENCE {
--     type  [0]  SECURITY-CATEGORY,
--     value [1]  ANY DEFINED BY type }
--
--SECURITY-CATEGORY MACRO ::=
--BEGIN
--TYPE NOTATION ::= type | empty
--VALUE NOTATION ::= value (VALUE OBJECT IDENTIFIER)
--END
```

3.3 Security Label Components

This section gives more detail on the the various components of the eSSSecurityLabel syntax.

3.3.1 Security Policy Identifier

A security policy is a set of criteria for the provision of security services. The eSSSecurityLabel security-policy-identifier is used to identify the security policy in force to which the security label relates. It indicates the semantics of the other security label components.

3.3.2 Security Classification

This specification defines the use of the Security Classification field exactly as is specified in the X.411 Recommendation, which states in part:

> If present, a security-classification may have one of a hierarchical list of values. The basic security-classification hierarchy is defined in this Recommendation, but the use of these values is defined by the security-policy in force. Additional values of security-classification, and their position in the hierarchy, may also be defined by a security-policy as a local matter or by bilateral agreement. The basic security-classification hierarchy is, in ascending order: unmarked, unclassified, restricted, confidential, secret, top-secret.

This means that the security policy in force (identified by the eSSSecurityLabel security-policy-identifier) defines the SecurityClassification integer values and their meanings.

An organization can develop its own security policy that defines the SecurityClassification INTEGER values and their meanings. However, the general interpretation of the X.411 specification is that the values of 0 through 5 are reserved for the "basic hierarchy" values of unmarked, unclassified, restricted, confidential, secret, and top-secret. Note that X.411 does not provide the rules for how these values are used to label data and how access control is performed using these values.

There is no universal definition of the rules for using these "basic hierarchy" values. Each organization (or group of organizations) will define a security policy which documents how the "basic hierarchy" values are used (if at all) and how access control is enforced (if at all) within their domain.

Therefore, the security-classification value MUST be accompanied by a security-policy-identifier value to define the rules for its use. For example, a company's "secret" classification may convey a different meaning than the US Government "secret" classification. In summary, a security policy SHOULD NOT use integers 0 through 5 for other than their X.411 meanings, and SHOULD instead use other values in a hierarchical fashion.

Note that the set of valid security-classification values MUST be hierarchical, but these values do not necessarily need to be in

ascending numerical order. Further, the values do not need to be contiguous.

For example, in the Defense Message System 1.0 security policy, the security-classification value of 11 indicates Sensitive-But-Unclassified and 5 indicates top-secret. The hierarchy of sensitivity ranks top-secret as more sensitive than Sensitive-But-Unclassified even though the numerical value of top-secret is less than Sensitive-But-Unclassified.

(Of course, if security-classification values are both hierarchical and in ascending order, a casual reader of the security policy is more likely to understand it.)

An example of a security policy that does not use any of the X.411 values might be:
10 -- anyone
15 -- Morgan Corporation and its contractors
20 -- Morgan Corporation employees
25 -- Morgan Corporation board of directors

An example of a security policy that uses part of the X.411 hierarchy might be:
0 -- unmarked
1 -- unclassified, can be read by everyone
2 -- restricted to Timberwolf Productions staff
6 -- can only be read to Timberwolf Productions executives

3.3.3 Privacy Mark

If present, the eSSSecurityLabel privacy-mark is not used for access control. The content of the eSSSecurityLabel privacy-mark may be defined by the security policy in force (identified by the eSSSecurityLabel security-policy-identifier) which may define a list of values to be used. Alternately, the value may be determined by the originator of the security-label.

3.3.4 Security Categories

If present, the eSSSecurityLabel security-categories provide further granularity for the sensitivity of the message. The security policy in force (identified by the eSSSecurityLabel security-policy-identifier) is used to indicate the syntaxes that are allowed to be present in the eSSSecurityLabel security-categories. Alternately, the security-categories and their values may be defined by bilateral agreement.

3.4 Equivalent Security Labels

Because organizations are allowed to define their own security policies, many different security policies will exist. Some

organizations may wish to create equivalencies between their security policies with the security policies of other organizations. For example, the Acme Company and the Widget Corporation may reach a bilateral agreement that the "Acme private" security-classification value is equivalent to the "Widget sensitive" security-classification value.

Receiving agents MUST NOT process an equivalentLabels attribute in a message if the agent does not trust the signer of that attribute to translate the original eSSSecurityLabel values to the security policy included in the equivalentLabels attribute. Receiving agents have the option to process equivalentLabels attributes but do not have to. It is acceptable for a receiving agent to only process eSSSecurityLabels. All receiving agents SHOULD recognize equivalentLabels attributes even if they do not process them.

3.4.1 Creating Equivalent Labels

The EquivalentLabels signed attribute is defined as:

```
EquivalentLabels ::= SEQUENCE OF ESSSecurityLabel

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

As stated earlier, the ESSSecurityLabel contains the sensitivity values selected by the original signer of the signedData. If an ESSSecurityLabel is present in a signerInfo, all signerInfos in the signedData MUST contain an ESSSecurityLabel and they MUST all be identical. In addition to an ESSSecurityLabel, a signerInfo MAY also include an equivalentLabels signed attribute. If present, the equivalentLabels attribute MUST include one or more security labels that are believed by the signer to be semantically equivalent to the ESSSecurityLabel attribute included in the same signerInfo.

All security-policy object identifiers MUST be unique in the set of ESSSecurityLabel and EquivalentLabels security labels. Before using an EquivalentLabels attribute, a receiving agent MUST ensure that all security-policy OIDs are unique in the security label or labels included in the EquivalentLabels. Once the receiving agent selects the security label (within the EquivalentLabels) to be used for processing, then the security-policy OID of the selected EquivalentLabels security label MUST be compared with the ESSSecurityLabel security-policy OID to ensure that they are unique.

In the case that an ESSSecurityLabel attribute is not included in a signerInfo, then an EquivalentLabels attribute may still be included. For example, in the Acme security policy, the absence of an ESSSecurityLabel could be defined to equate to a security label composed of the Acme security-policy OID and the "unmarked" security-classification.

Note that equivalentLabels MUST NOT be used to convey security labels that are semantically different from the ESSSecurityLabel included in the signerInfos in the signedData. If an entity needs to apply a security label that is semantically different from the ESSSecurityLabel, then it MUST include the sematically different security label in an outer signedData object that encapsulates the signedData object that includes the ESSSecurityLabel.

If present, the equivalentLabels attribute MUST be a signed attribute; it MUST NOT be an unsigned attribute. [CMS] defines signedAttributes as a SET OF Attribute. A signerInfo MUST NOT include multiple instances of the equivalentLabels attribute. CMS defines the ASN.1 syntax for the signed attributes to include attrValues SET OF AttributeValue. A equivalentLabels attribute MUST only include a single instance of AttributeValue. There MUST NOT be zero or multiple instances of AttributeValue present in the attrValues SET OF AttributeValue.

3.4.2 Processing Equivalent Labels

A receiving agent SHOULD process the ESSSecurityLabel before processing any EquivalentLabels. If the policy in the ESSSecurityLabel is understood by the receiving agent, it MUST process that label and MUST ignore all EquivalentLabels.

When processing an EquivalentLabels attribute, the receiving agent MUST validate the signature on the EquivalentLabels attribute. A receiving agent MUST NOT act on an equivalentLabels attribute for which the signature could not be validated, and MUST NOT act on an equivalentLabels attribute unless that attribute is signed by an entity trusted to translate the original eSSSecurityLabel values to the security policy included in the equivalentLabels attribute. Determining who is allowed to specify equivalence mappings is a local policy. If a message has more than one EquivalentLabels attribute, the receiving agent SHOULD process the first one that it reads and validates that contains the security policy of interest to the receiving agent.

4. Mail List Management

Sending agents must create recipient-specific data structures for each recipient of an encrypted message. This process can impair performance for messages sent to a large number of recipients. Thus, Mail List Agents (MLAs) that can take a single message and perform the recipient-specific encryption for every recipient are often desired.

An MLA appears to the message originator as a normal message recipient, but the MLA acts as a message expansion point for a Mail List (ML). The sender of a message directs the message to the MLA, which then redistributes the message to the members of the ML. This process offloads the per-recipient processing from individual user agents and

allows for more efficient management of large MLs. MLs are true message recipients served by MLAs that provide cryptographic and expansion services for the mailing list.

In addition to cryptographic handling of messages, secure mailing lists also have to prevent mail loops. A mail loop is where one mailing list is a member of a second mailing list, and the second mailing list is a member of the first. A message will go from one list to the other in a rapidly-cascading succession of mail that will be distributed to all other members of both lists.

To prevent mail loops, MLAs use the mlExpansionHistory attribute of the outer signature of a triple wrapped message. The mlExpansionHistory attribute is essentially a list of every MLA that has processed the message. If an MLA sees its own unique entity identifier in the list, it knows that a loop has been formed, and does not send the message to the list again.

4.1 Mail List Expansion

Mail list expansion processing is noted in the value of the mlExpansionHistory attribute, located in the signed attributes of the MLA's SignerInfo block. The MLA creates or updates the signed mlExpansionHistory attribute value each time the MLA expands and signs a message for members of a mail list.

The MLA MUST add an MLData record containing the MLA's identification information, date and time of expansion, and optional receipt policy to the end of the mail list expansion history sequence. If the mlExpansionHistory attribute is absent, then the MLA MUST add the attribute and the current expansion becomes the first element of the sequence. If the mlExpansionHistory attribute is present, then the MLA MUST add the current expansion information to the end of the existing MLExpansionHistory sequence. Only one mlExpansionHistory attribute can be included in the signedAttributes of a SignerInfo.

Note that if the mlExpansionHistory attribute is absent, then the recipient is a first tier message recipient.

There can be multiple SignerInfos within a SignedData object, and each SignerInfo may include signedAttributes. Therefore, a single SignedData object may include multiple SignerInfos, each SignerInfo having a mlExpansionHistory attribute. For example, an MLA can send a signed message with two SignerInfos, one containing a DSS signature, the other containing an RSA signature.

If an MLA creates a SignerInfo that includes an mlExpansionHistory attribute, then all of the SignerInfos created by the MLA for that SignedData object MUST include an mlExpansionHistory attribute, and the value of each MUST be identical. Note that other agents might later add SignerInfo attributes to the SignedData block, and those additional

SignerInfos might not include mlExpansionHistory attributes.

A recipient MUST verify the signature of the SignerInfo which covers the mlExpansionHistory attribute before processing the mlExpansionHistory, and MUST NOT process the mlExpansionHistory attribute unless the signature over it has been verified. If a SignedData object has more than one SignerInfo that has an mlExpansionHistory attribute, the recipient MUST compare the mlExpansionHistory attributes in all the SignerInfos that it has verified, and MUST NOT process the mlExpansionHistory attribute unless every verified mlExpansionHistory attribute in the SignedData block is identical. If the mlExpansionHistory attributes in the verified signerInfos are not all identical, then the receiving agent MUST stop processing the message and SHOULD notify the user or MLA administrator of this error condition. In the mlExpansionHistory processing, SignerInfos that do not have an mlExpansionHistory attribute are ignored.

4.1.1 Detecting Mail List Expansion Loops

Prior to expanding a message, the MLA examines the value of any existing mail list expansion history attribute to detect an expansion loop. An expansion loop exists when a message expanded by a specific MLA for a specific mail list is redelivered to the same MLA for the same mail list.

Expansion loops are detected by examining the mailListIdentifier field of each MLData entry found in the mail list expansion history. If an MLA finds its own identification information, then the MLA must discontinue expansion processing and should provide warning of an expansion loop to a human mail list administrator. The mail list administrator is responsible for correcting the loop condition.

4.2 Mail List Agent Processing

The first few paragraphs of this section provide a high-level description of MLA processing. The rest of the section provides a detailed description of MLA processing.

MLA message processing depends on the structure of the S/MIME layers in the message sent to the MLA for expansion. In addition to sending triple wrapped messages to an MLA, an entity can send other types of messages to an MLA, such as:

- a single wrapped signedData or envelopedData message
- a double wrapped message (such as signed and enveloped, enveloped and signed, or signed and signed, and so on)
- a quadruple-wrapped message (such as if a well-formed triple wrapped message was sent through a gateway that added an outer SignedData layer)

In all cases, the MLA MUST parse all layers of the received message to determine if there are any signedData layers that include an

eSSSecurityLabel signedAttribute. This may include decrypting an EnvelopedData layer to determine if an encapsulated SignedData layer includes an eSSSecurityLabel attribute. The MLA MUST fully process each eSSSecurityLabel attribute found in the various signedData layers, including performing access control checks, before distributing the message to the ML members. The details of the access control checks are beyond the scope of this document. The MLA MUST verify the signature of the signerInfo including the eSSSecurityLabel attribute before using it.

In all cases, the MLA MUST sign the message to be sent to the ML members in a new "outer" signedData layer. The MLA MUST add or update an mlExpansionHistory attribute in the "outer" signedData that it creates to document MLA processing. If there was an "outer" signedData layer included in the original message received by the MLA, then the MLA-created "outer" signedData layer MUST include each signed attribute present in the original "outer" signedData layer, unless the MLA explicitly replaces an attribute (such as signingTime or mlExpansionHistory) with a new value.

When an S/MIME message is received by the MLA, the MLA MUST first determine which received signedData layer, if any, is the "outer" signedData layer. To identify the received "outer" signedData layer, the MLA MUST verify the signature and fully process the signedAttributes in each of the outer signedData layers (working from the outside in) to determine if any of them either include an mlExpansionHistory attribute or encapsulate an envelopedData object.

The MLA's search for the "outer" signedData layer is completed when it finds one of the following:

- the "outer" signedData layer that includes an mlExpansionHistory attribute or encapsulates an envelopedData object
- an envelopedData layer
- the original content (that is, a layer that is neither envelopedData nor signedData).

If the MLA finds an "outer" signedData layer, then the MLA MUST perform the following steps:

1. Strip off all of the signedData layers that encapsulated the "outer" signedData layer

2. Strip off the "outer" signedData layer itself (after remembering the included signedAttributes)

3. Expand the envelopedData (if present)

4. Sign the message to be sent to the ML members in a new "outer" signedData layer that includes the signedAttributes (unless explicitly replaced) from the original, received "outer" signedData layer.

If the MLA finds an "outer" signedData layer that includes an mlExpansionHistory attribute AND the MLA subsequently finds an

envelopedData layer buried deeper with the layers of the received message, then the MLA MUST strip off all of the signedData layers down to the envelopedData layer (including stripping off the original "outer" signedData layer) and MUST sign the expanded envelopedData in a new "outer" signedData layer that includes the signedAttributes (unless explicitly replaced) from the original, received "outer" signedData layer.

If the MLA does not find an "outer" signedData layer AND does not find an envelopedData layer, then the MLA MUST sign the original, received message in a new "outer" signedData layer. If the MLA does not find an "outer" signedData AND does find an envelopedData layer then it MUST expand the envelopedData layer, if present, and sign it in a new "outer" signedData layer.

4.2.1 Examples of Rule Processing

The following examples help explain the rules above:

1) A message (S1(Original Content)) (where S = SignedData) is sent to the MLA in which the signedData layer does not include an MLExpansionHistory attribute. The MLA verifies and fully processes the signedAttributes in S1. The MLA decides that there is not an original, received "outer" signedData layer since it finds the original content, but never finds an envelopedData and never finds an mlExpansionHistory attribute. The MLA calculates a new signedData layer, S2, resulting in the following message sent to the ML recipients: (S2(S1(Original Content))). The MLA includes an mlExpansionHistory attribute in S2.

2) A message (S3(S2(S1(Original Content)))) is sent to the MLA in which none of the signedData layers includes an MLExpansionHistory attribute. The MLA verifies and fully processes the signedAttributes in S3, S2 and S1. The MLA decides that there is not an original, received "outer" signedData layer since it finds the original content, but never finds an envelopedData and never finds an mlExpansionHistory attribute. The MLA calculates a new signedData layer, S4, resulting in the following message sent to the ML recipients: (S4(S3(S2(S1(Original Content))))). The MLA includes an mlExpansionHistory attribute in S4.

3) A message (E1(S1(Original Content))) (where E = envelopedData) is sent to the MLA in which S1 does not include an MLExpansionHistory attribute. The MLA decides that there is not an original, received "outer" signedData layer since it finds the E1 as the outer layer. The MLA expands the recipientInformation in E1. The MLA calculates a new signedData layer, S2, resulting in the following message sent to the ML recipients: (S2(E1(S1(Original Content)))). The MLA includes an mlExpansionHistory attribute in S2.

4) A message (S2(E1(S1(Original Content)))) is sent to the MLA in which S2 includes an MLExpansionHistory attribute. The MLA verifies the signature and fully processes the signedAttributes in S2. The MLA finds the mlExpansionHistory attribute in S2, so it decides that S2 is the "outer"

signedData. The MLA remembers the signedAttributes included in S2 for later inclusion in the new outer signedData that it applies to the message. The MLA strips off S2. The MLA then expands the recipientInformation in E1 (this invalidates the signature in S2 which is why it was stripped). The MLA calculates a new signedData layer, S3, resulting in the following message sent to the ML recipients: (S3(E1(S1(Original Content)))). The MLA includes in S3 the attributes from S2 (unless it specifically replaces an attribute value) including an updated mlExpansionHistory attribute.

5) A message (S3(S2(E1(S1(Original Content))))) is sent to the MLA in which none of the signedData layers include an MLExpansionHistory attribute. The MLA verifies the signature and fully processes the signedAttributes in S3 and S2. When the MLA encounters E1, then it decides that S2 is the "outer" signedData since S2 encapsulates E1. The MLA remembers the signedAttributes included in S2 for later inclusion in the new outer signedData that it applies to the message. The MLA strips off S3 and S2. The MLA then expands the recipientInformation in E1 (this invalidates the signatures in S3 and S2 which is why they were stripped). The MLA calculates a new signedData layer, S4, resulting in the following message sent to the ML recipients: (S4(E1(S1(Original Content)))). The MLA includes in S4 the attributes from S2 (unless it specifically replaces an attribute value) and includes a new mlExpansionHistory attribute.

6) A message (S3(S2(E1(S1(Original Content))))) is sent to the MLA in which S3 includes an MLExpansionHistory attribute. In this case, the MLA verifies the signature and fully processes the signedAttributes in S3. The MLA finds the mlExpansionHistory in S3, so it decides that S3 is the "outer" signedData. The MLA remembers the signedAttributes included in S3 for later inclusion in the new outer signedData that it applies to the message. The MLA keeps on parsing encapsulated layers because it must determine if there are any eSSSecurityLabel attributes contained within. The MLA verifies the signature and fully processes the signedAttributes in S2. When the MLA encounters E1, then it strips off S3 and S2. The MLA then expands the recipientInformation in E1 (this invalidates the signatures in S3 and S2 which is why they were stripped). The MLA calculates a new signedData layer, S4, resulting in the following message sent to the ML recipients: (S4(E1(S1(Original Content)))). The MLA includes in S4 the attributes from S3 (unless it specifically replaces an attribute value) including an updated mlExpansionHistory attribute.

4.2.3 Processing Choices

The processing used depends on the type of the outermost layer of the message. There are three cases for the type of the outermost data:

- EnvelopedData
- SignedData
- data

4.2.3.1 Processing for EnvelopedData

1. The MLA locates its own RecipientInfo and uses the information it

contains to obtain the message key.

2. The MLA removes the existing recipientInfos field and replaces it with a new recipientInfos value built from RecipientInfo structures created for each member of the mailing list. The MLA also removes the existing originatorInfo field and replaces it with a new originatorInfo value built from information describing the MLA.

3. The MLA encapsulates the expanded encrypted message in a SignedData block, adding an mlExpansionHistory attribute as described in the "Mail List Expansion" section to document the expansion.

4. The MLA signs the new message and delivers the updated message to mail list members to complete MLA processing.

4.2.3.2 Processing for SignedData

MLA processing of multi-layer messages depends on the type of data in each of the layers. Step 3 below specifies that different processing will take place depending on the type of CMS message that has been signed. That is, it needs to know the type of data at the next inner layer, which may or may not be the innermost layer.

1. The MLA verifies the signature value found in the outermost SignedData layer associated with the signed data. MLA processing of the message terminates if the message signature is invalid.

2. If the outermost SignedData layer includes a signed mlExpansionHistory attribute, the MLA checks for an expansion loop as described in the "Detecting Mail List Expansion Loops" section, then go to step 3. If the outermost SignedData layer does not include a signed mlExpansionHistory attribute, the MLA signs the whole message (including this outermost SignedData layer that doesn't have an mlExpansionHistory attribute), and delivers the updated message to mail list members to complete MLA processing.

3. Determine the type of the data that has been signed. That is, look at the type of data on the layer just below the SignedData, which may or may not be the "innermost" layer. Based on the type of data, perform either step 3.1 (EnvelopedData), step 3.2 (SignedData), or step 3.3 (all other types).

    3.1. If the signed data is EnvelopedData, the MLA performs expansion processing of the encrypted message as described previously. Note that this process invalidates the signature value in the outermost SignedData layer associated with the original encrypted message. Proceed to section 3.2 with the result of the expansion.

    3.2. If the signed data is SignedData, or is the result of expanding an EnvelopedData block in step 3.1:

3.2.1. The MLA strips the existing outermost SignedData layer after remembering the value of the mlExpansionHistory and all other signed attributes in that layer, if present.

3.2.2. If the signed data is EnvelopedData (from step 3.1), the MLA encapsulates the expanded encrypted message in a new outermost SignedData layer. On the other hand, if the signed data is SignedData (from step 3.2), the MLA encapsulates the signed dat a new outermost SignedData layer.

3.2.3. The outermost signedData layer created by the MLA replaces the original outermost signedData layer. The MLA MUST create an signed attribute list for the new outermost signedData layer wh MUST include each signed attribute present in the original outermost signedData layer, unless the MLA explicitly replaces or more particular attributes with new value. A special case is mlExpansionHistory attribute. The MLA MUST add an mlExpansionHistory signed attribute to the outer signedData lay as follows:

3.2.3.1. If the original outermost SignedData layer included an mlExpansionHistory attribute, the attribute's value is and updated with the current ML expansion information a described in the "Mail List Expansion" section.

3.2.3.2. If the original outermost SignedData layer did not include an mlExpansionHistory attribute, a new attribut is created with the current ML expansion information as described in the "Mail List Expansion" section.

3.3. If the signed data is not EnvelopedData or SignedData:

3.3.1. The MLA encapsulates the received signedData object in an outer SignedData object, and adds an mlExpansionHistory attribu to the outer SignedData object containing the current ML expans information as described in the "Mail List Expansion" section.

4. The MLA signs the new message and delivers the updated message to mail list members to complete MLA processing.

A flow chart for the above steps would be:

```
1. Has a valid signature?
       YES -> 2.
       NO  -> STOP.
2. Does outermost SignedData layer contain mlExpansionHistory?
       YES -> Check it, then -> 3.
       NO  -> Sign message (including outermost SignedData that
              doesn't have mlExpansionHistory), deliver it, STOP.
3. Check type of data just below outermost SignedData.
       EnvelopedData -> 3.1.
```

```
       SignedData -> 3.2.
       all others -> 3.3.
3.1. Expand the encrypted message, then -> 3.2.
3.2. -> 3.2.1.
3.2.1. Strip outermost SignedData layer, note value of mlExpansionHistory
       and other signed attributes, then -> 3.2.2.
3.2.2. Encapsulate in new signature, then -> 3.2.3.
3.2.3. Create new signedData layer. Was there an old mlExpansionHistory?
       YES -> copy the old mlExpansionHistory values, then -> 4.
       NO  -> create new mlExpansionHistory value, then -> 4.
3.3. Encapsulate in a SignedData layer and add an mlExpansionHistory
       attribute, then -> 4.
4. Sign message, deliver it, STOP.
```

4.2.3.3 Processing for data

1. The MLA encapsulates the message in a SignedData layer, and adds an mlExpansionHistory attribute containing the current ML expansion information as described in the "Mail List Expansion" section.

2. The MLA signs the new message and delivers the updated message to mail list members to complete MLA processing.

4.3 Mail List Agent Signed Receipt Policy Processing

If a mailing list (B) is a member of another mailing list (A), list B often needs to propagate forward the mailing list receipt policy of A. As a general rule, a mailing list should be conservative in propagating forward the mailing list receipt policy because the ultimate recipient need only process the last item in the ML expansion history. The MLA builds the expansion history to meet this requirement.

The following table describes the outcome of the union of mailing list A's policy (the rows in the table) and mailing list B's policy (the columns in the table).

	B's policy			
A's policy	none	insteadOf	inAdditionTo	missing
none	none	none	none	none
insteadOf	none	insteadOf(B)	*1	insteadOf(A)
inAdditionTo	none	insteadOf(B)	*2	inAdditionTo(A)
missing	none	insteadOf(B)	inAdditionTo(B)	missing

*1 = insteadOf(insteadOf(A) + inAdditionTo(B))
*2 = inAdditionTo(inAdditionTo(A) + inAdditionTo(B))

4.4 Mail List Expansion History Syntax

An mlExpansionHistory attribute value has ASN.1 type MLExpansionHistory. If there are more than ub-ml-expansion-history mailing lists in the sequence,

the receiving agent should provide notification of the error to a human mail list administrator. The mail list administrator is responsible for correcting the overflow condition.

```
MLExpansionHistory ::= SEQUENCE
        SIZE (1..ub-ml-expansion-history) OF MLData

id-aa-mlExpandHistory OBJECT IDENTIFIER ::= { iso(1) member-body(2)
    us(840) rsadsi(113549) pkcs(1) pkcs-9(9) smime(16) id-aa(2) 3}

ub-ml-expansion-history INTEGER ::= 64
```

MLData contains the expansion history describing each MLA that has processed a message. As an MLA distributes a message to members of an ML, the MLA records its unique identifier, date and time of expansion, and receipt policy in an MLData structure.

```
MLData ::= SEQUENCE {
  mailListIdentifier EntityIdentifier,
  expansionTime GeneralizedTime,
  mlReceiptPolicy MLReceiptPolicy OPTIONAL }

EntityIdentifier ::= CHOICE {
  issuerAndSerialNumber IssuerAndSerialNumber,
  subjectKeyIdentifier SubjectKeyIdentifier }
```

The receipt policy of the ML can withdraw the originator's request for the return of a signed receipt. However, if the originator of the message has not requested a signed receipt, the MLA cannot request a signed receipt. In the event that a ML's signed receipt policy supersedes the originator's request for signed receipts, such that the originator will not receive any signed receipts, then the MLA MAY inform the originator of that fact.

When present, the mlReceiptPolicy specifies a receipt policy that supersedes the originator's request for signed receipts. The policy can be one of three possibilities: receipts MUST NOT be returned (none); receipts should be returned to an alternate list of recipients, instead of to the originator (insteadOf); or receipts should be returned to a list of recipients in addition to the originator (inAdditionTo).

```
MLReceiptPolicy ::= CHOICE {
  none [0] NULL,
  insteadOf [1] SEQUENCE SIZE (1..MAX) OF GeneralNames,
  inAdditionTo [2] SEQUENCE SIZE (1..MAX) OF GeneralNames }
```

5. Signing Certificate Attribute

Concerns have been raised over the fact that the certificate which the signer of a CMS SignedData object desired to be bound into the verification process of the SignedData object is not cryptographically bound into the

signature itself. This section addresses this issue by creating a new attribute to be placed in the signed attributes section of a SignerInfo object.

This section also presents a description of a set of possible attacks dealing with the substitution of one certificate to verify the signature for the desired certificate. A set of ways for preventing or addressing these attacks is presented to deal with the simplest of the attacks.

Authorization information can be used as part of a signature verification process. This information can be carried in either attribute certificates and other public key certificates. The signer needs to have the ability to restrict the set of certificates used in the signature verification process, and information needs to be encoded so that is covered by the signature on the SignedData object. The methods in this section allow for the set of authorization certificates to be listed as part of the signing certificate attribute.

Explicit certificate policies can also be used as part of a signature verification process. If a signer desires to state an explicit certificate policy that should be used when validating the signature, that policy needs to be cryptographically bound into the signing process. The methods described in this section allows for a set of certificate policy statements to be listed as part of the signing certificate attribute.

5.1. Attack Descriptions

At least three different attacks can be launched against a possible signature verification process by replacing the certificate or certficates used in the signature verification process.

5.1.1 Substitution Attack Description

The first attack deals with simple substitution of one certificate for another certificate. In this attack, the issuer and serial number in the SignerInfo is modified to refer to a new certificate. This new certificate is used during the signature verification process.

The first version of this attack is a simple denial of service attack where an invalid certificate is substituted for the valid certificate. This renders the message unverifiable, as the public key in the certificate no longer matches the private key used to sign the message.

The second version is a substitution of one valid certificate for the original valid certificate where the public keys in the certificates match. This allows the signature to be validated under potentially different certificate constraints than the originator of the message intended.

5.1.2 Reissue of Certificate Description

The second attack deals with a certificate authority (CA) re-issuing the

signing certificate (or potentially one of its certificates). This attack may start becoming more frequent as Certificate Authorities reissue their own root certificates, or as certificate authorities change policies in the certificate while reissuing their root certificates. This problem also occurs when cross certificates (with potentially different restrictions) are used in the process of verifying a signature.

5.1.3 Rogue Duplicate CA Description

The third attack deals with a rogue entity setting up a certificate authority that attempts to duplicate the structure of an existing CA. Specifically, the rogue entity issues a new certificate with the same public keys as the signer used, but signed by the rogue entity's private key.

5.2 Attack Responses

This document does not attempt to solve all of the above attacks; however, a brief description of responses to each of the attacks is given in this section.

5.2.1 Substitution Attack Response

The denial of service attack cannot be prevented. After the certificate identifier has been modified in transit, no verification of the signature is possible. There is also no way to automatically identify the attack because it is indistinguishable from a message corruption.

The substitution of a valid certificate can be responded to in two different manners. The first is to make a blanket statement that the use of the same public key in two different certificates is bad practice and has to be avoided. In practice, there is no practical way to prevent users from getting new certificates with the same public keys, and it should be assumed that they will do this. Section 5.4 provides a new attribute that can be included in the SignerInfo signed attributes. This binds the correct certificate identifier into the signature. This will convert the attack from a potentially successful one to simply a denial of service attack.

5.2.2 Reissue of Certificate Response

A CA should never reissue a certificate with different attributes. Certificate Authorities that do so are following poor practices and cannot be relied on. Using the hash of the certificate as the reference to the certificate prevents this attack for end-entity certificates.

Preventing the attack based on reissuing of CA certificates would require a substantial change to the usage of the signingCertificate attribute presented in section 5.4. It would require that ESSCertIDs would need to be included in the attribute to represent the issuer certificates in the signer's certification path. This presents problems when the relying party is using a cross-certificate as part of its authentication process, and

this certificate does not appear on the list of certificates. The problems outside of a closed PKI make the addition of this information prone to error, possibly causing the rejection of valid chains.

5.2.3 Rogue Duplicate CA Response

The best method of preventing this attack is to avoid trusting the rogue CA. The use of the hash to identify certificates prevents the use of end-entity certificates from the rogue authority. However the only true way to prevent this attack is to never trust the rogue CA.

5.3 Related Signature Verification Context

Some applications require that additional information be used as part of the signature validation process. In particular, authorization information from attribute certificates and other public key certificates or policy identifiers provide additional information about the abilities and intent of the signer. The signing certificate attribute described in Section 5.4 provides the ability to bind this context information as part of the signature.

5.3.1 Authorization Information

Some applications require that authorization information found in attribute certificates and/or other public key certificates be validated. This validation requires that the application be able to find the correct certificates to perform the verification process; however there is no list of the certificates to used in a SignerInfo object. The sender has the ability to include a set of attribute certificates and public key certificates in a SignedData object. The receiver has the ability to retrieve attribute certificates and public key certificates from a directory service. There are some circumstances where the signer may wish to limit the set of certificates that may be used in verifying a signature. It is useful to be able to list the set of certificates the signer wants the recipient to use in validating the signature.

5.3.2 Policy Information

A related aspect of the certificate binding is the issue of multiple certification paths. In some instances, the semantics of a certificate in its use with a message may depend on the Certificate Authorities and policies that apply. To address this issue, the signer may also wish to bind that context under the signature. While this could be done by either signing the complete certification path or a policy ID, only a binding for the policy ID is described here.

5.4 Signing Certificate Attribute Definition

The signing certificate attribute is designed to prevent the simple substitution and re-issue attacks, and to allow for a restricted set of authorization certificates to be used in verifying a signature.

The definition of SigningCertificate is

```
SigningCertificate ::=  SEQUENCE {
    certs        SEQUENCE OF ESSCertID,
    policies     SEQUENCE OF PolicyInformation OPTIONAL
}

id-aa-signingCertificate OBJECT IDENTIFIER ::= { iso(1)
    member-body(2) us(840) rsadsi(113549) pkcs(1) pkcs9(9)
    smime(16) id-aa(2) 12 }
```

The first certificate identified in the sequence of certificate identifiers MUST be the certificate used to verify the signature. The encoding of the ESSCertID for this certificate SHOULD include the issuerSerial field. If other constraints ensure that issuerAndSerialNumber will be present in the SignerInfo, the issuerSerial field MAY be omitted. The certificate identified is used during the signature verification process. If the hash of the certificate does not match the certificate used to verify the signature, the signature MUST be considered invalid.

If more than one certificate is present in the sequence of ESSCertIDs, the certificates after the first one limit the set of authorization certificates that are used during signature validation. Authorization certificates can be either attribute certificates or normal certificates. The issuerSerial field (in the ESSCertID structure) SHOULD be present for these certificates, unless the client who is validating the signature is expected to have easy access to all the certificates requred for validation. If only the signing certificate is present in the sequence, there are no restrictions on the set of authorization certificates used in validating the signature.

The sequence of policy information terms identifies those certificate policies that the signer asserts apply to the certificate, and under which the certificate should be relied upon. This value suggests a policy value to be used in the relying party's certification path validation.

If present, the SigningCertificate attribute MUST be a signed attribute; it MUST NOT be an unsigned attribute. CMS defines SignedAttributes as a SET OF Attribute. A SignerInfo MUST NOT include multiple instances of the SigningCertificate attribute. CMS defines the ASN.1 syntax for the signed attributes to include attrValues SET OF AttributeValue. A SigningCertificate attribute MUST include only a single instance of AttributeValue. There MUST NOT be zero or multiple instances of AttributeValue present in the attrValues SET OF AttributeValue.

5.4.1 Certificate Identification

The best way to identify certificates is an often-discussed issue. [CERT] has imposed a restriction for SignedData objects that the issuer DN must be present in all signing certificates. The issuer/serial number pair is

therefore sufficient to identify the correct signing certificate. This information is already present, as part of the SignerInfo object, and duplication of this information would be unfortunate. A hash of the entire certificate serves the same function (allowing the receiver to verify that the same certificate is being used as when the message was signed), is smaller, and permits a detection of the simple substitution attacks.

Attribute certificates and additional public key certificates containing authorization information do not have an issuer/serial number pair represented anywhere in a SignerInfo object. When an attribute certificate or an additional public key certificate is not included in the SignedData object, it becomes much more difficult to get the correct set of certificates based only on a hash of the certificate. For this reason, these certificates SHOULD be identified by the IssuerSerial object.

This document defines a certificate identifier as:

```
ESSCertID ::=  SEQUENCE {
     certHash                 Hash,
     issuerSerial             IssuerSerial OPTIONAL
}

Hash ::= OCTET STRING -- SHA1 hash of entire certificate

IssuerSerial ::= SEQUENCE {
     issuer                   GeneralNames,
     serialNumber             CertificateSerialNumber
}
```

When creating an ESSCertID, the certHash is computed over the entire DER encoded certificate including the signature. The issuerSerial would normally be present unless the value can be inferred from other information.

When encoding IssuerSerial, serialNumber is the serial number that uniquely identifies the certificate. For non-attribute certificates, the issuer MUST contain only the issuer name from the certificate encoded in the directoryName choice of GeneralNames. For attribute certificates, the issuer MUST contain the issuer name field from the attribute certificate.

6. Security Considerations

All security considerations from [CMS] and [SMIME3] apply to applications that use procedures described in this document.

As stated in Section 2.3, a recipient of a receipt request must not send back a reply if it cannot validate the signature. Similarly, if there conflicting receipt requests in a message, the recipient must not send back receipts, since an attacker may have inserted the conflicting request.

Sending a signed receipt to an unvalidated sender can expose information about the recipient that it may not want to expose to unknown senders.

Senders of receipts should consider encrypting the receipts to prevent a passive attacker from gleaning information in the receipts.

Senders must not rely on recipients' processing software to correctly process security labels. That is, the sender cannot assume that adding a security label to a message will prevent recipients from viewing messages the sender doesn't want them to view. It is expected that there will be many S/MIME clients that will not understand security labels but will still display a labelled message to a recipient.

A receiving agent that processes security labels must handle the content of the messages carefully. If the agent decides not to show the message to the intended recipient after processing the security label, the agent must take care that the recipient does not accidentally see the content at a later time. For example, if an error response sent to the originator contains the content that was hidden from the recipient, and that error response bounces back to the sender due to addressing errors, the original recipient can possibly see the content since it is unlikely that the bounce message will have the proper security labels.

A man-in-the-middle attack can cause a recipient to send receipts to an attacker if that attacker has a signature that can be validated by the recipient. The attack consists of intercepting the original message and adding a mLData attribute that says that a receipt should be sent to the attacker in addition to whoever else was going to get the receipt.

Mailing lists that encrypt their content may be targets for denial-of-service attacks if they do not use the mailing list management described in Section 4. Using simple RFC822 header spoofing, it is quite easy to subscribe one encrypted mailing list to another, thereby setting up an infinite loop.

Mailing List Agents need to be aware that they can be used as oracles for the the adaptive chosen ciphertext attack described in [CMS]. MLAs should notify an administrator if a large number of undecryptable messages are received.

When verifying a signature using certificates that come with a [CMS] message, the recipient should only verify using certificates previously known to be valid, or certificates that have come from a signed SigningCertificate attribute. Otherwise, the attacks described in Section 5 can cause the receiver to possibly think a signature is valid when it is not.

A. ASN.1 Module

ExtendedSecurityServices

```
     { iso(1) member-body(2) us(840) rsadsi(113549)
       pkcs(1) pkcs-9(9) smime(16) modules(0) ess(2) }

DEFINITIONS IMPLICIT TAGS ::=
BEGIN

IMPORTS

-- Cryptographic Message Syntax (CMS)
    ContentType, IssuerAndSerialNumber, SubjectKeyIdentifier
    FROM CryptographicMessageSyntax { iso(1) member-body(2) us(840)
    rsadsi(113549) pkcs(1) pkcs-9(9) smime(16) modules(0) cms(1)}

-- PKIX Certificate and CRL Profile, Sec A.2 Implicitly Tagged Module,
--  1988 Syntax
    PolicyInformation FROM PKIX1Implicit88 {iso(1)
    identified-organization(3) dod(6) internet(1) security(5)
    mechanisms(5) pkix(7)id-mod(0) id-pkix1-implicit-88(2)}

-- X.509
    GeneralNames, CertificateSerialNumber FROM CertificateExtensions
    {joint-iso-ccitt ds(5) module(1) certificateExtensions(26) 0};


-- Extended Security Services

-- The construct "SEQUENCE SIZE (1..MAX) OF" appears in several ASN.1
-- constructs in this module. A valid ASN.1 SEQUENCE can have zero or
-- more entries. The SIZE (1..MAX) construct constrains the SEQUENCE to
-- have at least one entry. MAX indicates the upper bound is unspecified.
-- Implementations are free to choose an upper bound that suits their
-- environment.

UTF8String ::= [UNIVERSAL 12] IMPLICIT OCTET STRING
    -- The contents are formatted as described in [UTF8]

-- Section 2.7

ReceiptRequest ::= SEQUENCE {
  signedContentIdentifier ContentIdentifier,
  receiptsFrom ReceiptsFrom,
  receiptsTo SEQUENCE SIZE (1..ub-receiptsTo) OF GeneralNames }

ub-receiptsTo INTEGER ::= 16

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

ContentIdentifier ::= OCTET STRING

id-aa-contentIdentifier OBJECT IDENTIFIER ::= { iso(1) member-body(2)
```

```
    us(840) rsadsi(113549) pkcs(1) pkcs-9(9) smime(16) id-aa(2) 7}

ReceiptsFrom ::= CHOICE {
  allOrFirstTier [0] AllOrFirstTier,
  -- formerly "allOrNone [0]AllOrNone"
  receiptList [1] SEQUENCE OF GeneralNames }

AllOrFirstTier ::= INTEGER { -- Formerly AllOrNone
  allReceipts (0),
  firstTierRecipients (1) }


-- Section 2.8

Receipt ::= SEQUENCE {
  version ESSVersion,
  contentType ContentType,
  signedContentIdentifier ContentIdentifier,
  originatorSignatureValue OCTET STRING }

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

ESSVersion ::= INTEGER  { v1(1) }

-- Section 2.9

ContentHints ::= SEQUENCE {
  contentDescription UTF8String (SIZE (1..MAX)) OPTIONAL,
  contentType ContentType }

id-aa-contentHint OBJECT IDENTIFIER ::= { iso(1) member-body(2) us(840)
    rsadsi(113549) pkcs(1) pkcs-9(9) smime(16) id-aa(2) 4}

-- Section 2.10

MsgSigDigest ::= OCTET STRING

id-aa-msgSigDigest OBJECT IDENTIFIER ::= { iso(1) member-body(2)
    us(840) rsadsi(113549) pkcs(1) pkcs-9(9) smime(16) id-aa(2) 5}

-- Section 2.11

ContentReference ::= SEQUENCE {
  contentType ContentType,
  signedContentIdentifier ContentIdentifier,
  originatorSignatureValue OCTET STRING }

id-aa-contentReference   OBJECT IDENTIFIER ::= { iso(1) member-body(2)
    us(840) rsadsi(113549) pkcs(1) pkcs-9(9) smime(16) id-aa(2) 10 }
```

```
-- Section 3.2

ESSSecurityLabel ::= SET {
  security-policy-identifier SecurityPolicyIdentifier,
  security-classification SecurityClassification OPTIONAL,
  privacy-mark ESSPrivacyMark OPTIONAL,
  security-categories SecurityCategories OPTIONAL }

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

SecurityPolicyIdentifier ::= OBJECT IDENTIFIER

SecurityClassification ::= INTEGER {
  unmarked (0),
  unclassified (1),
  restricted (2),
  confidential (3),
  secret (4),
  top-secret (5) } (0..ub-integer-options)

ub-integer-options INTEGER ::= 256

ESSPrivacyMark ::= CHOICE {
    pString      PrintableString (SIZE (1..ub-privacy-mark-length)),
    utf8String   UTF8String (SIZE (1..MAX))
}

ub-privacy-mark-length INTEGER ::= 128

SecurityCategories ::= SET SIZE (1..ub-security-categories) OF
        SecurityCategory

ub-security-categories INTEGER ::= 64

SecurityCategory ::= SEQUENCE {
  type  [0] OBJECT IDENTIFIER,
  value [1] ANY DEFINED BY type -- defined by type
}

--Note: The aforementioned SecurityCategory syntax produces identical
--hex encodings as the following SecurityCategory syntax that is
--documented in the X.411 specification:
--
--SecurityCategory ::= SEQUENCE {
--     type  [0]  SECURITY-CATEGORY,
--     value [1]  ANY DEFINED BY type }
--
--SECURITY-CATEGORY MACRO ::=
--BEGIN
```

```
--TYPE NOTATION ::= type | empty
--VALUE NOTATION ::= value (VALUE OBJECT IDENTIFIER)
--END

-- Section 3.4

EquivalentLabels ::= SEQUENCE OF ESSSecurityLabel

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


-- Section 4.4

MLExpansionHistory ::= SEQUENCE
        SIZE (1..ub-ml-expansion-history) OF MLData

id-aa-mlExpandHistory OBJECT IDENTIFIER ::= { iso(1) member-body(2)
    us(840) rsadsi(113549) pkcs(1) pkcs-9(9) smime(16) id-aa(2) 3}

ub-ml-expansion-history INTEGER ::= 64

MLData ::= SEQUENCE {
  mailListIdentifier EntityIdentifier,
  expansionTime GeneralizedTime,
  mlReceiptPolicy MLReceiptPolicy OPTIONAL }

EntityIdentifier ::= CHOICE {
  issuerAndSerialNumber IssuerAndSerialNumber,
  subjectKeyIdentifier SubjectKeyIdentifier }

MLReceiptPolicy ::= CHOICE {
  none [0] NULL,
  insteadOf [1] SEQUENCE SIZE (1..MAX) OF GeneralNames,
  inAdditionTo [2] SEQUENCE SIZE (1..MAX) OF GeneralNames }


-- Section 5.4

SigningCertificate ::=  SEQUENCE {
    certs        SEQUENCE OF ESSCertID,
    policies     SEQUENCE OF PolicyInformation OPTIONAL
}

id-aa-signingCertificate OBJECT IDENTIFIER ::= { iso(1)
    member-body(2) us(840) rsadsi(113549) pkcs(1) pkcs9(9)
    smime(16) id-aa(2) 12 }

ESSCertID ::=  SEQUENCE {
     certHash                 Hash,
     issuerSerial             IssuerSerial OPTIONAL
```

```
}

Hash ::= OCTET STRING -- SHA1 hash of entire certificate

IssuerSerial ::= SEQUENCE {
     issuer                   GeneralNames,
     serialNumber             CertificateSerialNumber
}

END -- of ExtendedSecurityServices
```

B. References

[ASN1-1988] "Recommendation X.208: Specification of Abstract Syntax Notation One (ASN.1)"

[ASN1-1994] "Recommendation X.680: Specification of Abstract Syntax Notation One (ASN.1)"

[CERT] "S/MIME Version 3 Certificate Handling", Internet Draft draft-ietf-smime-cert-xx.

[CMS] "Cryptographic Message Syntax", Internet Draft draft-ietf-smime-cms-xx.

[MSG] "S/MIME Version 3 Message Specification", Internet Draft draft-ietf-smime-msg-xx.

[MUSTSHOULD] "Key Words for Use in RFCs to Indicate Requirement Levels", RFC 2119.

[MSP4] "Secure Data Network System (SDNS) Message Security Protocol (MSP) 4.0", Specification SDN.701, Revision A, 1997-02-06.

[MTSABS] "1988 International Telecommunication Union (ITU) Data Communication Networks Message Handling Systems: Message Transfer System: Abstract Service Definition and Procedures, Volume VIII, Fascicle VIII.7, Recommendation X.411"; MTSAbstractService {joint-iso-ccitt mhs-motis(6) mts(3) modules(0) mts-abstract-service(1)}

[PKCS7-1.5] "PKCS #7: Cryptographic Message Syntax", RFC 2315.

[SMIME2] "S/MIME Version 2 Message Specification", RFC 2311, and "S/MIME Version 2 Certificate Handling", RFC 2312.

[UTF8] "UTF-8, a transformation format of ISO 10646", RFC 2279.

C. Acknowledgments

The first draft of this work was prepared by David Solo. John Pawling did a huge amount of very detailed revision work during the many phases of the document.

Many other people have contributed hard work to this draft, including:
Andrew Farrell
Bancroft Scott
Bengt Ackzell
Bill Flanigan
Blake Ramsdell
Carlisle Adams
Darren Harter
David Kemp
Denis Pinkas
Francois Rousseau
Jim Schaad
Russ Housley
Scott Hollenbeck
Steve Dusse

D. Changes from draft-ietf-smime-ess-11 to draft-ietf-smime-ess-12

[[Should be removed when becoming an RFC]]

General: Corrected "an signed" to "a signed".

1, 1.3: Corrected "three services" to "four services".

2.6: Corrected spelling of "recipients".

5: Made changes to make it clear that the section covered certs that were used for authorization.

5.2.2: Corrected "attach" to "attack".

5.4: Third paragraph, replaced the third sentence to make it clearer.

C: Added Francois Rousseau.

E. Editor's Address

Paul Hoffman
Internet Mail Consortium
127 Segre Place
Santa Cruz, CA 95060
phoffman@imc.org