Network Working Group
Request for Comments: 2025
Category: Standards Track

C. Adams
Bell-Northern Research

October 1996

The Simple Public-Key GSS-API Mechanism (SPKM)

Status of this Memo

Abstract

This specification defines protocols, procedures, and conventions to be employed by peers implementing the Generic Security Service Application Program Interface (as specified in RFCs 1508 and 1509) when using the Simple Public-Key Mechanism.

Background

Although the Kerberos Version 5 GSS-API mechanism [KRB5] is becoming well-established in many environments, it is important in some applications to have a GSS-API mechanism which is based on a public-key, rather than a symmetric-key, infrastructure. The mechanism described in this document has been proposed to meet this need and to provide the following features.

1) The SPKM allows both unilateral and mutual authentication to be accomplished without the use of secure timestamps. This enables environments which do not have access to secure time to nevertheless have access to secure authentication.

2) The SPKM uses Algorithm Identifiers to specify various algorithms to be used by the communicating peers. This allows maximum flexibility for a variety of environments, for future enhancements, and for alternative algorithms.

3) The SPKM allows the option of a true, asymmetric algorithm-based, digital signature in the gss_sign() and gss_seal() operations (now called gss_getMIC() and gss_wrap() in [GSSv2]), rather than an integrity checksum based on a MAC computed with a symmetric algorithm (e.g., DES). For some environments, the availability of true digital signatures supporting non-repudiation is a necessity.

4) SPKM data formats and procedures are designed to be as similar to those of the Kerberos mechanism as is practical. This is done for ease of implementation in those environments where Kerberos has already been implemented.

For the above reasons, it is felt that the SPKM will offer flexibility and functionality, without undue complexity or overhead.

Key Management

The key management employed in SPKM is intended to be as compatible as possible with both X.509 [X.509] and PEM [RFC-1422], since these represent large communities of interest and show relative maturity in standards.

Acknowledgments

Much of the material in this document is based on the Kerberos Version 5 GSS-API mechanism [KRB5], and is intended to be as compatible with it as possible. This document also owes a great debt to Warwick Ford and Paul Van Oorschot of Bell-Northern Research for many fruitful discussions, to Kelvin Desplanque for implementation-related clarifications, to John Linn of OpenVision Technologies for helpful comments, and to Bancroft Scott of OSS for ASN.1 assistance.

1. Overview

The goal of the Generic Security Service Application Program Interface (GSS-API) is stated in the abstract of [RFC-1508] as follows:

> "This Generic Security Service Application Program Interface (GSS-API) definition provides security services to callers in a generic fashion, supportable with a range of underlying mechanisms and technologies and hence allowing source-level portability of applications to different environments. This specification defines GSS-API services and primitives at a level independent of underlying mechanism and programming language environment, and is to be complemented by other, related specifications:
>
> - documents defining specific parameter bindings for particular language environments;
>
> - documents defining token formats, protocols, and procedures to be implemented in order to realize GSS-API services atop particular security mechanisms."

The SPKM is an instance of the latter type of document and is therefore termed a "GSS-API Mechanism". This mechanism provides authentication, key establishment, data integrity, and data confidentiality in an on-line distributed application environment using a public-key infrastructure. Because it conforms to the interface defined by [RFC-1508], SPKM can be used as a drop-in replacement by any application which makes use of security services through GSS-API calls (for example, any application which already uses the Kerberos GSS-API for security). The use of a public-key infrastructure allows digital signatures supporting non-repudiation to be employed for message exchanges, and provides other benefits such as scalability to large user populations.

The tokens defined in SPKM are intended to be used by application programs according to the GSS API "operational paradigm" (see [RFC-1508] for further details):

> The operational paradigm in which GSS-API operates is as follows. A typical GSS-API caller is itself a communications protocol [or is an application program which uses a communications protocol], calling on GSS-API in order to protect its communications with authentication, integrity, and/or confidentiality security services. A GSS-API caller accepts tokens provided to it by its local GSS-API implementation [i.e., its GSS-API mechanism] and transfers the tokens to a peer on a remote system; that peer passes the received tokens to its local GSS-API implementation for processing.
>
> This document defines two separate GSS-API mechanisms, SPKM-1 and SPKM-2, whose primary difference is that SPKM-2 requires the presence of secure timestamps for the purpose of replay detection during context establishment and SPKM-1 does not. This allows greater flexibility for applications since secure timestamps cannot always be guaranteed to be available in a given environment.

2. Algorithms

A number of algorithm types are employed in SPKM. Each type, along with its purpose and a set of specific examples, is described in this section. In order to ensure at least a minimum level of interoperability among various implementations of SPKM, one of the integrity algorithms is specified as MANDATORY; all remaining examples (and any other algorithms) may optionally be supported by a given SPKM implementation (note that a GSS-conformant mechanism need not support confidentiality). Making a confidentiality algorithm mandatory may preclude exportability of the mechanism implementation; this document therefore specifies certain algorithms as RECOMMENDED (that is, interoperability will be enhanced if these algorithms are included in all SPKM implementations for which exportability is not a concern).

2.1 Integrity Algorithm (I-ALG):

Purpose:

This algorithm is used to ensure that a message has not been altered in any way after being constructed by the legitimate sender. Depending on the algorithm used, the application of this algorithm may also provide authenticity and support non-repudiation for the message.

Examples:

```
md5WithRSAEncryption OBJECT IDENTIFIER ::= {
  iso(1) member-body(2) US(840) rsadsi(113549) pkcs(1)
  pkcs-1(1) 4        -- imported from [PKCS1]
}
```

This algorithm (MANDATORY) provides data integrity and authenticity and supports non-repudiation by computing an RSA signature on the MD5 hash of that data. This is essentially equivalent to md5WithRSA {1 3 14 3 2 3}, which is defined by OIW (the Open Systems Environment Implementors' Workshop).

Note that since this is the only integrity/authenticity algorithm specified to be mandatory at this time, for interoperability reasons it is also stipulated that md5WithRSA be the algorithm used to sign all context establishment tokens which are signed rather than MACed -- see Section 3.1.1 for details. In future versions of this document, alternate or additional algorithms may be specified to be mandatory and so this stipulation on the

context establishment tokens may be removed.

```
DES-MAC OBJECT IDENTIFIER ::= {
   iso(1) identified-organization(3) oiw(14) secsig(3)
   algorithm(2) 10  -- carries length in bits of the MAC as
}                   -- an INTEGER parameter, constrained to
                    -- multiples of eight from 16 to 64
```

This algorithm (RECOMMENDED) provides integrity by computing a DES MAC (as specified by [FIPS-113]) on that data.

```
md5-DES-CBC OBJECT IDENTIFIER ::= {
   iso(1) identified-organization(3) dod(6) internet(1)
   security(5) integrity(3) md5-DES-CBC(1)
}
```

This algorithm provides data integrity by encrypting, using DES CBC, the "confounded" MD5 hash of that data (see Section 3.2.2.1 for the definition and purpose of confounding). This will typically be faster in practice than computing a DES MAC unless the input data is extremely short (e.g., a few bytes). Note that without the confounder the strength of this integrity mechanism is (at most) equal to the strength of DES under a known-plaintext attack.

```
sum64-DES-CBC OBJECT IDENTIFIER ::= {
   iso(1) identified-organization(3) dod(6) internet(1)
   security(5) integrity(3) sum64-DES-CBC(2)
}
```

This algorithm provides data integrity by encrypting, using DES CBC, the concatenation of the confounded data and the sum of all the input data blocks (the sum computed using addition modulo 2**64 - 1). Thus, in this algorithm, encryption is a requirement for the integrity to be secure.

For comments regarding the security of this integrity algorithm, see [Juen84, Davi89].

2.2 Confidentiality Algorithm (C-ALG):

Purpose:

This symmetric algorithm is used to generate the encrypted data for gss_seal() / gss_wrap().

Example:

```
DES-CBC OBJECT IDENTIFIER ::= {
   iso(1) identified-organization(3) oiw(14) secsig(3)
   algorithm(2) 7 -- carries IV (OCTET STRING) as a parameter;
}                 -- this (optional) parameter is unused in
                  -- SPKM due to the use of confounding
```

This algorithm is RECOMMENDED.

2.3 Key Establishment Algorithm (K-ALG):

Purpose:

This algorithm is used to establish a symmetric key for use by both the initiator and the target over the established context. The keys used for C-ALG and any keyed I-ALGs (for example, DES-MAC) are derived from this context key. As will be seen in Section 3.1, key establishment is done within the X.509 authentication exchange and so the resulting shared symmetric key is authenticated.

Examples:

```
RSAEncryption OBJECT IDENTIFIER ::= {
  iso(1) member-body(2) US(840) rsadsi(113549) pkcs(1)
  pkcs-1(1) 1        -- imported from [PKCS1] and [RFC-1423]
}
```

In this algorithm (MANDATORY), the context key is generated by the initiator, encrypted with the RSA public key of the target, and sent to the target. The target need not respond to the initiator for the key to be established.

```
id-rsa-key-transport OBJECT IDENTIFIER ::= {
   iso(1) identified-organization(3) oiw(14) secsig(3)
   algorithm(2) 22   -- imported from [X9.44]
}
```

Similar to RSAEncryption, but source authenticating info. is also encrypted with the target's RSA public key.

```
dhKeyAgreement OBJECT IDENTIFIER ::= {
   iso(1) member-body(2) US(840) rsadsi(113549) pkcs(1)
   pkcs-3(3) 1
}
```

In this algorithm, the context key is generated jointly by the initiator and the target using the Diffie-Hellman key establishment algorithm. The target must therefore respond to the initiator for the key to be established (so this K-ALG cannot be used with unilateral authentication in SPKM-2 (see Section 3.1)).

2.4 One-Way Function (O-ALG) for Subkey Derivation Algorithm:

Purpose:

Having established a context key using the negotiated K-ALG, both initiator and target must be able to derive a set of subkeys for the various C-ALGs and keyed I-ALGs supported over the context. Let the (ordered) list of agreed C-ALGs be numbered consecutively, so that the first algorithm (the "default") is numbered "0", the next is numbered "1", and so on. Let the numbering for the (ordered) list of agreed I-ALGs be identical. Finally, let the context key be a binary string of arbitrary length "M", subject to the following constraint: L <= M <= U (where the lower limit "L" is the bit length of the longest key needed by any agreed C-ALG or keyed I-ALG, and the upper limit "U" is the largest bit size which will fit within the K-ALG parameters).

For example, if DES and two-key-triple-DES are the negotiated confidentiality algorithms and DES-MAC is the negotiated keyed integrity algorithm (note that digital signatures do not use a context key), then the context key must be at least 112 bits long. If 512-bit RSAEncryption is the K-ALG in use then the originator can randomly generate a context key of any greater length up to 424 bits (the longest allowable RSA input specified in [PKCS-1]) -- the target can determine the length which was chosen by removing the padding bytes during the RSA decryption operation. On the other hand, if dhKeyAgreement is the K-ALG in use then the context key is the result of the Diffie-Hellman computation (with the exception of the high-order byte, which is discarded for security reasons), so that its length is that of the Diffie-Hellman modulus, p, minus 8 bits.

The derivation algorithm for a k-bit subkey is specified as follows:

```
rightmost_k_bits (OWF(context_key || x || n || s || context_key))
```

where

- "x" is the ASCII character "C" (0x43) if the subkey is for a confidentiality algorithm or the ASCII character "I" (0x49) if the subkey is for a keyed integrity algorithm;
- "n" is the number of the algorithm in the appropriate agreed list for the context (the ASCII character "0" (0x30), "1" (0x31), and so on);
- "s" is the "stage" of processing -- always the ASCII character "0" (0x30), unless "k" is greater than the output size of OWF, in which case the OWF is computed repeatedly with increasing ASCII values of "stage" (each OWF output being concatenated to the end of previous OWF outputs), until "k" bits have been generated;
- "||" is the concatenation operation; and
- "OWF" is any appropriate One-Way Function.

Examples:

```
MD5 OBJECT IDENTIFIER ::= {
  iso(1) member-body(2) US(840) rsadsi(113549)
  digestAlgorithm(2) 5
}
```

This algorithm is MANDATORY.

```
SHA OBJECT IDENTIFIER ::= {
   iso(1) identified-organization(3) oiw(14) secsig(3)
   algorithm(2) 18
}
```

It is recognized that existing hash functions may not satisfy all required properties of OWFs. This is the reason for allowing negotiation of the O-ALG OWF during the context establishment process (see Section 2.5), since in this way future improvements in OWF design can easily be accommodated. For example, in some environments a preferred OWF technique might be an encryption algorithm which encrypts the input specified above using the context_key as the encryption key.

2.5 Negotiation:

During context establishment in SPKM, the initiator offers a set of possible confidentiality algorithms and a set of possible integrity algorithms to the target (note that the term "integrity algorithms" includes digital signature algorithms). The confidentiality algorithms selected by the target become ones that may be used for C-ALG over the established context, and the integrity algorithms selected by the target become ones that may be used for I-ALG over the established context (the target "selects" algorithms by returning, in the same relative order, the subset of each offered list that it supports). Note that any C-ALG and I-ALG may be used for any message over the context and that the first confidentiality algorithm and the first integrity algorithm in the agreed sets become the default algorithms for that context.

The agreed confidentiality and integrity algorithms for a specific context define the valid values of the Quality of Protection (QOP) parameter used in the gss_getMIC() and gss_wrap() calls -- see Section 5.2 for further details. If no response is expected from the target (unilateral authentication in SPKM-2) then the algorithms offered by the initiator are the ones that may be used over the context (if this is unacceptable to the target then a delete token must be sent to the initiator so that the context is never established).

Furthermore, in the first context establishment token the initiator offers a set of possible K-ALGs, along with the key (or key half) corresponding to the first algorithm in the set (its preferred algorithm). If this K-ALG is unacceptable to the target then the target must choose one of the other K-ALGs in the set and send this choice along with the key (or key half) corresponding to this choice in its response (otherwise a delete token must be sent so that the context is never established). If necessary (that is, if the target chooses a 2-pass K-ALG such as dhKeyAgreement), the initiator will send its key half in a response to the target.

Finally, in the first context establishment token the initiator offers a set of possible O-ALGs (only a single O-ALG if no response is expected). The (single) O-ALG chosen by the target becomes the subkey derivation algorithm OWF to be used over the context.

In future versions of SPKM, other algorithms may be specified for any or all of I-ALG, C-ALG, K-ALG, and O-ALG.

3. Token Formats

This section discusses protocol-visible characteristics of the SPKM; it defines elements of protocol for interoperability and is independent of language bindings per [RFC-1509].

The SPKM GSS-API mechanism will be identified by an Object Identifier representing "SPKM-1" or "SPKM-2", having the value {spkm spkm-1(1)} or {spkm spkm-2(2)}, where spkm has the value {iso(1) identified-organization(3) dod(6) internet(1) security(5) mechanisms(5) spkm(1)}. SPKM-1 uses random numbers for replay detection during context establishment and SPKM-2 uses timestamps (note that for both mechanisms, sequence numbers are used to provide replay and out-of-sequence detection during the context, if this has been requested by the application).

Tokens transferred between GSS-API peers (for security context management and per-message protection purposes) are defined.

3.1. Context Establishment Tokens

Three classes of tokens are defined in this section: "Initiator" tokens, emitted by calls to gss_init_sec_context() and consumed by calls to gss_accept_sec_context(); "Target" tokens, emitted by calls to gss_accept_sec_context() and consumed by calls to gss_init_sec_context(); and "Error" tokens, potentially emitted by calls to gss_init_sec_context() or gss_accept_sec_context(), and potentially consumed by calls to gss_init_sec_context() or gss_accept_sec_context().

Per RFC-1508, Appendix B, the initial context establishment token will be enclosed within framing as follows:

```
InitialContextToken ::= [APPLICATION 0] IMPLICIT SEQUENCE {
        thisMech           MechType,
                -- MechType is OBJECT IDENTIFIER
                -- representing "SPKM-1" or "SPKM-2"
        innerContextToken  ANY DEFINED BY thisMech
}               -- contents mechanism-specific
```

When thisMech is SPKM-1 or SPKM-2, innerContextToken is defined as follows:

```
SPKMInnerContextToken ::= CHOICE {
   req    [0] SPKM-REQ,
   rep-ti [1] SPKM-REP-TI,
   rep-it [2] SPKM-REP-IT,
   error  [3] SPKM-ERROR,
   mic    [4] SPKM-MIC,
   wrap   [5] SPKM-WRAP,
   del    [6] SPKM-DEL
}
```

The above GSS-API framing shall be applied to all tokens emitted by the SPKM GSS-API mechanism, including SPKM-REP-TI (the response from the Target to the Initiator), SPKM-REP-IT (the response from the Initiator to the Target), SPKM-ERROR, context-deletion, and per-message tokens, not just to the initial token in a context establishment exchange. While not required by RFC-1508, this enables implementations to perform enhanced error-checking. The tag values provided in SPKMInnerContextToken ("[0]" through "[6]") specify a token-id for each token; similar information is contained in each token's tok-id field. While seemingly redundant, the tag value and tok-id actually perform different tasks: the tag ensures that InitialContextToken can be properly decoded; tok-id ensures, among other things, that data associated with the per-message tokens is cryptographically linked to the intended token type. Every innerContextToken also includes a context-id field; see Section 6 for a discussion of both token-id and context-id information and their use in an SPKM support function).

The innerContextToken field of context establishment tokens for the SPKM GSS-API mechanism will contain one of the following messages: SPKM-REQ; SPKM-REP-TI; SPKM-REP-IT; and SPKM-ERROR. Furthermore, all innerContextTokens are encoded using ASN.1 BER (constrained, in the interests of parsing simplicity, to the DER subset defined in [X.509], clause 8.7).

The SPKM context establishment tokens are defined according to [X.509] Section 10 and are compatible with [9798]. SPKM-1 (random numbers) uses Section 10.3, "Two-way Authentication", when performing unilateral authentication of the target to the initiator and uses Section 10.4, "Three-way Authentication", when mutual authentication is requested by the initiator. SPKM-2 (timestamps) uses Section 10.2, "One-way Authentication", when performing unilateral authentication of the initiator to the target and uses Section 10.3, "Two-way Authentication", when mutual authentication is requested by the initiator.

The implication of the previous paragraph is that for SPKM-2 unilateral authentication no negotiation of K-ALG can be done (the target either accepts the K-ALG and context key given by the initiator or disallows the context). For SPKM-2 mutual or SPKM-1 unilateral authentication some negotiation is possible, but the target can only choose among the one-pass K-ALGs offered by the initiator (or disallow the context). Alternatively, the initiator can request that the target generate and transmit the context key. For SPKM-1 mutual authentication the target can choose any one- or two-pass K-ALG offered by the initiator and, again, can be requested to generate and transmit the context key.

It is envisioned that typical use of SPKM-1 or SPKM-2 will involve mutual authentication. Although unilateral authentication is available for both mechanisms, its use is not generally recommended.

3.1.1. Context Establishment Tokens - Initiator (first token)

In order to accomplish context establishment, it may be necessary that both the initiator and the target have access to the other partys public-key certificate(s). In some environments the initiator may choose to acquire all certificates and send the relevant ones to the target in the first token. In other environments the initiator may request that the target send certificate data in its response token, or each side may individually obtain the certificate data it needs. In any case, however, the SPKM implementation must have the ability to obtain certificates which correspond to a supplied Name. The actual mechanism to be used to achieve this is a local implementation matter and is therefore outside the scope of this specification.

Relevant SPKM-REQ syntax is as follows (note that imports from other documents are given in Appendix A):

```
SPKM-REQ ::= SEQUENCE {
        requestToken      REQ-TOKEN,
        certif-data [0]   CertificationData OPTIONAL,
        auth-data [1]     AuthorizationData OPTIONAL
           -- see [RFC-1510] for a discussion of auth-data
}

CertificationData ::= SEQUENCE {
        certificationPath [0]          CertificationPath OPTIONAL,
        certificateRevocationList [1]  CertificateList OPTIONAL
}  -- at least one of the above shall be present
```

```
CertificationPath ::= SEQUENCE {
        userKeyId [0]         OCTET STRING OPTIONAL,
           -- identifier for user's public key
        userCertif [1]        Certificate OPTIONAL,
           -- certificate containing user's public key
        verifKeyId [2]        OCTET STRING OPTIONAL,
           -- identifier for user's public verification key
        userVerifCertif [3]   Certificate OPTIONAL,
           -- certificate containing user's public verification key
        theCACertificates [4] SEQUENCE OF CertificatePair OPTIONAL
}          -- certification path from target to source
```

Having separate verification fields allows different key pairs (possibly corresponding to different algorithms) to be used for encryption/decryption and signing/verification. Presence of [0] or [1] and absence of [2] and [3] implies that the same key pair is to be used for enc/dec and verif/signing (note that this practice is not typically recommended). Presence of [2] or [3] implies that a separate key pair is to be used for verif/signing, and so [0] or [1] must also be present. Presence of [4] implies that at least one of [0], [1], [2], and [3] must also be present.

```
REQ-TOKEN ::= SEQUENCE {
        req-contents     Req-contents,
        algId            AlgorithmIdentifier,
        req-integrity    Integrity  -- "token" is Req-contents
}

Integrity ::= BIT STRING
  -- If corresponding algId specifies a signing algorithm,
  -- "Integrity" holds the result of applying the signing procedure
  -- specified in algId to the BER-encoded octet string which results
  -- from applying the hashing procedure (also specified in algId) to
  -- the DER-encoded octets of "token".
  -- Alternatively, if corresponding algId specifies a MACing
  -- algorithm, "Integrity" holds the result of applying the MACing
  -- procedure specified in algId to the DER-encoded octets of
  -- "token" (note that for MAC, algId must be one of the integrity
  -- algorithms offered by the initiator with the appropriate subkey
  -- derived from the context key (see Section 2.4) used as the key
  -- input)
```

It is envisioned that typical use of the Integrity field for each of REQ-TOKEN, REP-TI-TOKEN, and REP-IT-TOKEN will be a true digital signature, providing unilateral or mutual authentication along with replay protection, as required. However, there are situations in which the MAC choice will be appropriate. One example is the case in which the initiator wishes to remain anonymous (so that the first, or

first and third, token(s) will be MACed and the second token will be signed). Another example is the case in which a previously authenticated, established, and cached context is being re-established at some later time (here all exchanged tokens will be MACed).

The primary advantage of the MAC choice is that it reduces processing overhead for cases in which either authentication is not required (e.g., anonymity) or authentication is established by some other means (e.g., ability to form the correct MAC on a "fresh" token in context re-establishment).

```
Req-contents ::= SEQUENCE {
        tok-id          INTEGER (256),    -- shall contain 0100(hex)
        context-id      Random-Integer,   -- see Section 6.3
        pvno            BIT STRING,       -- protocol version number
        timestamp       UTCTime OPTIONAL, -- mandatory for SPKM-2
        randSrc         Random-Integer,
        targ-name       Name,
        src-name [0]    Name OPTIONAL,
           -- must be supplied unless originator is "anonymous"
        req-data        Context-Data,
        validity [1]    Validity OPTIONAL,
           -- validity interval for key (may be used in the
           -- computation of security context lifetime)
        key-estb-set    Key-Estb-Algs,
           -- specifies set of key establishment algorithms
        key-estb-req     BIT STRING OPTIONAL,
           -- key estb. parameter corresponding to first K-ALG in set
           -- (not used if initiator is unable or unwilling to
           -- generate and securely transmit key material to target).
           -- Established key must satisfy the key length constraints
           -- specified in Section 2.4.
        key-src-bind      OCTET STRING OPTIONAL
           -- Used to bind the source name to the symmetric key.
           -- This field must be present for the case of SPKM-2
           -- unilateral authen. if the K-ALG in use does not provide
           -- such a binding (but is optional for all other cases).
           -- The octet string holds the result of applying the
           -- mandatory hashing procedure MD5 (in MANDATORY I-ALG;
           -- see Section 2.1) as follows:  MD5(src || context_key),
           -- where "src" is the DER-encoded octets of src-name,
           -- "context-key" is the symmetric key (i.e., the
           -- unprotected version of what is transmitted in
           -- key-estb-req), and "||" is the concatenation operation.
        }
```

```
-- The protocol version number (pvno) parameter is a BIT STRING which
-- uses as many bits as necessary to specify all the SPKM protocol
-- versions supported by the initiator (one bit per protocol
-- version).  The protocol specified by this document is version 0.
-- Bit 0 of pvno is therefore set if this version is supported;
-- similarly, bit 1 is set if version 1 (if defined in the future) is
-- supported, and so on.  Note that for unilateral authentication
-- using SPKM-2, no response token is expected during context
-- establishment, so no protocol negotiation can take place; in this
-- case, the initiator must set exactly one bit of pvno.  The version
-- of REQ-TOKEN must correspond to the highest bit set in pvno.
-- The "validity" parameter above is the only way within SPKM for
-- the initiator to transmit desired context lifetime to the target.
-- Since it cannot be guaranteed that the initiator and target have
-- synchronized time, the span of time specified by "validity" is to
-- be taken as definitive (rather than the actual times given in this
-- parameter).

Random-Integer ::= BIT STRING

-- Each SPKM implementation is responsible for generating a "fresh"
-- random number for the purpose of context establishment; that is,
-- one which (with high probability) has not been used previously.
-- There are no cryptographic requirements on this random number
-- (i.e., it need not be unpredictable, it simply needs to be fresh).

Context-Data ::= SEQUENCE {
        channelId       ChannelId OPTIONAL, -- channel bindings
        seq-number      INTEGER OPTIONAL,   -- sequence number
        options         Options,
        conf-alg        Conf-Algs,          -- confidentiality. algs.
        intg-alg        Intg-Algs,          -- integrity algorithm
        owf-alg         OWF-Algs            -- for subkey derivation
}

ChannelId ::= OCTET STRING

Options ::= BIT STRING {
        delegation-state (0),
        mutual-state (1),
        replay-det-state (2), -- used for replay det. during context
        sequence-state (3),   -- used for sequencing during context
        conf-avail (4),
        integ-avail (5),
        target-certif-data-required (6)
                              -- used to request targ's certif. data
}
```

```
Conf-Algs ::= CHOICE {
        algs [0]        SEQUENCE OF AlgorithmIdentifier,
        null [1]        NULL
         -- used when conf. is not available over context
} -- for C-ALG (see Section 5.2 for discussion of QOP)

Intg-Algs ::= SEQUENCE OF AlgorithmIdentifier
    -- for I-ALG (see Section 5.2 for discussion of QOP)

OWF-Algs ::= SEQUENCE OF AlgorithmIdentifier
    -- Contains exactly one algorithm in REQ-TOKEN for SPKM-2
    -- unilateral, and contains at least one algorithm otherwise.
    -- Always contains exactly one algorithm in REP-TOKEN.

Key-Estb-Algs ::= SEQUENCE OF AlgorithmIdentifier
    -- to allow negotiation of K-ALG
```

A context establishment sequence based on the SPKM will perform unilateral authentication if the mutual-req bit is not set in the application's call to gss_init_sec_context(). SPKM-2 accomplishes this using only SPKM-REQ (thereby authenticating the initiator to the target), while SPKM-1 accomplishes this using both SPKM-REQ and SPKM-REP-TI (thereby authenticating the target to the initiator).

Applications requiring authentication of both peers (initiator as well as target) must request mutual authentication, resulting in "mutual-state" being set within SPKM-REQ Options. In response to such a request, the context target will reply to the initiator with an SPKM-REP-TI token. If mechanism SPKM-2 has been chosen, this completes the (timestamp-based) mutual authentication context establishment exchange. If mechanism SPKM-1 has been chosen and SPKM-REP-TI is sent, the initiator will then reply to the target with an SPKM-REP-IT token, completing the (random-number-based) mutual authentication context establishment exchange.

Other bits in the Options field of Context-Data are explained in RFC-1508, with the exception of target-certif-data-required, which the initiator sets to TRUE to request that the target return its certification data in the SPKM-REP-TI token. For unilateral authentication in SPKM-2 (in which no SPKM-REP-TI token is constructed), this option bit is ignored by both initiator and target.

3.1.2. Context Establishment Tokens - Target

```
SPKM-REP-TI ::= SEQUENCE {
        responseToken    REP-TI-TOKEN,
        certif-data      CertificationData OPTIONAL
          -- included if target-certif-data-required option was
          -- set to TRUE in SPKM-REQ
}


REP-TI-TOKEN ::= SEQUENCE {
        rep-ti-contents Rep-ti-contents,
        algId           AlgorithmIdentifier,
        rep-ti-integ    Integrity  -- "token" is Rep-ti-contents
}

Rep-ti-contents ::= SEQUENCE {
        tok-id           INTEGER (512),   -- shall contain 0200 (hex)
        context-id       Random-Integer,  -- see Section 6.3
        pvno [0]         BIT STRING OPTIONAL, -- prot. version number
        timestamp        UTCTime OPTIONAL, -- mandatory for SPKM-2
        randTarg         Random-Integer,
        src-name [1]     Name OPTIONAL,
          -- must contain whatever value was supplied in REQ-TOKEN
        targ-name        Name,
        randSrc          Random-Integer,
        rep-data         Context-Data,
        validity [2]     Validity  OPTIONAL,
          -- validity interval for key (used if the target can only
          -- support a shorter context lifetime than was offered in
          -- REQ-TOKEN)
        key-estb-id      AlgorithmIdentifier OPTIONAL,
          -- used if target is changing key estb. algorithm (must be
          -- a member of initiators key-estb-set)
        key-estb-str      BIT STRING OPTIONAL
          -- contains (1) the response to the initiators
          -- key-estb-req (if init. used a 2-pass K-ALG), or (2) the
          -- key-estb-req corresponding to the K-ALG supplied in
          -- above key-estb-id, or (3) the key-estb-req corresponding
          -- to the first K-ALG supplied in initiator's key-estb-id,
          -- if initiator's (OPTIONAL) key-estb-req was not used
          -- (target's key-estb-str must be present in this case).
          -- Established key must satisfy the key length constraints
          -- specified in Section 2.4.
        }
```

The protocol version number (pvno) parameter is a BIT STRING which uses as many bits as necessary to specify a single SPKM protocol version offered by the initiator which is supported by the target (one bit per protocol version); that is, the target sets exactly one bit of pvno. If none of the versions offered by the initiator are supported by the target, a delete token must be returned so that the context is never established. If the initiator's pvno has only one bit set and the target happens to support this protocol version, then this version is used over the context and the pvno parameter of REP-TOKEN can be omitted. Finally, if the initiator and target do have one or more versions in common but the version of the REQ-TOKEN received is not supported by the target, a REP-TOKEN must be sent with a desired version bit set in pvno (and dummy values used for all subsequent token fields). The initiator can then respond with a new REQ-TOKEN of the proper version (essentially starting context establishment anew).

3.1.3. Context Establishment Tokens - Initiator (second token)

Relevant SPKM-REP-IT syntax is as follows:

```
SPKM-REP-IT ::= SEQUENCE {
        responseToken    REP-IT-TOKEN,
        algId            AlgorithmIdentifier,
        rep-it-integ     Integrity  -- "token" is REP-IT-TOKEN
}

REP-IT-TOKEN ::= SEQUENCE {
        tok-id           INTEGER (768), -- shall contain 0300 (hex)
        context-id       Random-Integer,
        randSrc          Random-Integer,
        randTarg         Random-Integer,
        targ-name        Name,  -- the targ-name specified in REP-TI
        src-name         Name OPTIONAL,
          -- must contain whatever value was supplied in REQ-TOKEN
        key-estb-rep     BIT STRING OPTIONAL
              -- contains the response to targets key-estb-str
              -- (if target selected a 2-pass K-ALG)
        }
```

3.1.4. Error Token

The syntax of SPKM-ERROR is as follows:

```
SPKM-ERROR ::= SEQUENCE {
        error-token      ERROR-TOKEN,
        algId            AlgorithmIdentifier,
        integrity        Integrity  -- "token" is ERROR-TOKEN
```

```
}

ERROR-TOKRN ::=   SEQUENCE {
        tok-id          INTEGER (1024), -- shall contain 0400 (hex)
        context-id      Random-Integer
        }
```

The SPKM-ERROR token is used only during the context establishment process. If an SPKM-REQ or SPKM-REP-TI token is received in error, the receiving function (either gss_init_sec_context() or gss_accept_sec_context()) will generate an SPKM-ERROR token to be sent to the peer (if the peer is still in the context establishment process) and will return GSS_S_CONTINUE_NEEDED. If, on the other hand, no context establishment response is expected from the peer (i.e., the peer has completed context establishment), the function will return the appropriate major status code (e.g., GSS_S_BAD_SIG) along with a minor status of GSS_SPKM_S_SG_CONTEXT_ESTB_ABORT and all context-relevant information will be deleted. The output token will not be an SPKM-ERROR token but will instead be an SPKM-DEL token which will be processed by the peer's gss_process_context_token().

If gss_init_sec_context() receives an error token (whether valid or invalid), it will regenerate SPKM-REQ as its output token and return a major status code of GSS_S_CONTINUE_NEEDED. (Note that if the peer's gss_accept_sec_context() receives SPKM-REQ token when it is expecting a SPKM-REP-IT token, it will ignore SPKM-REQ and return a zero-length output token with a major status of GSS_S_CONTINUE_NEEDED.)

Similarly, if gss_accept_sec_context() receives an error token (whether valid or invalid), it will regenerate SPKM-REP-TI as its output token and return a major status code of GSS_S_CONTINUE_NEEDED.

md5WithRsa is currently stipulated for the signing of context establishment tokens. Discrepancies involving modulus bitlength can be resolved through judicious use of the SPKM-ERROR token. The context initiator signs REQ-TOKEN using the strongest RSA it supports (e.g., 1024 bits). If the target is unable to verify signatures of this length, it sends SPKM-ERROR signed with the strongest RSA that it supports (e.g. 512).

At the completion of this exchange, both sides know what RSA bitlength the other supports, since the size of the signature is equal to the size of the modulus. Further exchanges can be made (using successively smaller supported bitlengths) until either an agreement is reached or context establishment is aborted because no agreement is possible.

3.2. Per-Message and Context Deletion Tokens

Three classes of tokens are defined in this section: "MIC" tokens, emitted by calls to gss_getMIC() and consumed by calls to gss_verifyMIC(); "Wrap" tokens, emitted by calls to gss_wrap() and consumed by calls to gss_unwrap(); and context deletion tokens, emitted by calls to gss_init_sec_context(), gss_accept_sec_context(), or gss_delete_sec_context() and consumed by calls to gss_process_context_token().

3.2.1. Per-message Tokens - Sign / MIC

Use of the gss_sign() / gss_getMIC() call yields a token, separate from the user data being protected, which can be used to verify the integrity of that data as received. The token and the data may be sent separately by the sending application and it is the receiving application's responsibility to associate the received data with the received token.

The SPKM-MIC token has the following format:

```
SPKM-MIC ::= SEQUENCE {
        mic-header       Mic-Header,
        int-cksum        BIT STRING
                             -- Checksum over header and data,
                             -- calculated according to algorithm
                             -- specified in int-alg field.
}

Mic-Header ::= SEQUENCE {
        tok-id           INTEGER (257),
                             -- shall contain 0101 (hex)
        context-id       Random-Integer,
        int-alg [0]      AlgorithmIdentifier OPTIONAL,
                             -- Integrity algorithm indicator (must
                             -- be one of the agreed integrity
                             -- algorithms for this context).
                             -- field not present = default id.
        snd-seq [1]      SeqNum OPTIONAL  -- sequence number field.
}

SeqNum ::= SEQUENCE {
        num      INTEGER, -- the sequence number itself
        dir-ind  BOOLEAN  -- a direction indicator
}
```

3.2.1.1. Checksum

Checksum calculation procedure (common to all algorithms -- note that for SPKM the term "checksum" includes digital signatures as well as hashes and MACs): Checksums are calculated over the data field, logically prepended by the bytes of the plaintext token header (mic-header). The result binds the data to the entire plaintext header, so as to minimize the possibility of malicious splicing.

For example, if the int-alg specifies the md5WithRSA algorithm, then the checksum is formed by computing an MD5 [RFC-1321] hash over the plaintext data (prepended by the header), and then computing an RSA signature [PKCS1] on the 16-byte MD5 result. The signature is computed using the RSA private key retrieved from the credentials structure and the result (whose length is implied by the "modulus" parameter in the private key) is stored in the int-cksum field.

If the int-alg specifies a keyed hashing algorithm (for example, DES-MAC or md5-DES-CBC), then the key to be used is the appropriate subkey derived from the context key (see Section 2.4). Again, the result (whose length is implied by int-alg) is stored in the int-cksum field.

3.2.1.2. Sequence Number

It is assumed that the underlying transport layers (of whatever protocol stack is being used by the application) will provide adequate communications reliability (that is, non-malicious loss, re-ordering, etc., of data packets will be handled correctly). Therefore, sequence numbers are used in SPKM purely for security, as opposed to reliability, reasons (that is, to avoid malicious loss, replay, or re-ordering of SPKM tokens) -- it is therefore recommended that applications request sequencing and replay detection over all contexts. Note that sequence numbers are used so that there is no requirement for secure timestamps in the message tokens. The initiator’s initial sequence number for the current context may be explicitly given in the Context-Data field of SPKM-REQ and the target’s initial sequence number may be explicitly given in the Context-Data field of SPKM-REP-TI; if either of these is not given then the default value of 00 is to be used.

Sequence number field: The sequence number field is formed from the sender’s four-byte sequence number and a Boolean direction-indicator (FALSE - sender is the context initiator, TRUE - sender is the context acceptor). After constructing a gss_sign/getMIC() or gss_seal/wrap() token, the sender’s seq. number is incremented by 1.

3.2.1.3. Sequence Number Processing

The receiver of the token will verify the sequence number field by comparing the sequence number with the expected sequence number and the direction indicator with the expected direction indicator. If the sequence number in the token is higher than the expected number, then the expected sequence number is adjusted and GSS_S_GAP_TOKEN is returned. If the token sequence number is lower than the expected number, then the expected sequence number is not adjusted and GSS_S_DUPLICATE_TOKEN, GSS_S_UNSEQ_TOKEN, or GSS_S_OLD_TOKEN is returned, whichever is appropriate. If the direction indicator is wrong, then the expected sequence number is not adjusted and GSS_S_UNSEQ_TOKEN is returned.

Since the sequence number is used as part of the input to the integrity checksum, sequence numbers need not be encrypted, and attempts to splice a checksum and sequence number from different messages will be detected. The direction indicator will detect tokens which have been maliciously reflected.

3.2.2. Per-message Tokens - Seal / Wrap

Use of the gss_seal() / gss_wrap() call yields a token which encapsulates the input user data (optionally encrypted) along with associated integrity check quantities. The token emitted by gss_seal() / gss_wrap() consists of an integrity header followed by a body portion that contains either the plaintext data (if conf-alg = NULL) or encrypted data (using the appropriate subkey specified in Section 2.4 for one of the agreed C-ALGs for this context).

The SPKM-WRAP token has the following format:

```
SPKM-WRAP ::= SEQUENCE {
        wrap-header       Wrap-Header,
        wrap-body         Wrap-Body
}

Wrap-Header ::= SEQUENCE {
        tok-id           INTEGER (513),
                             -- shall contain 0201 (hex)
        context-id       Random-Integer,
        int-alg [0]      AlgorithmIdentifier OPTIONAL,
                             -- Integrity algorithm indicator (must
                             -- be one of the agreed integrity
                             -- algorithms for this context).
                             -- field not present = default id.
```

```
            conf-alg [1]     Conf-Alg OPTIONAL,
                                 -- Confidentiality algorithm indicator
                                 -- (must be NULL or one of the agreed
                                 -- confidentiality algorithms for this
                                 -- context).
                                 -- field not present = default id.
                                 -- NULL = none (no conf. applied).
            snd-seq [2]      SeqNum OPTIONAL
                                 -- sequence number field.
    }



    Wrap-Body ::= SEQUENCE {
            int-cksum        BIT STRING,
                                 -- Checksum of header and data,
                                 -- calculated according to algorithm
                                 -- specified in int-alg field.
            data             BIT STRING
                                 -- encrypted or plaintext data.
    }

    Conf-Alg ::= CHOICE {
            algId [0]        AlgorithmIdentifier,
            null [1]         NULL
    }
```

3.2.2.1: Confounding

As in [KRB5], an 8-byte random confounder is prepended to the data to compensate for the fact that an IV of zero is used for encryption. The result is referred to as the "confounded" data field.

3.2.2.2. Checksum

Checksum calculation procedure (common to all algorithms): Checksums are calculated over the plaintext data field, logically prepended by the bytes of the plaintext token header (wrap-header). As with gss_sign() / gss_getMIC(), the result binds the data to the entire plaintext header, so as to minimize the possibility of malicious splicing.

The examples for md5WithRSA and DES-MAC are exactly as specified in 3.2.1.1.

If int-alg specifies md5-DES-CBC and conf-alg specifies anything other than DES-CBC, then the checksum is computed according to

3.2.1.1 and the result is stored in int-cksum. However, if conf-alg specifies DES-CBC then the encryption and the integrity are done as follows. An MD5 [RFC-1321] hash is computed over the plaintext data (prepended by the header). This 16-byte value is appended to the concatenation of the "confounded" data and 1-8 padding bytes (the padding is as specified in [KRB5] for DES-CBC). The result is then CBC encrypted using the DES-CBC subkey (see Section 2.4) and placed in the "data" field of Wrap-Body. The final two blocks of ciphertext (i.e., the encrypted MD5 hash) are also placed in the int-cksum field of Wrap-Body as the integrity checksum.

If int-alg specifies sum64-DES-CBC then conf-alg must specify DES-CBC (i.e., confidentiality must be requested by the calling application or SPKM will return an error). Encryption and integrity are done in a single pass using the DES-CBC subkey as follows. The sum (modulo 2**64 - 1) of all plaintext data blocks (prepended by the header) is computed. This 8-byte value is appended to the concatenation of the "confounded" data and 1-8 padding bytes (the padding is as specified in [KRB5] for DES-CBC). As above, the result is then CBC encrypted and placed in the "data" field of Wrap-Body. The final block of ciphertext (i.e., the encrypted sum) is also placed in the int-cksum field of Wrap-Body as the integrity checksum.

3.2.2.3 Sequence Number

Sequence numbers are computed and processed for gss_wrap() exactly as specified in 3.2.1.2 and 3.2.1.3.

3.2.2.4: Data Encryption

The following procedure is followed unless (a) conf-alg is NULL (no encryption), or (b) conf-alg is DES-CBC and int-alg is md5-DES-CBC (encryption as specified in 3.2.2.2), or (c) int-alg is sum64-DES-CBC (encryption as specified in 3.2.2.2):

The "confounded" data is padded and encrypted according to the algorithm specified in the conf-alg field. The data is encrypted using CBC with an IV of zero. The key used is the appropriate subkey derived from the established context key using the subkey derivation algorithm described in Section 2.4 (this ensures that the subkey used for encryption and the subkey used for a separate, keyed integrity algorithm -- for example DES-MAC, but not sum64-DES-CBC -- are different).

3.2.3. Context deletion token

The token emitted by gss_delete_sec_context() is based on the format for tokens emitted by gss_sign() / gss_getMIC().

The SPKM-DEL token has the following format:

```
SPKM-DEL ::= SEQUENCE {
        del-header       Del-Header,
        int-cksum        BIT STRING
                             -- Checksum of header, calculated
                             -- according to algorithm specified
                             -- in int-alg field.
}

Del-Header ::= SEQUENCE {
        tok-id           INTEGER (769),
                             -- shall contain 0301 (hex)
        context-id       Random-Integer,
        int-alg [0]      AlgorithmIdentifier OPTIONAL,
                             -- Integrity algorithm indicator (must
                             -- be one of the agreed integrity
                             -- algorithms for this context).
                             -- field not present = default id.
        snd-seq [1]      SeqNum OPTIONAL
                             -- sequence number field.
}
```

The field snd-seq will be calculated as for tokens emitted by gss_sign() / gss_getMIC(). The field int-cksum will be calculated as for tokens emitted by gss_sign() / gss_getMIC(), except that the user-data component of the checksum data will be a zero-length string.

If a valid delete token is received, then the SPKM implementation will delete the context and gss_process_context_token() will return a major status of GSS_S_COMPLETE and a minor status of GSS_SPKM_S_SG_CONTEXT_DELETED. If, on the other hand, the delete token is invalid, the context will not be deleted and gss_process_context_token() will return the appropriate major status (GSS_S_BAD_SIG, for example) and a minor status of GSS_SPKM_S_SG_BAD_DELETE_TOKEN_RECD. The application may wish to take some action at this point to check the context status (such as sending a sealed/wrapped test message to its peer and waiting for a sealed/wrapped response).

4. Name Types and Object Identifiers

No mandatory name forms have yet been defined for SPKM. This section is for further study.

4.1. Optional Name Forms

This section discusses name forms which may optionally be supported by implementations of the SPKM GSS-API mechanism. It is recognized that OS-specific functions outside GSS-API are likely to exist in order to perform translations among these forms, and that GSS-API implementations supporting these forms may themselves be layered atop such OS-specific functions. Inclusion of this support within GSS-API implementations is intended as a convenience to applications.

4.1.1. User Name Form

This name form shall be represented by the Object Identifier {iso(1) member-body(2) United States(840) mit(113554) infosys(1) gssapi(2) generic(1) user_name(1)}. The recommended symbolic name for this type is "GSS_SPKM_NT_USER_NAME".

This name type is used to indicate a named user on a local system. Its interpretation is OS-specific. This name form is constructed as:

username

4.1.2. Machine UID Form

This name form shall be represented by the Object Identifier {iso(1) member-body(2) United States(840) mit(113554) infosys(1) gssapi(2) generic(1) machine_uid_name(2)}. The recommended symbolic name for this type is "GSS_SPKM_NT_MACHINE_UID_NAME".

This name type is used to indicate a numeric user identifier corresponding to a user on a local system. Its interpretation is OS-specific. The gss_buffer_desc representing a name of this type should contain a locally-significant uid_t, represented in host byte order. The gss_import_name() operation resolves this uid into a username, which is then treated as the User Name Form.

4.1.3. String UID Form

This name form shall be represented by the Object Identifier {iso(1) member-body(2) United States(840) mit(113554) infosys(1) gssapi(2) generic(1) string_uid_name(3)}. The recommended symbolic name for this type is "GSS_SPKM_NT_STRING_UID_NAME".

This name type is used to indicate a string of digits representing the numeric user identifier of a user on a local system. Its interpretation is OS-specific. This name type is similar to the Machine UID Form, except that the buffer contains a string representing the uid_t.

5. Parameter Definitions

This section defines parameter values used by the SPKM GSS-API mechanism. It defines interface elements in support of portability.

5.1. Minor Status Codes

This section recommends common symbolic names for minor_status values to be returned by the SPKM GSS-API mechanism. Use of these definitions will enable independent implementors to enhance application portability across different implementations of the mechanism defined in this specification. (In all cases, implementations of gss_display_status() will enable callers to convert minor_status indicators to text representations.) Each implementation must make available, through include files or other means, a facility to translate these symbolic names into the concrete values which a particular GSS-API implementation uses to represent the minor_status values specified in this section. It is recognized that this list may grow over time, and that the need for additional minor_status codes specific to particular implementations may arise.

5.1.1. Non-SPKM-specific codes (Minor Status Code MSB, bit 31, SET)

5.1.1.1. GSS-Related codes (Minor Status Code bit 30 SET)

```
GSS_S_G_VALIDATE_FAILED
    /* "Validation error" */
GSS_S_G_BUFFER_ALLOC
    /* "Couldn't allocate gss_buffer_t data" */
GSS_S_G_BAD_MSG_CTX
    /* "Message context invalid" */
GSS_S_G_WRONG_SIZE
    /* "Buffer is the wrong size" */
GSS_S_G_BAD_USAGE
    /* "Credential usage type is unknown" */
GSS_S_G_UNAVAIL_QOP
    /* "Unavailable quality of protection specified" */
```

5.1.1.2. Implementation-Related codes (Minor Status Code bit 30 OFF)

```
GSS_S_G_MEMORY_ALLOC
    /* "Couldn't perform requested memory allocation" */
```

5.1.2. SPKM-specific-codes (Minor Status Code MSB, bit 31, OFF)

```
GSS_SPKM_S_SG_CONTEXT_ESTABLISHED
    /* "Context is already fully established" */
GSS_SPKM_S_SG_BAD_INT_ALG_TYPE
```

```
    /* "Unknown integrity algorithm type in token" */
GSS_SPKM_S_SG_BAD_CONF_ALG_TYPE
    /* "Unknown confidentiality algorithm type in token" */
GSS_SPKM_S_SG_BAD_KEY_ESTB_ALG_TYPE
    /* "Unknown key establishment algorithm type in token" */
GSS_SPKM_S_SG_CTX_INCOMPLETE
    /* "Attempt to use incomplete security context" */
GSS_SPKM_S_SG_BAD_INT_ALG_SET
    /* "No integrity algorithm in common from offered set" */
GSS_SPKM_S_SG_BAD_CONF_ALG_SET
    /* "No confidentiality algorithm in common from offered set" */
GSS_SPKM_S_SG_BAD_KEY_ESTB_ALG_SET
    /* "No key establishment algorithm in common from offered set" */
GSS_SPKM_S_SG_NO_PVNO_IN_COMMON
    /* "No protocol version number in common from offered set" */
GSS_SPKM_S_SG_INVALID_TOKEN_DATA
    /* "Data is improperly formatted:  cannot encode into token" */
GSS_SPKM_S_SG_INVALID_TOKEN_FORMAT
    /* "Received token is improperly formatted:  cannot decode" */
GSS_SPKM_S_SG_CONTEXT_DELETED
    /* "Context deleted at peer's request" */
GSS_SPKM_S_SG_BAD_DELETE_TOKEN_RECD
    /* "Invalid delete token received -- context not deleted" */
GSS_SPKM_S_SG_CONTEXT_ESTB_ABORT
   /* "Unrecoverable context establishment error. Context deleted" */
```

5.2. Quality of Protection Values

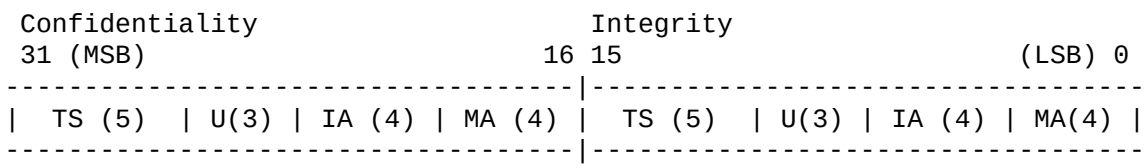
The Quality of Protection (QOP) parameter is used in the SPKM GSS-API mechanism as input to gss_sign() and gss_seal() (gss_getMIC() and gss_wrap()) to select among alternate confidentiality and integrity-checking algorithms. Once these sets of algorithms have been agreed upon by the context initiator and target, the QOP parameter simply selects from these ordered sets.

More specifically, the SPKM-REQ token sends an ordered sequence of Alg. IDs specifying integrity-checking algorithms supported by the initiator and an ordered sequence of Alg. IDs specifying confidentiality algorithms supported by the initiator. The target returns the subset of the offered integrity-checking Alg. IDs which it supports and the subset of the offered confidentiality Alg. IDs which it supports in the SPKM-REP-TI token (in the same relative orders as those given by the initiator). Thus, the initiator and target each know the algorithms which they themselves support and the algorithms which both sides support (the latter are defined to be those supported over the established context). The QOP parameter has meaning and validity with reference to this knowledge. For example, an application may request integrity algorithm number 3 as defined by

the mechanism specification. If this algorithm is supported over this context then it is used; otherwise, GSS_S_FAILURE and an appropriate minor status code are returned.

If the SPKM-REP-TI token is not used (unilateral authentication using SPKM-2), then the "agreed" sets of Alg. IDs are simply taken to be the initiator's sets (if this is unacceptable to the target then it must return an error token so that the context is never established). Note that, in the interest of interoperability, the initiator is not required to offer every algorithm it supports; rather, it may offer only the mandated/recommended SPKM algorithms since these are likely to be supported by the target.

The QOP parameter for SPKM is defined to be a 32-bit unsigned integer (an OM_uint32) with the following bit-field assignments:



where

TS is a 5-bit Type Specifier (a semantic qualifier whose value specifies the type of algorithm which may be used to protect the corresponding token -- see below for details);

U is a 3-bit Unspecified field (available for future use/expansion);

IA is a 4-bit field enumerating Implementation-specific Algorithms; and

MA is a 4-bit field enumerating Mechanism-defined Algorithms.

The interpretation of the QOP parameter is as follows (note that the same procedure is used for both the confidentiality and the integrity halves of the parameter). The MA field is examined first. If it is non-zero then the algorithm used to protect the token is the mechanism-specified algorithm corresponding to that integer value.

If MA is zero then IA is examined. If this field value is non-zero then the algorithm used to protect the token is the implementation-specified algorithm corresponding to that integer value (if this algorithm is available over the established context). Note that use of this field may hinder portability since a particular value may specify one algorithm in one implementation of the mechanism and may

not be supported or may specify a completely different algorithm in another implementation of the mechanism.

Finally, if both MA and IA are zero then TS is examined. A value of zero for TS specifies the default algorithm for the established context, which is defined to be the first algorithm on the initiator's list of offered algorithms (confidentiality or integrity, depending on which half of QOP is being examined) which is supported over the context. A non-zero value for TS corresponds to a particular algorithm qualifier and selects the first algorithm supported over the context which satisfies that qualifier.

The following TS values (i.e., algorithm qualifiers) are specified; other values may be added in the future.

For the Confidentiality TS field:

```
00001 (1) = SPKM_SYM_ALG_STRENGTH_STRONG
00010 (2) = SPKM_SYM_ALG_STRENGTH_MEDIUM
00011 (3) = SPKM_SYM_ALG_STRENGTH_WEAK
```

For the Integrity TS field:

```
00001 (1) = SPKM_INT_ALG_NON_REP_SUPPORT
00010 (2) = SPKM_INT_ALG_REPUDIABLE
```

Clearly, qualifiers such as strong, medium, and weak are debatable and likely to change with time, but for the purposes of this version of the specification we define these terms as follows. A confidentiality algorithm is "weak" if the effective key length of the cipher is 40 bits or less; it is "medium-strength" if the effective key length is strictly between 40 and 80 bits; and it is "strong" if the effective key length is 80 bits or greater. (Note that "effective key length" describes the computational effort required to break a cipher using the best-known cryptanalytic attack against that cipher.)

A five-bit TS field allows up to 31 qualifiers for each of confidentiality and integrity (since "0" is reserved for "default"). This document specifies three for confidentiality and two for integrity, leaving a lot of room for future specification. Suggestions of qualifiers such as "fast", "medium-speed", and "slow" have been made, but such terms are difficult to quantify (and in any case are platform- and processor-dependent), and so have been left out of this initial specification. The intention is that the TS terms be quantitative, environment-independent qualifiers of algorithms, as much as this is possible.

Use of the QOP structure as defined above is ultimately meant to be as follows.

- TS values are specified at the GSS-API level and are therefore portable across mechanisms. Applications which know nothing about algorithms are still able to choose "quality" of protection for their message tokens.

- MA values are specified at the mechanism level and are therefore portable across implementations of a mechanism. For example, all implementations of the Kerberos V5 GSS mechanism must support

  ```
  GSS_KRB5_INTEG_C_QOP_MD5     (value: 1)
  GSS_KRB5_INTEG_C_QOP_DES_MD5 (value: 2)
  GSS_KRB5_INTEG_C_QOP_DES_MAC (value: 3).
  ```

  (Note that these Kerberos-specified integrity QOP values do not conflict with the QOP structure defined above.)

- IA values are specified at the implementation level (in user documentation, for example) and are therefore typically non-portable. An application which is aware of its own mechanism implementation and the mechanism implementation of its peer, however, is free to use these values since they will be perfectly valid and meaningful over that context and between those peers.

The receiver of a token must pass back to its calling application a QOP parameter with all relevant fields set. For example, if triple-DES has been specified by a mechanism as algorithm 8, then a receiver of a triple-DES-protected token must pass to its application (QOP Confidentiality TS=1, IA=0, MA=8). In this way, the application is free to read whatever part of the QOP it understands (TS or IA/MA).

To aid in implementation and interoperability, the following stipulation is made. The set of integrity Alg. IDs sent by the initiator must contain at least one specifying an algorithm which computes a digital signature supporting non-repudiation, and must contain at least one specifying any other (repudiable) integrity algorithm. The subset of integrity Alg. IDs returned by the target must also contain at least one specifying an algorithm which computes a digital signature supporting non-repudiation, and at least one specifying a repudiable integrity algorithm.

The reason for this stipulation is to ensure that every SPKM implementation will provide an integrity service which supports non-repudiation and one which does not support non-repudiation. An application with no knowledge of underlying algorithms can choose one or the other by passing (QOP Integrity TS=1, IA=MA=0) or (QOP

Integrity TS=2, IA=MA=0). Although an initiator who wishes to remain anonymous will never actually use the non-repudiable digital signature, this integrity service must be available over the context so that the target can use it if desired.

Finally, in accordance with the MANDATORY and RECOMMENDED algorithms given in Section 2, the following QOP values are specified for SPKM.

For the Confidentiality MA field:

0001 (1) = DES-CBC

For the Integrity MA field:

0001 (1) = md5WithRSA
0010 (2) = DES-MAC

6. Support Functions

This section describes a mandatory support function for SPKM-conformant implementations which may, in fact, be of value in all GSS-API mechanisms. It makes use of the token-id and context-id information which is included in SPKM context-establishment, error, context-deletion, and per-message tokens. The function is defined in the following section.

6.1. SPKM_Parse_token call

Inputs:

- input_token OCTET STRING

Outputs:

- major_status INTEGER,
- minor_status INTEGER,
- mech_type OBJECT IDENTIFIER,
- token_type INTEGER,
- context_handle CONTEXT HANDLE,

Return major_status codes:

- GSS_S_COMPLETE indicates that the input_token could be parsed for all relevant fields. The resulting values are stored in mech_type, token_type and context_handle, respectively (with NULLs in any parameters which are not relevant).

- GSS_S_DEFECTIVE_TOKEN indicates that either the token-id or the context-id (if it was expected) information could not be parsed. A non-NULL return value in token_type indicates that the latter situation occurred.

- GSS_S_NO_TYPE indicates that the token-id information could be parsed, but it did not correspond to any valid token_type.

  (Note that this major status code has not been defined for GSS in RFC-1508. Until such a definition is made (if ever), SPKM implementations should instead return GSS_S_DEFECTIVE_TOKEN with both token_type and context_handle set to NULL. This essentially implies that unrecognized token-id information is considered to be equivalent to token-id information which could not be parsed.)

- GSS_S_NO_CONTEXT indicates that the context-id could be parsed, but it did not correspond to any valid context_handle.

- GSS_S_FAILURE indicates that the mechanism type could not be parsed (for example, the token may be corrupted).

SPKM_Parse_token() is used to return to an application the mechanism type, token type, and context handle which correspond to a given input token. Since GSS-API tokens are meant to be opaque to the calling application, this function allows the application to determine information about the token without having to violate the opaqueness intention of GSS. Of primary importance is the token type, which the application can then use to decide which GSS function to call in order to have the token processed.

If all tokens are framed as suggested in RFC-1508, Appendix B (specified both in the Kerberos V5 GSS mechanism [KRB5] and in this document), then any mechanism implementation should be able to return at least the mech_type parameter (the other parameters being NULL) for any uncorrupted input token. If the mechanism implementation whose SPKM_Parse_token() function is being called does recognize the token, it can return token_type so that the application can subsequently call the correct GSS function. Finally, if the mechanism provides a context-id field in its tokens (as SPKM does), then an implementation can map the context-id to a context_handle and return this to the application. This is necessary for the situation

where an application has multiple contexts open simultaneously, all using the same mechanism. When an incoming token arrives, the application can use this function to determine not only which GSS function to call, but also which context_handle to use for the call. Note that this function does no cryptographic processing to determine the validity of tokens; it simply attempts to parse the mech_type, token_type, and context-id fields of any token it is given. Thus, it is conceivable, for example, that an arbitrary buffer of data might start with random values which look like a valid mech_type and that SPKM_Parse_token() would return incorrect information if given this buffer. While conceivable, however, such a situation is unlikely.

The SPKM_Parse_token() function is mandatory for SPKM-conformant implementations, but it is optional for applications. That is, if an application has only one context open and can guess which GSS function to call (or is willing to put up with some error codes), then it need never call SPKM_Parse_token(). Furthermore, if this function ever migrates up to the GSS-API level, then SPKM_Parse_token() will be deprecated at that time in favour of GSS_Parse_token(), or whatever the new name and function specification might be. Note finally that no minor status return codes have been defined for this function at this time.

6.2. The token_type Output Parameter

The following token types are defined:

```
GSS_INIT_TOKEN   = 1
GSS_ACCEPT_TOKEN = 2
GSS_ERROR_TOKEN  = 3
GSS_SIGN_TOKEN   = GSS_GETMIC_TOKEN = 4
GSS_SEAL_TOKEN   = GSS_WRAP_TOKEN   = 5
GSS_DELETE_TOKEN = 6
```

All SPKM mechanisms shall be able to perform the mapping from the token-id information which is included in every token (through the tag values in SPKMInnerContextToken or through the tok-id field) to one of the above token types. Applications should be able to decide, on the basis of token_type, which GSS function to call (for example, if the token is a GSS_INIT_TOKEN then the application will call gss_accept_sec_context(), and if the token is a GSS_WRAP_TOKEN then the application will call gss_unwrap()).

6.3. The context_handle Output Parameter

The SPKM mechanism implementation is responsible for maintaining a mapping between the context-id value which is included in every token and a context_handle, thus associating an individual token with its

proper context. Clearly the value of context_handle may be locally determined and may, in fact, be associated with memory containing sensitive data on the local system, and so having the context-id actually be set equal to a computed context_handle will not work in general. Conversely, having the context_handle actually be set equal to a computed context-id will not work in general either, because context_handle must be returned to the application by the first call to gss_init_sec_context() or gss_accept_sec_context(), whereas uniqueness of the context-id (over all contexts at both ends) may require that both initiator and target be involved in the computation. Consequently, context_handle and context-id must be computed separately and the mechanism implementation must be able to map from one to the other by the completion of context establishment at the latest.

Computation of context-id during context establishment is accomplished as follows. Each SPKM implementation is responsible for generating a "fresh" random number; that is, one which (with high probability) has not been used previously. Note that there are no cryptographic requirements on this random number (i.e., it need not be unpredictable, it simply needs to be fresh). The initiator passes its random number to the target in the context-id field of the SPKM-REQ token. If no further context establishment tokens are expected (as for unilateral authentication in SPKM-2), then this value is taken to be the context-id (if this is unacceptable to the target then an error token must be generated). Otherwise, the target generates its random number and concatenates it to the end of the initiator's random number. This concatenated value is then taken to be the context-id and is used in SPKM-REP-TI and in all subsequent tokens over that context.

Having both peers contribute to the context-id assures each peer of freshness and therefore precludes replay attacks between contexts (where a token from an old context between two peers is maliciously injected into a new context between the same or different peers). Such assurance is not available to the target in the case of unilateral authentication using SPKM-2, simply because it has not contributed to the freshness of the computed context-id (instead, it must trust the freshness of the initiator's random number, or reject the context). The key-src-bind field in SPKM-REQ is required to be present for the case of SPKM-2 unilateral authentication precisely to assist the target in trusting the freshness of this token (and its proposed context key).

7. Security Considerations

Security issues are discussed throughout this memo.

8. References

[Davi89]: D. W. Davies and W. L. Price, "Security for Computer Networks", Second Edition, John Wiley and Sons, New York, 1989.

[FIPS-113]: National Bureau of Standards, Federal Information Processing Standard 113, "Computer Data Authentication", May 1985.

[GSSv2]: Linn, J., "Generic Security Service Application Program Interface Version 2", Work in Progress.

[Juen84]: R. R. Jueneman, C. H. Meyer and S. M. Matyas, Message Authentication with Manipulation Detection Codes, in Proceedings of the 1983 IEEE Symposium on Security and Privacy, IEEE Computer Society Press, 1984, pp.33-54.

[KRB5]: Linn, J., "The Kerberos Version 5 GSS-API Mechanism", RFC 1964, June 1996.

[PKCS1]: RSA Encryption Standard, Version 1.5, RSA Data Security, Inc., Nov. 1993.

[PKCS3]: Diffie-Hellman Key-Agreement Standard, Version 1.4, RSA Data Security, Inc., Nov. 1993.

[RFC-1321]: Rivest, R., "The MD5 Message-Digest Algorithm", RFC 1321.

[RFC-1422]: Kent, S., "Privacy Enhancement for Internet Electronic Mail: Part II: Certificate-Based Key Management", RFC 1422.

[RFC-1423]: Balenson, D., "Privacy Enhancement for Internet Elecronic Mail: Part III: Algorithms, Modes, and Identifiers", RFC 1423.

[RFC-1508]: Linn, J., "Generic Security Service Application Program Interface", RFC 1508.

[RFC-1509]: Wray, J., "Generic Security Service Application Program Interface: C-bindings", RFC 1509.

[RFC-1510]: Kohl J., and C. Neuman, "The Kerberos Network Authentication Service (V5)", RFC 1510.

[9798]: ISO/IEC 9798-3, "Information technology - Security Techniques - Entity authentication mechanisms - Part 3: Entitiy authentication using a public key algorithm", ISO/IEC, 1993.

[X.501]: ISO/IEC 9594-2, "Information Technology - Open Systems Interconnection - The Directory: Models", CCITT/ITU Recommendation X.501, 1993.

[X.509]: ISO/IEC 9594-8, "Information Technology - Open Systems Interconnection - The Directory: Authentication Framework", CCITT/ITU Recommendation X.509, 1993.

[X9.44]: ANSI, "Public Key Cryptography Using Reversible Algorithms for the Financial Services Industry: Transport of Symmetric Algorithm Keys Using RSA", X9.44-1993.

9. Author's Address

Carlisle Adams
Bell-Northern Research
P.O.Box 3511, Station C
Ottawa, Ontario, CANADA K1Y 4H7

Phone: +1 613.763.9008
EMail: cadams@bnr.ca

Appendix A: ASN.1 Module Definition

```
SpkmGssTokens {iso(1) identified-organization(3) dod(6) internet(1)
               security(5) mechanisms(5) spkm(1) spkmGssTokens(10)}


DEFINITIONS IMPLICIT TAGS ::=
BEGIN


-- EXPORTS ALL --


IMPORTS

   Name
      FROM InformationFramework {joint-iso-ccitt(2) ds(5) module(1)
                                 informationFramework(1) 2}

   Certificate, CertificateList, CertificatePair, AlgorithmIdentifier,
   Validity
      FROM AuthenticationFramework {joint-iso-ccitt(2) ds(5) module(1)
                                    authenticationFramework(7) 2}  ;



-- types --

   SPKM-REQ ::= SEQUENCE {
           requestToken      REQ-TOKEN,
           certif-data [0]   CertificationData OPTIONAL,
           auth-data [1]     AuthorizationData OPTIONAL
   }


   CertificationData ::= SEQUENCE {
           certificationPath [0]          CertificationPath OPTIONAL,
           certificateRevocationList [1]  CertificateList OPTIONAL
   } -- at least one of the above shall be present


   CertificationPath ::= SEQUENCE {
           userKeyId [0]         OCTET STRING OPTIONAL,
           userCertif [1]        Certificate OPTIONAL,
           verifKeyId [2]        OCTET STRING OPTIONAL,
           userVerifCertif [3]   Certificate OPTIONAL,
           theCACertificates [4] SEQUENCE OF CertificatePair OPTIONAL
   } -- Presence of [2] or [3] implies that [0] or [1] must also be
```

```
     -- present.  Presence of [4] implies that at least one of [0], [1],
     -- [2], and [3] must also be present.

   REQ-TOKEN ::= SEQUENCE {
           req-contents     Req-contents,
           algId            AlgorithmIdentifier,
           req-integrity    Integrity  -- "token" is Req-contents
   }

  Integrity ::= BIT STRING
     -- If corresponding algId specifies a signing algorithm,
     -- "Integrity" holds the result of applying the signing procedure
     -- specified in algId to the BER-encoded octet string which results
     -- from applying the hashing procedure (also specified in algId) to
     -- the DER-encoded octets of "token".
     -- Alternatively, if corresponding algId specifies a MACing
     -- algorithm, "Integrity" holds the result of applying the MACing
     -- procedure specified in algId to the DER-encoded octets of
     -- "token"

   Req-contents ::= SEQUENCE {
           tok-id           INTEGER (256),  -- shall contain 0100 (hex)
           context-id       Random-Integer,
           pvno             BIT STRING,
           timestamp        UTCTime OPTIONAL, -- mandatory for SPKM-2
           randSrc          Random-Integer,
           targ-name        Name,
           src-name [0]     Name OPTIONAL,
           req-data         Context-Data,
           validity [1]     Validity OPTIONAL,
           key-estb-set     Key-Estb-Algs,
           key-estb-req     BIT STRING OPTIONAL,
           key-src-bind     OCTET STRING OPTIONAL
              -- This field must be present for the case of SPKM-2
              -- unilateral authen. if the K-ALG in use does not provide
              -- such a binding (but is optional for all other cases).
              -- The octet string holds the result of applying the
              -- mandatory hashing procedure (in MANDATORY I-ALG;
              -- see Section 2.1) as follows:  MD5(src || context_key),
              -- where "src" is the DER-encoded octets of src-name,
              -- "context-key" is the symmetric key (i.e., the
              -- unprotected version of what is transmitted in
              -- key-estb-req), and "||" is the concatenation operation.
   }

   Random-Integer ::= BIT STRING
```

```
   Context-Data ::= SEQUENCE {
           channelId       ChannelId OPTIONAL,
           seq-number      INTEGER OPTIONAL,
           options         Options,
           conf-alg        Conf-Algs,
           intg-alg        Intg-Algs,
           owf-alg         OWF-Algs
   }

   ChannelId ::= OCTET STRING

   Options ::= BIT STRING {
           delegation-state (0),
           mutual-state (1),
           replay-det-state (2),
           sequence-state (3),
           conf-avail (4),
           integ-avail (5),
           target-certif-data-required (6)
   }

   Conf-Algs ::= CHOICE {
           algs [0]         SEQUENCE OF AlgorithmIdentifier,
           null [1]         NULL
   }

   Intg-Algs ::= SEQUENCE OF AlgorithmIdentifier

   OWF-Algs ::= SEQUENCE OF AlgorithmIdentifier

   Key-Estb-Algs ::= SEQUENCE OF AlgorithmIdentifier


   SPKM-REP-TI ::= SEQUENCE {
           responseToken    REP-TI-TOKEN,
           certif-data      CertificationData OPTIONAL
             -- present if target-certif-data-required option was
   }         -- set to TRUE in SPKM-REQ

   REP-TI-TOKEN ::= SEQUENCE {
           rep-ti-contents  Rep-ti-contents,
           algId            AlgorithmIdentifier,
           rep-ti-integ     Integrity  -- "token" is Rep-ti-contents
   }

   Rep-ti-contents ::= SEQUENCE {
           tok-id           INTEGER (512),   -- shall contain 0200 (hex)
           context-id       Random-Integer,
```

```
        pvno [0]         BIT STRING OPTIONAL,
        timestamp        UTCTime OPTIONAL, -- mandatory for SPKM-2
        randTarg         Random-Integer,
        src-name [1]     Name OPTIONAL,
        targ-name        Name,
        randSrc          Random-Integer,
        rep-data         Context-Data,
        validity [2]     Validity  OPTIONAL,
        key-estb-id      AlgorithmIdentifier OPTIONAL,
        key-estb-str     BIT STRING OPTIONAL
}


SPKM-REP-IT ::= SEQUENCE {
        responseToken    REP-IT-TOKEN,
        algId            AlgorithmIdentifier,
        rep-it-integ     Integrity  -- "token" is REP-IT-TOKEN
}

REP-IT-TOKEN ::= SEQUENCE {
        tok-id           INTEGER (768),  -- shall contain 0300 (hex)
        context-id       Random-Integer,
        randSrc          Random-Integer,
        randTarg         Random-Integer,
        targ-name        Name,
        src-name         Name OPTIONAL,
        key-estb-rep     BIT STRING OPTIONAL
}

SPKM-ERROR ::= SEQUENCE {
        errorToken       ERROR-TOKEN,
        algId            AlgorithmIdentifier,
        integrity        Integrity  -- "token" is ERROR-TOKEN
}

ERROR-TOKEN ::=   SEQUENCE {
        tok-id           INTEGER (1024), -- shall contain 0400 (hex)
        context-id       Random-Integer
}

SPKM-MIC ::= SEQUENCE {
        mic-header       Mic-Header,
        int-cksum        BIT STRING
}

Mic-Header ::= SEQUENCE {
        tok-id           INTEGER (257), -- shall contain 0101 (hex)
        context-id       Random-Integer,
```

```
        int-alg [0]      AlgorithmIdentifier OPTIONAL,
        snd-seq [1]      SeqNum OPTIONAL
   }

   SeqNum ::= SEQUENCE {
        num              INTEGER,
        dir-ind          BOOLEAN
   }

   SPKM-WRAP ::= SEQUENCE {
        wrap-header       Wrap-Header,
        wrap-body         Wrap-Body
   }

   Wrap-Header ::= SEQUENCE {
        tok-id           INTEGER (513), -- shall contain 0201 (hex)
        context-id       Random-Integer,
        int-alg [0]      AlgorithmIdentifier OPTIONAL,
        conf-alg [1]     Conf-Alg OPTIONAL,
        snd-seq [2]      SeqNum OPTIONAL
   }

   Wrap-Body ::= SEQUENCE {
        int-cksum        BIT STRING,
        data             BIT STRING
   }

   Conf-Alg ::= CHOICE {
        algId [0]        AlgorithmIdentifier,
        null [1]         NULL
   }


   SPKM-DEL ::= SEQUENCE {
        del-header       Del-Header,
        int-cksum        BIT STRING
   }

   Del-Header ::= SEQUENCE {
        tok-id           INTEGER (769), -- shall contain 0301 (hex)
        context-id       Random-Integer,
        int-alg [0]      AlgorithmIdentifier OPTIONAL,
        snd-seq [1]      SeqNum OPTIONAL
   }


-- other types --
```

```
   -- from [RFC-1508] --

   MechType ::= OBJECT IDENTIFIER

   InitialContextToken ::= [APPLICATION 0] IMPLICIT SEQUENCE {
      thisMech              MechType,
      innerContextToken     SPKMInnerContextToken
   }     -- when thisMech is SPKM-1 or SPKM-2

   SPKMInnerContextToken ::= CHOICE {
      req    [0] SPKM-REQ,
      rep-ti [1] SPKM-REP-TI,
      rep-it [2] SPKM-REP-IT,
      error  [3] SPKM-ERROR,
      mic    [4] SPKM-MIC,
      wrap   [5] SPKM-WRAP,
      del    [6] SPKM-DEL
   }


   -- from [RFC-1510] --

   AuthorizationData ::= SEQUENCE OF SEQUENCE {
     ad-type  INTEGER,
     ad-data  OCTET STRING
   }


-- object identifier assignments --

   md5-DES-CBC OBJECT IDENTIFIER ::=
      {iso(1) identified-organization(3) dod(6) internet(1) security(5)
       integrity(3) md5-DES-CBC(1)}

   sum64-DES-CBC OBJECT IDENTIFIER ::=
      {iso(1) identified-organization(3) dod(6) internet(1) security(5)
       integrity(3) sum64-DES-CBC(2)}

   spkm-1 OBJECT IDENTIFIER ::=
      {iso(1) identified-organization(3) dod(6) internet(1) security(5)
       mechanisms(5) spkm(1) spkm-1(1)}

   spkm-2 OBJECT IDENTIFIER ::=
      {iso(1) identified-organization(3) dod(6) internet(1) security(5)
       mechanisms(5) spkm(1) spkm-2(2)}


END
```

Appendix B: Imported Types

This appendix contains, for completeness, the relevant ASN.1 types imported from InformationFramework (1993), AuthenticationFramework (1993), and [PKCS3].

```
AttributeType ::= OBJECT IDENTIFIER
AttributeValue ::= ANY
AttributeValueAssertion ::= SEQUENCE {AttributeType,AttributeValue}
RelativeDistinguishedName ::= SET OF AttributeValueAssertion
   -- note that the 1993 InformationFramework module uses
   -- different syntax for the above constructs
RDNSequence ::= SEQUENCE OF RelativeDistinguishedName
DistinguishedName ::= RDNSequence
Name ::= CHOICE {  -- only one for now
        rdnSequence       RDNSequence
}

Certificate ::= SEQUENCE {
        certContents      CertContents,
        algID             AlgorithmIdentifier,
        sig               BIT STRING
}  -- sig holds the result of applying the signing procedure
   -- specified in algId to the BER-encoded octet string which
   -- results from applying the hashing procedure (also specified in
   -- algId) to the DER-encoded octets of CertContents

CertContents ::= SEQUENCE {
        version [0]        Version DEFAULT v1,
        serialNumber       CertificateSerialNumber,
        signature          AlgorithmIdentifier,
        issuer             Name,
        validity           Validity,
        subject            Name,
        subjectPublicKeyInfo     SubjectPublicKeyInfo,
        issuerUID [1]      IMPLICIT UID OPTIONAL,  -- used in v2 only
        subjectUID [2]     IMPLICIT UID OPTIONAL   -- used in v2 only
}

Version ::= INTEGER {v1(0), v2(1)}
CertificateSerialNumber ::= INTEGER
UID ::= BIT STRING

Validity ::= SEQUENCE {
        notBefore         UTCTime,
        notAfter          UTCTime
}
```

```
   SubjectPublicKeyInfo ::= SEQUENCE {
           algorithm         AlgorithmIdentifier,
           subjectPublicKey  BIT STRING
   }

   CertificatePair ::= SEQUENCE {
           forward [0]      Certificate OPTIONAL,
           reverse [1]      Certificate OPTIONAL
   }         -- at least one of the pair shall be present

   CertificateList ::= SEQUENCE {
           certListContents        CertListContents,
           algId                   AlgorithmIdentifier,
           sig                     BIT STRING
   }  -- sig holds the result of applying the signing procedure
      -- specified in algId to the BER-encoded octet string which
      -- results from applying the hashing procedure (also specified in
      -- algId) to the DER-encoded octets of CertListContents

   CertListContents ::= SEQUENCE {
           signature               AlgorithmIdentifier,
           issuer                  Name,
           thisUpdate              UTCTime,
           nextUpdate              UTCTime OPTIONAL,
           revokedCertificates     SEQUENCE OF SEQUENCE {
                userCertificate       CertificateSerialNumber,
                revocationDate        UTCTime           } OPTIONAL
   }

   AlgorithmIdentifier ::= SEQUENCE {
           algorithm         OBJECT IDENTIFIER,
           parameter         ANY DEFINED BY algorithm OPTIONAL
   }  -- note that the 1993 AuthenticationFramework module uses
      -- different syntax for this construct



   --from [PKCS3] (the parameter to be used with dhKeyAgreement) --

   DHParameter ::= SEQUENCE {
     prime              INTEGER,  -- p
     base               INTEGER,  -- g
     privateValueLength INTEGER OPTIONAL
   }
```