Network Working Group
Internet-Draft
Expires: April 16, 2005

W. Hardaker
Sparta
D. Perkins
SNMPInfo

October 16, 2004

A Session-Based Security Model (SBSM) for version 3 of the Simple Network Management Protocol (SNMPv3) draft-hardaker-snmp-session-sm-03.txt

Status of this Memo

Copyright Notice

Abstract

This document describes a Session Based Security Model (SBSM) for use within version 3 of the Simple Network Management Protocol (SNMPv3). The security model is designed to establish a "session" between two interacting SNMPv3 entities, over which SNMP operations can be sent securely. It provides a number of security properties not previously available in defined SNMPv3 security models, such as public key based identity authentication, limited life-time keying, and the ability to

make use of previously implemented and deployed security infrastructures for purposes of identification and authentication.

Table of Contents

1. Introduction

This document describes a Session Based Security Model (SBSM) for use within version 3 of the Simple Network Management Protocol (SNMPv3). The security model is designed to establish a "session" between two interacting SNMPv3 entities, over which SNMP operations can be sent securely. It provides a number of security properties not previously available in defined SNMPv3 security models, such as public key based identity authentication, limited life-time keying, and the ability to make use of previously implemented and deployed security infrastructures for purposes of identification and authentication. It also supports creation of a authenticated and possibly encrypted session when the identity of the initiator of the session is anonymous or unknown. These properties and the other goals of the (SBSM) are documented in Section Section 3.

The details of the technology and concepts on which the SBSM is built comes from previously described and operationally proven works, such as the SIGMA security protocol, and the IKEv2 key exchange specification. Although it is not required that the reader understands the concepts in these other documents, it certainly wouldn't hurt. And to ease the review of this document, note that no new cryptographic algorithms or security protocols are defined in this document beyond those defined in previous or other SNMPv3 standards documents.

1.1 SNMPv3 background information

Although all of the SNMPv3 protocol specifications are described in RFCs 3410-3415 those who are new to SNMPv3 may find it useful to read a companion document instead, which is a concise and easy to understand summary of the SNMPv3 protocol specifications [refs.v3overview]. It is designed to be especially helpful for people which wish to read this document but are not well versed in how the security aspects of the SNMPv3 protocol specification are designed.

1.2 Status of this document

This document is a work in progress.

2. Document conventions

2.1 SBSM Definitions and Terminology

The following terms are used through this document:

session: A potentially long lived interaction between two SNMPv3 entities.

initiator: The SNMPv3 entity that starts a session by sending the first SBSM initiation message. An initiator can be either a manager and/or a managed device and once the session is established all types of transactions may flow through it regardless of origin (that is, the responder can be a manager or managed device). For example, if a manager becomes an initiator and opens a session, it can send SNMP GET operations through it and the managed device can send SNMP INFORM operations back through the same session.

responder: The SNMPv3 entity that listens for connections and responds to initiation requests from the initiator.

identity authentication: Verifying that the SNMPv3 entity is who it claims to be. This can be a process running on a computer system, or a network operator acting through an application.

message authentication: Verifying that a SNMPv3 message has not been modified, reordered, or replayed and that it belongs to the sessions under which it was received.

message encryption: Protecting portions of a message from disclosure during transmission through the use of cryptographic algorithms.

The | operator used in multiple equations in this document refers to the string concatenation operator, *not* the xor operator. In the ASN.1 and MIB portions of this document, it refers to options.

2.2 Protocol documentation conventions

Portions of this document contain simple assignment operations in order to simplify understanding of what happens at particular points during processing of the protocol operations. They are expressed in a pseudo-code style text block, such as:

```
outgoingMessage.init-identifier = store.local-identifier
```

In the simplest and most common case, this is simply a copy operation which dictates what should be copied and to where it should be copied (in this case the local-identifies stored in the "store" is copied to the init-identifier field of the outgoingMessage construct). Generally the usage of these code blocks should be simple to understand and shorter than what a text sentence could quickly convey.

One particular information source which might take a bit more explanation: "generated", EG:

outgoingMessage.init-DH-value = generated.diffie-helman-half

In this case, the value to be stored in the outgoingMessage is generated from a diffie-helman calculation, which is frequently described elsewhere in text.

Finally, it should be important to note that both these equations and the surrounding text must be read and understood in order to get the protocol correct. IE, a successful implementation must take everything into account: both the text wording and the equations. Order of execution of both the text and equations are critical for preserving some of the security properties of the SBSM protocol.

3. Goals and Objectives

The brief list of goals and objectives met by this protocol include:

- Security transactions that make use of previously deployed and widely used mechanisms for establishing identity authentication. This includes public/private key technologies (including PKI infrastructures), and other common and currently deployed authenticating mechanisms such as Radius and TACACS+.

- Session-based keying properties such as dynamically created keys, limited lifetime keys, separate negotiated keys for message authentication and message encryption, and perfect forward secrecy (PFS) support of those keys.

- Retransmissions and replays of SNMP protocol operations do NOT result in reprocessing of the message within the protocol. EG, a managed device which receives and processes a SET request will not reprocess that same SET request in the future, even if a manager retransmits its original request due to packets being dropped within the network. This is done to nullify the damage possible via retransmitted SNMP messages which would have previously been reprocessed within the security time window of other protocols, such as the USM.

- SBSM sessions will work over any lower layer transports, which include both UDP and TCP, for example. As well, the session parameters are not bound to the lower layer transport.

- SNMP message exchange that is authenticated and even private when the session initiator or responder is anonymous.

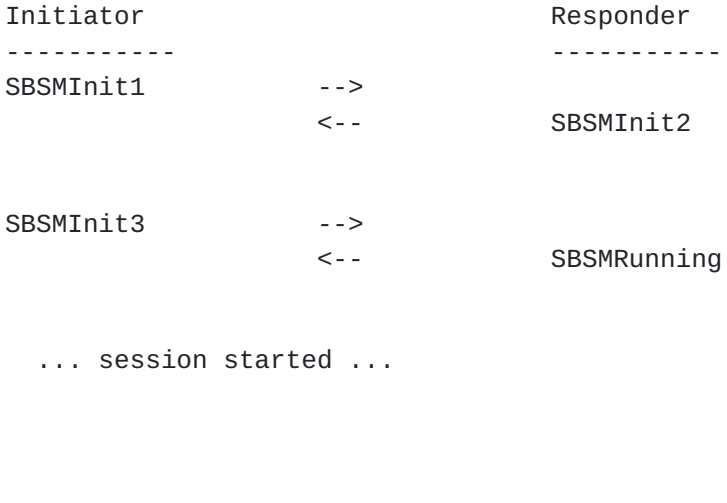
- Negotiated compression to reduce the overhead of BER encoding rules before encryption is processed.

4. Protocol Overview

The SBSM protocol is designed to meet the goals and objectives listed in Section Section 3. The SBSM session gets established through some initial hand-shake transactions. These transactions exist entirely within the security parameter field of the SNMPv3 message and the application is not involved. Generally an application sending something through a SBSM security model will trigger the creation of a session within the initiator, and the responder will trigger session creation when it receives the first message from a hand-shake.

Establishing a SBSM security session between an initiator and a responder takes some negotiation between the two pairs. The complexity of this exchange has been kept to a bare minimum wherever possible. It would be easy to conclude that more parameters should be included since they would be convenient (such as timeout values, session length values, etc) but they offer little benefit for their increased complexity and thus have been left out.

The initial exchange for creating a session looks roughly like the following series of security-parameter exchanges, assuming no errors occur during the establishment:



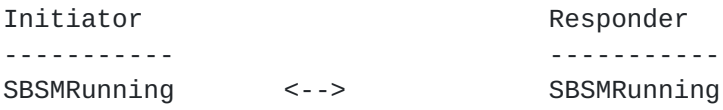
Note: The above flow diagram is the most simple case. When

challenge-response identification protocols, such as EAP, are used to authenticate identities, then more messages need to be sent than those above. E.G., an SBSMError message may be used by the identification protocol to trigger the need for additional SBSMInit3 messages to be sent before the Responder is satisfied with the initiators credentials.

Note that the initiation of a session can occur at either end of the protocol. E.G., a management station can establish a session with a device through which it can send management operations (E.G. for sending GETs, SETs, ...) and a managed device can also establish a session with a management station (E.G. for sending TRAPs, INFORMs, ...). Additionally, a peer MUST expect management operations of any type to be sent through a given session. EG, just because a managed device opens a session to send a notification, it must be able to accept management operations of other types (GETs, etc) to be sent from the management station to the device under the same session.

The details of how the session establishment exchange works is described in Section Section 6.5.

Once a session has been established, the security parameters switch to using the running form:



The security model sent within the SNMPv3 message is always the security model number assigned to the SBSM security model. Within the application, however, the security model assigned to the identity type is typically used which will differ from the security model number assigned to the SBSM security model. The use of these sub-security models is further discussed in the elements of procedure below (Section Section 6.

There are several differences from the way the previous User Based Security model (USM) [refs.RFC3414] worked that are important to understand. Most importantly, the User Based Security model was based on shared secrets and thus was a symmetric protocol. This is starkly different from the way the SBSM protocol works, which is asymmetric in nature. For example, two identities exist (the initiator and responder) within the SBSM session and both sides of the transaction MUST check the identity of the other side for proper authentication and authorization.

Since identity types within the security model can differ on each side (EG, one side may have an identity associated with a public key certificate and the other side may have an identity associated with a

user name and password pair), there can be two sub-security models in use within a session, one for each direction. This may seem odd to those previously familiar with the USM, but will not affect usage of the SNMP protocol's applications.

The details of how the session operates once it has been established

is described in Section Section 6.6.

5. Protocol Definitions

Here are the ASN.1 definitions that describe how the msgSecurityParameters field within the msgGlobalData [RFC3412] should be encoded. Note that the msgSecurityParameters field is an OCTET STRING, and the SBSMSecurityParameters CHOICE, defined below, would be encoded as a normal BER-encoded CHOICE/SEQUENCE and then wrapped inside the OCTET STRING when encoded into the msgSecurityParameters field of the msgGlobalData.

Many readers less familiar with ASN.1 may choose to skip to Section where the elements of procedure are defined in English text. (Section 6)

```
SBSMSecurityParametersSyntax DEFINITIONS IMPLICIT TAGS ::= BEGIN

    -- Needed data types copied from RFC3416:

    Unsigned32 ::= [APPLICATION 2] IMPLICIT INTEGER (0..4294967295)

    --
    -- TODO:
    --  1) State-keeping DoS protection for the Responder
    --  5) too-large packet sizing -- works now, document though
    --  7) handle multiple responses to requests properly.  (don't
    --     assume first message is correct unless authenticated)
    --     (Some places are documented now, need to check all spots)
    --     (mostly done.  need to double check everywhere though)
    --  8) encrypt errors when possible, use NULL when not.

    SBSMSecurityParameters ::=
        CHOICE {
            sbsm-establishment1[0]        SBSMInit1,        -- 0xA0
            sbsm-establishment2[1]        SBSMInit2,        -- 0xA1
            sbsm-establishment3[2]        SBSMInit3,        -- 0xA2
            sbsm-running[3]               SBSMRunning,      -- 0xA3
            sbsm-error[4]                 SBSMError         -- 0xA4
        }
```

```
SBSMInit1 ::=
    SEQUENCE {
        init-identifier             Unsigned32,
        dhgroup-list                NegotiationList,
        init-DH-value               NegotiationOctetList,
        init-nonce                  OCTET STRING,
        authentication-list         NegotiationList,
```

```
            encryption-list              NegotiationList,
            compression-list             NegotiationList,
            init-encryption-parameters   OCTET STRING,
            init-accepted-identity-types IdentityTypeList


        }

    SBSMInit2 ::=
        SEQUENCE {
            init-identifier              Unsigned32,
            resp-identifier              Unsigned32,
            sequence-number              Unsigned32
            dhgroup                      OBJECT IDENTIFIER,
            resp-DH-value                OCTET STRING,
            resp-nonce                   OCTET STRING
            authentication-algorithm     OBJECT IDENTIFIER,
            resp-encryption-parameters   OCTET STRING,
            encryption-algorithm         OBJECT IDENTIFIER,
            encryption-parameters        OCTET STRING,
            compression-algorithm        OBJECT IDENTIFIER,
            compression-parameters       OCTET STRING,


            -- Encrypted SBSMInit2Encr:
            resp-information             OCTET STRING,
        }

    SBSMInit2Encr ::=
        SEQUENCE {
            max-window-size              INTEGER (0..255),
            resp-engineID                OCTET STRING (0|5..32),
            resp-accepted-identity-types IdentityTypeList,


            resp-identity-type           Unsigned32,
            resp-identity                OCTET STRING,
            resp-proof1                  OCTET STRING,
            resp-proof2                  OCTET STRING,
        }

    SBSMInit3 ::=
```

```
        SEQUENCE {
            to-identifier                Unsigned32,
            sequence-number              Unsigned32
            encryption-parameters        OCTET STRING,
            compression-parameters       OCTET STRING,
            -- Encrypted SBSMInit3Encr:
```

```
            init-information              OCTET STRING,
        }

    SBSMInit3Encr ::=
        SEQUENCE {
            window-size                   INTEGER (0..255),
            init-engineID                 OCTET STRING (0|5..32)


            init-identity-type            Unsigned32,
            init-identity                 OCTET STRING,
            init-proof1                   OCTET STRING,
            init-proof2                   OCTET STRING,
        }

    SBSMRunning ::=
        SEQUENCE {
            to-identifier                 Unsigned32,
            sequence-number               Unsigned32
            authentication-parameters     OCTET STRING,
            encryption-parameters         OCTET STRING
            compression-parameters        OCTET STRING,
        }


    --
    -- Error structures
    --


    SBSMError ::=
        SEQUENCE {
            to-identifier                 Unsigned32,
            error-code                    SBSMErrorCode
            error-description             OCTET STRING,
            sequence-number               Unsigned32
            authentication-parameters     OCTET STRING,
        }


    -- numbers are synched with SNMPv2 PDU error codes just for ease
    -- of #defines and enum lists.
    SBSMErrorCode ::=
        INTEGER {
            noError(0),   -- never used
```

```
                genErr(5),
                resourceUnavailable(13),


                noSupportedAuthAlgorthim(100),
                noSupportedPrivAlgorthim(101),
                noSupportedDHGroup(102),
```

```
            insufficientNonce(103),
            insufficientEncryptionParameters(104),
            insufficientCompressionParameters(105),
            noSupportedIdentityType(106),


            incorrectIdentityType(107),
            identificationError(108),
            identityAuthenticationError(109),
            unacceptableIdentity(110),
            identityContinuationNeeded(111)
            messageAuthenticationError(112),
            messageEncryptionError(113),
            messageCompressionerror(114),


            sessionClosing(150)
            sessionClosed(151)
        }


    --
    -- Support structures
    --


    NegotiationList ::=
         SEQUENCE (SIZE (0..32)) OF OBJECT IDENTIFIER


    NegotiationOctetList ::=
         SEQUENCE (SIZE (0..32)) OF OCTET STRING


    -- This is a list of supported SNMP security models which are
    -- valid for use within a SBSM session.
    IdentityTypeList ::=
         SEQUENCE (SIZE (0..255)) OF Unsigned32


    --
    -- Security sequences for signing
    --
    -- the contents of these two sequences MUST NOT be transmitted in
    -- this form (the values are transmitted in other sequences).
    -- They exist purely for BER encoding before being signed by
    -- an identity.
```

```
SBSMResponderProofInfo ::=
    SEQUENCE {
        init-nonce                  OCTET STRING,
        resp-messages               SEQUENCE (SIZE (0..255))
                                        OF OCTET STRING
```

```
        }

    SBSMInitiatorProofInfo ::=
        SEQUENCE {
            resp-nonce                  OCTET STRING,
            init-messages               SEQUENCE (SIZE (0..255))
                                            OF OCTET STRING


        }
END
```

6. Elements of Procedure

6.1 Session State Information

When a session exists within a SNMP engine, a certain amount of state must be kept and associated with it. This amounts to the following collection of information. The data is listed as normal SNMP SMIv2 data types, but can be stored in any fashion as long as the bits on the wire end up being encoded properly as the elements of procedures require. In particular, the startTime value would be more efficiently implemented if stored as a local clock value format (like an integer value as returned by the common time() function).

```
SBSMSessionStoreDefs DEFINITIONS IMPLICIT TAGS ::= BEGIN

   Unsigned32 ::= [APPLICATION 2] IMPLICIT INTEGER (0..4294967295)

   SBSMSessionStore ::=
      SEQUENCE {
          local-identifier          Unsigned32,
          remote-identifier         Unsigned32,
          session-status            INTEGER { init1(1),
                                              init2(2),
                                              up(3),
                                              closed(4) }
          security-model            Unsigned32,
          diffieHelmanExponent      NegotiationOctetList,
```

```
        remote-nonce                    OCTET STRING,
        outgoingSequenceNumber          Unsigned32,
        incomingMinSequenceNumber       Unsigned32,
        window-size                     INTEGER (1..255),
        securityName                    OCTET STRING,
        authenticationType              OBJECT IDENTIFER,
        encryptionType                  OBJECT IDENTIFER,
```

```
            incomingEncryptionParameters     OCTET STRING,
            outgoingEncryptionParameters     OCTET STRING,
            incomingAuthenticationKey        OBJECT STRING,
            outgoingAuthenticationKey        OBJECT STRING,
            incomingEncryptionKey            OBJECT STRING,
            outgoingEncryptionKey            OBJECT STRING,
            startTime                        Unsigned32,
            legalSessionLength               Unsigned32,  -- seconds
            remoteEngineID                   OCTET STRING (0|5..32)
            -- data store array for replaying responses
            lastIncomingInit                 OCTET STRING,
            messageStoreList                 SEQUENCE (SIZE(0..255))
                                                OF SBSMMessageStore


            -- Other session information may be useful to keep in the
            -- session store, such as the remote destination
            -- transport address, etc.
        }


    SBSMMessageStore ::=
        SEQUENCE {
            sequence-number                  Unsigned32,
            timestamp                        Unsigned32,
            message                          OCTET STRING
        }
END
```

The descriptions of how the value for each field is obtained is outlined in section Section 6.5.

SNMP Engines MUST occasionally review their open session list and close any sessions where the current time minus the startTime is greater the number of seconds indicated by the legalSessionLength field (see section Section 6.1.1).

The legalSessionLength field MAY be implemented as a global system policy. IE, it is not required that each session's length be individually configurable and a global system policy may be used instead.

6.1.1 Closing sessions

When a session is closed either due to a normal operation, or due to an error condition which mandates that the session be closed, the store.session-status field should be set to closed(4) and all future traffic to such a session MUST trigger a unknownSBSMSession error condition message (described below). After a period of time defined by local policy (a suggested default is 300 seconds) or after a

maximum number of allowed closed connections is hit (a suggested default is 30), then the session should be deleted from the session store.

When a session is first set to closed, an implementation MUST zeroize the following fields:

diffieHelmanExponent
incomingEncryptionParameters
outgoingEncryptionParameters
incomingAuthenticationKey
outgoingAuthenticationKey
incomingEncryptionKey
outgoingEncryptionKey

Implementations SHOULD zeroize the entire memory contents for the session state just before the session is actually deleted from the store.

6.2 The msgSecurityModel field in the msgGlobalData

[refs.RFC3412] documents the msgGlobalData field which is used to indicate the security model is use. In the following elements of procedure, the value XXX:IANA ASSIGNMENT MUST be used. However, the VACM processing [refs.RFC3415] documents processing of authorization of incoming requests. For use within authorization processing within the VACM or any other security models, the value passed to the isAccessAllowed directive MUST be the security-model value from the current session store. IE, each identification algorithm is always transmitted across the wire using the XXX:IANA ASSIGMENT but for authorization purposes the individual identity type's specified value must be used instead.

6.3 Diffie-Helman exchange and key derivation

[this section needs a lot more work, but the basic concepts are there. A very large portion of this text was stolen from the current IKEv2 internet-draft.]

The output of a diffie-helman exchange produces a negotiated

symmetric secret key known only to the two sides of the negotiation. The keying material needed for both the authentication and encryption algorithms to be used are derived from this initial negotiated key using the following procedure. In the following text, prf indicates a pseudo-random function. This function, for purposes of this security model, is the HMAC algorithm combined with the negotiated authentication algorithm.

6.3.1 Generating Keying Material

Keying material will always be derived as the output of the negotiated message authentication algorithm (HMAC). Since the amount of keying material needed may be greater than the size of the output of the prf algorithm, we will use the prf iteratively. We will use the terminology prf+ to describe the function that outputs a pseudo-random stream based on the inputs to a prf as follows: (where | indicates concatenation)

```
prf+ (K,S) = T1 | T2 | T3 | T4 | ...
```

```
where:
T1 = prf (K, S | 0x01)
T2 = prf (K, T1 | S | 0x02)
T3 = prf (K, T2 | S | 0x03)
T4 = prf (K, T3 | S | 0x04)
```

continuing as needed to compute all required keys. The keys are taken from the output string without regard to boundaries (e.g. if the required keys are a 256 bit AES key and a 160 bit HMAC key, and the prf function generates 160 bits, the AES key will come from T1 and the beginning of T2, while the HMAC key will come from the rest of T2 and the beginning of T3).

The constant concatenated to the end of each string feeding the prf is a single octet. prf+ in this document is not defined beyond 255 times the size of the prf output.

6.3.2 Generating the session keys

```
SKEYSEED = prf(init-nonce | resp-nonce, g^ir)
```

```
{K-ai, K-ar, K-ei, K-er}
          = prf+ (SKEYSEED, g^ir | init-nonce | resp-nonce |
                  init-identifier | resp-identifier )
```

Note: the init-identifier and resp-identifier MUST be 4 bytes and stored in network byte order.

The 4 derived session keys are used for the following purposes:

K-ai: Authentication of messages from the initiator.

K-ar: Authentication of messages from the responder.

K-ei: Encryption of messages from the initiator.

K-er: Encryption of messages from the responder.

The proper use of these keys will be further discussed in the following sections.

6.4 Authenticaton and Encryption Algorithms

The negotiated authentication and encryption algorithms used by the SBSM security model duplicate those defined for the User Based Security model (USM) [refs.RFC3414]. The mechanisms for calling their ASI primitives are the same, although some minor implementation details are slightly different for use within SBSM. Future equivalent or better authentication and encryption algorithms defined in future documents for use within the SBSM framework and those documents MUST specify if there are any changes for use within the SBSM protocol. At the time of this writing, the current list of acceptable authentication and encryption algorithms include:

Authentication:

* SNMP-USER-BASED-SM-MIB::usmHMACMD5AuthProtocol

* SNMP-USER-BASED-SM-MIB::usmHMACSHAAuthProtocol

Encryption:

* SNMP-USER-BASED-SM-MIB:usmDESPrivProtocol

* SNMP-USM-AES-MIB::usmAesCfb128Protocol

NULL-equivalent authentication algorithms (IE, SNMP-USER-BASED-SM-MIB::usmNoAuthProtocol) MUST NOT be used within the SBSM framework, as both authentication and encryption algorithms will be needed to securely finish the establishment of a session.

At a minimum, the usmHMACSHAAuthProtocol protocol MUST be supported and the usmAesCfb128Protocol SHOULD be supported. Implementations MAY choose to implement the usmHMACMD5AuthProtocol and usmDESPrivProtocol values as well.

6.4.1 Differences from USM encryption algorithm implementations

One difference exists between how encryption algorithms are used within the USM and how they are used within the SBSM. Within the USM, the initialization vectors (IVs) passed to the encryption algorithms are created using the engineBoots and engineTime values,

which are not required for implementation of the SBSM protocol.

To alleviate this, when the encryption algorithms are used within the SBSM their IVs are created as follows. First a vector of the appropriate length (L) for the encryption algorithm (for DES this would be 64 bits, and for AES this would be 128 bits) is filled by concatenating first the 32 bit sequence-number encoded in network byte order (see the rest of this section for the details on calculating this value) along with a random value calculated at session initialization time for each side. The random value should be of sufficient length to fill the vector for the encryption algorthim being used. IE, if L is the required IV length in bits for an algorthim, then the vector is generated using:

```
vector = sequence-number | random(L - 32)
```

For usmDESPrivProtocol, the vector is then used as the "salt" according to section 8.1.1.1 of [refs.RFC3414].

For usmAesCfb128Protocol, the vector is then used as the IV for the protocol.

The init-encryption-parameters field of the SBSMInit1 message MUST be filled with a sufficient length vector suitable for use by any of the encryption algorithms offered in the encryption-list field.

For the algorithms mentioned in this document, the init-encryption-parameters field of the SBSMInit1 and resp-encryption-parameters field of the SBSMInit2 MUST be filled in using the random portion of the vector. For the algorithms mentioned in this document, the encryption-parameters field of the SBSMINIT3 and SBSMRunning messages MUST be left as a zero length octet string. This requires that each side retain the random portion of the vector values for the incoming and outgoing directions in the session state store in the OutgoingEncryptionParameters and IncomingEncryptionParameters fields so that the calculation of the correct IV can take place during both encryption and decryption.

These procedures are required MUST be followed for the encryption algorithms listed in this document, and MAY be used by future algorithms defined in future documents.

These procedures are designed to ensure that a given vector is never reused for a given encryption key and that the vectors are only transmitted once to reduce packet sizes for running sessions.

6.5 Creating new sessions

This section describes the process by which new sessions are created on both sides of the protocol. This is done using a handshake process that will eventually result in the creation of a valid session, or unrecoverable errors in extreme cases. Once a session is established, the procedures in Section Section 6.6 should be followed to make use of the live session.

All SBSMInit1 and SBSMInit3 messages MUST be sent with a contained PDU payload of an empty GET payload. All SBSMInit2 messages MUST be sent containing a PDU payload of an empty REPORT PDU. All SBSMInit1, SBSMInit2, SBSMInit3 and SBSMRunning messages MUST be sent with a securityModel value for the assigned SBSM security model value (see Section Section 6.2)

It should be noted that within the Session Initialization phase *only* the fields within the msgSecurityParameters field can be trusted. Modification on the wire of any of the rest of the parameters in a normal SNMPv3 message will not be detected by the security model as a session is getting set up. However, this is of no consequence since all of the values will be safely ignored or will generate errors at a higher layer (E.G., within the message processor) that will cause the packet to be dropped before it gets to the security model. No real SNMP transmitted packet is ever acted upon during session initialization, and thus only the session parameters need to be protected against modification and/or disclosure (and they are as appropriate).

6.5.1 Session initialization and generation of SBSMInit1

The sequence values of the first message should be filled in as follows:

1. The store.local-identifier field is filled in using a unique value which has not been assigned to any other session within the session store storage. An entry in the session store is created for this store.local-identifier index value.

2. The SBSMInit1.init-DH-value value is the initiator's half of the Diffie-Helman transaction. One value should be generated for

every dhgroup being offered in the dhgroup-lis field using the Diffie-Helman group information defined in Appendix Appendix A or other appropriate standards documents.

3. XXX The SBSMInit1.init-nonce value MUST be composed of randomly chosen octets and of size equal to half of the sum of the maximum key length of all the authentication algorithms

potentially in use and the maximum key length of all the encryption algorithms potentially in use. IE, length = (Ka + Ke)/2.

4. The SBSMInit1.dhgroup-list should be filled in using values supported by the local system that were desired to be used by the calling system.

5. The SBSMInit1.authentication-list and SBSMInit1.encryption-list fields are filled in using desired algorithms to be used by the session for message authentication checking and encryption (respectively). The valid values for these fields are dictated by the list of authentication and encryption protocols supported by the implementation (see Section Section 6.4 for details).

   At least one authentication and encryption algorithm MUST be specified and the list MUST NOT include any NULL-equivalent algorithms.

6. The following assignments are made relative to the recently created store and the SBSMInit1 message (some of these have already been discussed above, but are repeated here for completeness:

7. The SBSMInit1.init-encryption-parameters value MUST be randomly chosen of size equal to the maximum size value needed by any of the values in SBSMInit1.encryption-list as dictated by the selected encryption algorithm (see Section Section 6.4.1). The value is then stored in the store.outgoingEncryptionParameters field of the session store.

8. The SBSMInit1.init-accepted-identity-types field should be filled in with acceptable identity types, in order of preference, for the responder to use when returning an identity, such as those specified in Section Section 8 for a list of identity types.

9. The SBSMInit1.compression-parameters field should be filled in according to the initialization needs of the compression algorithms being proposed.

```
store.session-status = init1(1)

SBSMInit1.init-identifier = store.local-identifier
SBSMInit1.init-DH-value = generated.list.diffie-helman-half
store.diffieHelmanExponent = generated.list.diffie-helman-half
SBSMInit1.init-nonce = generated.init-nonce
SBSMInit1.dhgroup-list = policy.dhgroup-list
SBSMInit1.authentication-list = policy.authentication-list
SBSMInit1.encryption-list = policy.encryption-list
store.outgoingEncryptionParameters =
    generated.encryption-parameters
SBSMInit1.init-accepted-identity-types =
    policy.accepted-identity-types
```

10. Timers should be used to determine if a packet was lost and to retransmit the exact same copy of the SBSMInit1 message after a suitable period of time. A new SNMPv3 message MAY be created, but a new SBSMInit1 message SHOULD NOT be created and the previous exact copy should be sent instead. After a implementation dependent number of retries, the session SHOULD be deleted and an error reported to the application.

6.5.2 Reception of SBSMInit1 and generation of SBSMInit2

When a SNMPv3Message is received containing a SBSMInit1 message encoded into the securityParameters field, it MUST follow the following elements of procedure below to establish it's side of the SBSM session.

Note: If at any time during processing of the SBSMInit1 message an error occurs which prevents further processing of the message (such as insufficient resources, etc), then a SBSMError message may be returned containing a error-code of genErr or resourceUnavailable and processing should stop and the active session store deleted (if it exists yet).

1. If an existing session exists within the session store with the session-status of init1(1) and a lastIncomingInit value equal to the SBSMInit1 incoming message, the SBSMInit2 message contained in the messageStore[0].message session state MUST be resent. Processing then MUST be stopped and the packet dropped. This

processing serves to respond to retransmitted packets from the other side, but prevents recalculation on the responder's side. If no such matching session exists, it is deemed to be a new request (IE, not a retransmission of a previously sent SBSMInit1 message) and processing continues.

2. If an existing session exists within the session store with the session-status of init2(1) and a lastIncomingInit value equal to the SBSMInit1 incoming message, the SBSMInit1 message must be dropped.

3. The lists of offered message authentication and encryption algorithms (authentication-list, encryption-list) are examined for support and accepted values. One of each type MUST be selected and the first in each list that is acceptable SHOULD be the one selected. The resulting selected algorithms are later referred to as authAlgorthim and encrAlgorthim.

4. If a message authentication algorithm, encryption algorithm or diffie-helman group can't be picked (E.G., they are unsupported or administratively prohibited), then: A SBSMError message should be returned to the sender with a error-code of either unsupportedAuthAlgorthim, unsupportedPrivAlgorthim or unsupportedDHGroup. The message is dropped and further processing is stopped.

5. If the init-nonce value is not of sufficient length to support the selected authentication and encryption algorithm (See section Section 6.5.1 for length requirement details), then: A SBSMError message should be returned to the sender with a error-code of insufficientNonce. The message is dropped and further processing is stopped.

6. The list of offered identity types, found in the init-accepted-identity-types field, is examined. If multiple acceptable identities are listed, then the first acceptable value in the list SHOULD be selected although responder implementations MAY choose to select a different one based on local policy. We will refer to this selection later as policy.selectedOutgoingIdentityType. If no acceptable identity type is found within the list then an SBSMError message should be returned with a error-code of noSupportedIdentityType

7. If the encryption algorithm chosen requires the use of the init-encryption-parameters field and it is not of sufficient length, then: A SBSMError message should be returned to the sender with a error-code of insufficientEncryptionParameters. The message is dropped and further processing is stopped.

8. If the compression algorithm chosen requires the use of the init-compression-parameters field and it is not of sufficient length, then: A SBSMError message should be returned to the sender with a error-code of insufficientCompressionParameters. The message is dropped and further processing is stopped.

9. A unique local-identifier is generated and a new session store is created to store session parameters. A SBSMInit2 response message (including the SBSMInit2Encr SEQUENCE) is also created. The two structures are filled in using the following guidelines:

```
SBSMInit2.init-identifier = SBSMInit1.init-identifier
SBSMInit2.resp-identifier = store.local-identifier
SBSMInit2.dhgroup = policy.dhgroup
SBSMInit2.authentication-algorithm = policy.authAlgorthim
SBSMInit2.encryption-algorithm = policy.encrAlgorthim
SBSMInit2.sequence-number = 0
```

```
store.session-status = init1(1)
store.local-identifier = generated.identifier
store.remote-identifier = message.init-identifier
store.authenticationType = policy.authAlgorthim
store.encryptionType = policy.encrAlgorthim
store.init-encryption-parameters =
    SBSMInit1.init-encryption-parameters
store.outgoingSequenceNumber = 0
store.incomingSequenceNumber = 0
store.startTime = generated.now
store.legalSessionLength = policy.session-length
store.lastIncomingInit = message
```

```
// Other values may be needed or desired by implementations.
```

10. The store's resp-DH-value value is the responder's half of the Diffie-Helman transaction using the Diffie-Helman group defined by the accepted SBSMInit2.dhgroup field and the related values found in Appendix Appendix A or other appropriate standards documents. It's value is stored in the resp-DH-value field of the SBSMInit2 message.

11. The resp-nonce value MUST be randomly chosen and of size equal to half of the sum of the maximum key length of all the authentication algorithms potentially in use and the maximum key length of all the encryption algorithms potentially in use. IE, length = (Ka + Ke)/2.

```
SBSMInit2.resp-nonce = generated.resp-nonce
```

12. A suitable value for the SBSMInit2.resp-encryption-parameters field MUST be randomly chosen of size equal to the needed as dictated by the selected encryption algorithm (see Section Section 6.4.1). The value is then stored in both the store and the SBSMInit2 message:

```
store.outgoingCompressionParameters =
     generated.encr-parameters
```

13. A suitable value for the SBSMInit2.resp-compression-parameters field MUST be chosen according to the compression algorithm in use (see Section Section 9).

```
store.outgoingCompressionParameters =
     generated.compression-parameters
```

14. The session keys (K-ai, K-ar, K-ei, and K-er) are derived from the Diffie-Helman derived secret key (g^ir) and the init-nonce and resp-nonce values according to the procedures in Section Section 6.3.

15. These keys are stored in the session store according to the following mapping:

```
store.incomingAuthenticationKey = generated.K-ai
store.outgoingAuthenticationKey = generated.K-ar
store.incomingEncryptionKey = generated.K-ei
store.outgoingEncryptionKey = generated.K-er
```

16. The max-window-size field of the SBSMInit2Encr sequence are filled in according to local policy.

```
SBSMInit2Encr.max-window-size = policy.selected-window-value
```

17. The resp-engineID field is filled in with a suitable default engineID which can be used in the engineID field of a ScopedPDU for transmissions requiring them from the remote side, or a zero length string if no value is suitable.

```
SBSMInit2Encr.resp-engineID = policy.local-engineID
```

18. The resp-accepted-identity-types field should be filled in with acceptable identity types, in order of preference, for the initiator to use when returning an identity. For a list of identity types specified by this document, see Section Section 8

for a list of identity types.

```
SBSMInit2Encr.resp-accepted-identity-types =
    policy.accepted-identity-types
```

19. The resp-identity-type and resp-identity fields are filled in using the policy.selectedOutgoingIdentityType value selected above and the identity value for that type to be transmitted to the initiator. The proper format for this field is dictated by

the resp-identity-type value being used and its associated implementation details in Section Section 8.

```
SBSMInit2Encr.resp-identity-type =
    policy.selectedOutgoingIdentityType
SBSMInit2Encr.resp-identity = policy.identity
```

20. The securityName field of the session store is derived from the same policy.selectedOutgoingIdentityType security model's identity mapping transform, also described in Section Section 8. XXX: should be bi-directional sec names

21. The resp-proof2 field is filled in using the results of a message authentication signature created using the algorithm indicated by the store.authenticationType value and the store.outgoingAuthenticationKey key to sign the contents of the resp-identity field.

22. The resp-proof1 field is filled in using an identity authentication signature created using the key and signing algorithm associated with resp-identity (see Section Section 8) to sign an encoded SBSMResponderProofInfo SEQUENCE. This sequence includes the nonce value sent by the initiator as well as all of the messages sent by the responder to the initiator up till and including this message being sent. To include this message, it must be first encoded in its entirety except for the resp-proof1 field, which should be left as a proper length field containing as many 0x00 value octets as is needed to fill the field. Once the signature has been created, the field should be filled in with the newly generated value.

    Note: The init-nonce field within the SBSMResponderProofInfo sequence operation MUST include the BER tag and length fields from the on-the-wire packet format.

    To fill in the resp-information field during this identity authentication step, use the plain-text version of the SBSMInit2Encr sequence wrapped in an OCTET STRING and placed into the resp-information field.

23. The entire SBSMInit2Encr SEQUENCE is encoded according to BER

encoding rules and the resulting byte sequence is then encrypted using the store.encryptionType algorithm and the store.outgoingEncryptionKey key. The resulting cyphertext bytes are then stored, after being wrapped in an OCTET STING, in the resp-information field within the SBSMInit2 SEQUENCE.

24. The entire SBSMInit2 message, once constructed, is returned to

the sender of the initial SBSMInit1 message as a REPORT message.

25. The messageStore[0].message value is set to the entire encoded SBSMInit2 SEQUENCE. The session store is stored for later retrieval.

    store.messageStore[0] = SBSMInit2

26. Success is returned to the calling module, along with the contents of the SBSMInit1 packet to be sent. Note that the packet returned MUST NOT be processed by an application.

6.5.3 Reception of SBSMInit2 and generation of SBSMInit3

When a SNMPv2Message is received containing a SBSMInit2 message encoded into the securityParameters field, it MUST follow the elements of procedure below to finish establishing it's side of the SBSM session:

1. The local session store is examined to determine if a session exists where the store.local-identifier field matches the SBSMInit2.init-identifier field. If not, the message is dropped and processing is ceased. If one is found but the store.session-status field is set to up(2), the message is also dropped and processing is ceased. XXX: the only legal value should be init1

2. The anti-replay processing discussed in Section Section 6.9.2 is performed. This should only happen when a responder needed to retransmit an SBSMInit2 message that was deemed to be lost. After the SBSMInit3 message is retransmitted, further processing of the incoming SBSMInit2 message is stopped.

3. The diffie-helman exchange is completed using the appropriate store.diffieHelmanExponent value and the SBSMInit2.resp-DH-value value. This should produce a g^ir value.

4. The session keys (K-ai, K-ar, K-ei, and K-er) are derived from the Diffie-Helman derived secret key (g^ir) and the init-nonce

and resp-nonce values according to the procedures in Section Section 6.3.

5. The SBSMInit2.resp-information field is decrypted using the encryption type specified by the SBSMInit2.encryption-algorithm field, the SBSMInit2.encryption-parameters field and the generated.K-er key to produce a decrypted but possibly still compressed SBSMInit2Encr SEQUENCE. The results are then

decompressed using the compression algorithm defined and the SBSMInit2.compression-parameters field. The values of the SBSMInit2Encr field are then parsed. If the values can not be parsed, then the snmpInASNParseErrs counter [RFC3418] is incremented, and an error indication (parseError) is returned to the calling module.

6. The list of offered identity types, found in the SBSMInit2Encr.resp-accepted-identity-types field, are examined. If no acceptable identity type is found within the list then an authenticated and encrypted SBSMError message should be returned to the responder with an error-code of noSupportedIdentityType. If multiple acceptable identities are listed, then the first acceptable value in the list SHOULD be selected although responder implementations MAY choose to select a different one based on local policy. We will refer to this selection later as policy.selectedOutgoingIdentityType.

7. The SBSMInit2Encr.resp-identity-type field is examined and checked to see if the identity type matches one of the types sent in the initial SBSMInit1 message and that it matches locally acceptable identity types. If not, then an authenticated and encrypted SBSMError message should be returned to the responder with an error-code of incorrectIdentityType.

8. The SBSMInit2Encr.resp-identity field is examined, according to the security model and associated parameters (see Section Section 8) and if it does not match the expected identity for the other side of the session, processing is stopped and an authenticated and encrypted SBSMError message should be returned to the responder with an error-code of identificationError.

9. The SBSMInit2Encr.resp-proof1 field value is checked to ensure it matches the expected signature as described in Section Section 6.5.2 using the signing mode described by the security model in Section Section 8. If the signature in the SBSMInit2Encr.resp-proof1 field does not match the output of the signature alogrithm, then processing is stopped and an authenticated and encrypted SBSMError message should be returned to the responder with an error-code of identityAuthenticationError. Note that the session MUST NOT be dropped due to this error since a packet may arrive from the real identity with proper credentials.

10. the SBSMInit2Encr.resp-proof2 field value is checked to ensure it matches the expected message authentication signature as described in Section Section 6.5.2 using the authentication mode described by the store.authentication-algorithm field along with

the store.incomingAuthenticationKey key. If the signature in the SBSMInit2Encr.resp-proof2 field does not match the output of the authentication alogrithm, then an authenticated and encrypted SBSMError message should be returned to the responder with an error-code of messageAuthenticationError.

11. Now that the SBSMInit2 message has been deemed authentic, the initiator can fully establish its side of the session parameters:

```
store.session-status = init2(2)
store.outgoingAuthenticationKey = generated.K-ai
store.incomingAuthenticationKey = generated.K-ar
store.outgoingEncryptionKey = generated.K-ei
store.incomingEncryptionKey = generated.K-er
store.authenticationType = SBSMInit2.authentication-algorithm
store.encryptionType = SBSMInit2.encryption-algorithm
store.remoteEngineID = SBSMInit2.resp-engineID
store.incomingSequenceNumber = SBSMInit2.sequence-number
store.window-size =
    min(policy.window-size, SBSMInit2Encr.max-window-size)
store.startTime = generated.now
store.legalSessionLength = policy.session-length
```

A. The store.diffieHelmanExponent memory contents is erased/ zeroed.

B. The store.securityName field is filled in using the mapping described by the security model to extract it from the SBSMInit2.resp-identity field.

12. The SBSMInit3 SEQUENCE is created, as is a SBSMInit3Encr SEQUENCE, and its values are filled in as follows:

```
store.outgoingSequenceNumber = store.outgoingSequenceNumber + 1

XXX: local-identifier not needed in init3?
SBSMInit3.to-identifier = store.remote-identifier
SBSMInit3Encr.window-size = store.window-size
SBSMInit3.sequence-number = store.outgoingSequenceNumber
```

A. The init-engineID field is filled in with a suitable default engineID which can be used in the engineID field of a ScopedPDU for transmissions requiring them from the remote side, or a zero length string if no value is suitable:

SBSMInit3Encr.init-engineID = policy.local-engineID

B. The init-identity-type and init-identity fields are filled in using the identity-type value selected above and the identity value for that type to be transmitted to the initiator. The proper format for this field is dictated by the init-identity-type value being used and its associated implementation details in Section Section 8.

```
SBSMInit3Encr.init-identity-type =
    policy.selectedOutgoingIdentityType
SBSMInit3Encr.init-identity = policy.identity
```

C. The SBSMInit3Encr.init-proof1 field is filled in using a authentication signature created using the key associated with SBSMInit3Encr.init-identity (see Section Section 8) to sign an encoded SBSMInitiatorProofInfo SEQUENCE, which includes the necessary fields from the SBSMInit1, SBSMInit2 and SBSMInit3 messages to ensure proper authentication can be determined by the responder.

The SBSMResponderProofInfo SEQUENCE is encoded using the actual BER encoded values taken from the on-the-wire messages. They MUST be identical copies of the transmitted values using the exact same encoding as was transmitted on the wire. Care must be taken that the values transmitted by the other side are used exactly how they were sent.

Note: Identity authentication schemes which require multiple negotiation might specify that the init-proof1 field is to be left blank and implementations supporting negotiating identity authentication mechanisms should except this.

D. The SBSMInit3Encr.init-proof2 field is filled in using a HMAC authentication signature created using the store.outgoingAuthenticationKey to sign the contents of the SBSMInit3Encr.init-identity field using the store.authenticationType algorithm.

Note: The resp-identity field value within the HMAC operation includes the BER tag and length fields from the on-the-wire packet format.

Note: Identity authentication schemes which require multiple negotiation might specify that the init-proof2 field is to be left blank and implementations supporting negotiating identity authentication mechanisms should except this.

E. The SBSMInit3Encr SEQUENCE is encrypted using the store.encryptionType encryption algorithm and

store.outgoingEncryptionKey and then wrapped in an OCTET STRING and placed into the SBSMInit3.resp-information field. For the vector generation, described in Section Section 6.4.1, a sequence-number of SBSMInit3.sequence-number MUST be used.

13. The initiator of the session MAY begin transmitting messages under protection of the newly created session at this point, assuming the identity authentication for the initiator is complete. However, it should be noted that the SBSMInit3 message may not have been received by the responder and thus retransmissions may be necessary at a future time. Thus, the initiator SHOULD wait for the reception of a proper SBSMRunning acknowledgment message first.

14. Timers should be used to determine if the SBSMInit3 message was lost (IE, no SBSMRunning acknowledgment message was received) and to retransmit the exact same copy of the SBSMInit3 message after a suitable period of time (as dictated by local policy). A new SNMPv3 message MAY be created, but a new SBSMInit3 message MUST NOT be created and the previous exact same bitwise copy MUST be sent. Once the local session status has been set to up(3) by related processing from Section Section 6.6.2 then retransmissions of SBSMInit3 MUST stop. After an implementation dependent number of retries, the session SHOULD be deleted and a failure should be returned to the application which requested session creation.

15. Success is returned to the calling module, along with the contents of the packet to be sent. Note that the packet returned MUST NOT be processed by an application and is only intended for use by the SNMP engine.

6.5.4 Reception of the SBSMInit3 message and generation of the SBSMRunning REPORT

When a SNMPv3Message is received containing a SBSMInit3 message encoded into the securityParameters field, it MUST follow the elements of procedure below to finish establishing it's side of the SBSM session:

1. The local session store is examined to determine if a session exists where the store.local-identifier field matches the SBSMInit3.to-identifier field. If not, the message is dropped and processing is ceased.

2. If the store.session-status field is already set to up(2) then

processing continues as in Section Section 6.6.2, as the store.messageStore must contain an already generated answer for the SBSMInit3 message. XXX move to an independent section. XXX state should only be one state (init1?)

3. The SBSMInit3.init-information field is decrypted using the store.encryptionType and store.incomingEncryptionKey fields from the session store to produce a decrypted SBSMInit3Encr SEQUENCE. The values of the SBSMInit3Encr field are then parsed. If the values can not be parsed, then the snmpInASNParseErrs counter [RFC3418] is incremented, and an error indication (parseError) is returned to the calling module.

4. The SBSMInit3Encr.init-identity field is examined, according to the security model and associated parameters (see section Section 8) and if it does not match an acceptable identity for the other side of the session then: an authenticated SBSMError message should be returned to the responder with an error-code of unacceptableIdentity, processing is stopped and an error indication (authenticationFailed?) is returned. The store can be freed and the session establishment dropped.

5. The SBSMInit3Encr.init-proof1 field value is checked to ensure it matches the expected signature as described in Section Section 6.5.3 using the signing mode described by the security model in section Section 8.

   If the identity authentication mechanism specifies that further processing is needed, a SBSMError message should be returned to the responder with an error-code of identityContinuationNeeded and a error-description field specificly encoded according to the needs of identity authentication mechanism. Processing of the incoming message should be stopped, although the session should be left in its current state.

   If the signature in the SBSMInit3Encr.init-proof1 field does not match the output of the signature alogrithm, then: an authenticated SBSMError message should be returned to the responder with an error-code of identityAuthenticationError, processing is stopped and an error indication (authenticationFailed?) is returned. The store can be freed and the session establishment dropped.

6. The SBSMInit3Encr.init-proof2 field value is checked to ensure it matches the expected message authentication signature as described in Section Section 6.5.3 using the authentication mode described by the store.authenticationType field along with the store.incomingAuthenticationKey.

If the identity authentication mechanism specifies that further processing is needed, a SBSMError message should be returned to the responder with an error-code of identityContinuationNeeded and a error-description field specificly encoded according to the needs of identity authentication mechanism. Processing of the incoming message should be stopped, although the session should be left in its current state.

If the signature in the SBSMInit3Encr.init-proof1 field does not match the output of the message authentication alogrithm, then: an authenticated SBSMError message should be returned to the responder with an error-code of messageAuthenticationError, processing is stopped and an error indication (authenticationFailed?) is returned. The store can be freed and the session establishment dropped.

7. Now that the SBSMInit3 message has been deemed authentic and the remote identity has been verified, the responder can fully establish the remaining portions of its side of the session parameters:

```
store.session-status = up(2)
store.window-size =
    min(policy.maximum-window-size, SBSMInit3Encr.window-size)
store.startTime = generated.now
store.legalSessionLength = policy.session-length
store.remoteEngineID = SBSMInit3.init-engineID
```

A. The store.securityName field is filled in using the mapping described by the security model to extract it from the init-identity field.

8. A REPORT message is generated with a PDU containing the sbsmSessionsEstablished counter after it has been incremented. The REPORT is sent using the newly created session with a security level of authPriv and the mechanisms described in Section Section 6.6.1 using a SBSMRunning message. This message will be the first message sent over the live session, from the viewpoint of the responder and serves as an acknowledgment to the initiator that the SBSMInit3 message was received.

9. The responder of the session may begin transmitting messages under protection of the newly created session at this point.

10. XXX: store processing

6.6 Processing messages in an active session.

Once a session has been established, messages may be then sent and received through it using the procedures defined in this section.

6.6.1 Outgoing Messages on an open session.

This section describes the procedure followed by an SNMP engine whenever it generates a message containing a management operation (like a request, a response, a notification, or a report) through an open SBSM session. The elements of procedure below define how to fill in the values within the sbsm-running element which is then encoded by wrapping it as an OCTET STRING and placing it in the SNMPv3Message's msgSecurityParameters field.

1. If any securityStateReference is passed (EG, for a Response or Report message), then information concerning the session is extracted from the storedSecurityData. The storedSecurityData can now be discarded after its two values, the local-identifier and the from-sequence-number fields, are extracted from the securityStateReference.

2. If a securityStateReference is not passed, then a local-identifier must have been passed. If not, then a error indication (unknownSBSMSession) is returned to the calling module.

3. The security session state is looked up based on the value of the local-identifier parameter, and if not found then an error indication (unknownSBSMSession) is returned to the calling module.

4. If the current time minus the store.startTime is greater than the number of seconds from the store.legalSessionLength field (or any other value from a policy that restricts session time lengths), then the session MUST be immediately closed (see Section Section 6.1.1 for information on closing a session) and a unknownSBSMSession error returned to the calling module.

   Applications MAY choose to initiate another session under which

the new message will be sent if the message type is not in reponse to another message (E.G., a Response-PDU or a Report-PDU and thus no storedSecurityData was passed in and thus no from-sequence-number value is available). If it is a reponse-class message then no new session is open and processing of the PDU MUST be dropped after the unknownSBSMSession error is returned.

5. If the passed securityLevel specifies that the message is to be protected from disclosure, but the session does not support both an authentication and a encryption protocol then the message cannot be sent. An error indication (unsupportedSecurityLevel) is returned to the calling module.

6. If the securityLevel specifies that the message is not to be authenticated, then the message cannot be sent. An error indication (unsupportedSecurityLevel) is returned to the calling module.

   SBSMRunning.to-identifier = store.remote-identifier

7. The procedures for anti-replay protection described in Section Section 6.9.1 MUST be followed at this point.

8. If the session is making use of a compression algorithm, then the ScopedPDU is compressed according to the store.compression-algorithm and any compression-parameters required by the algorithm are stored in the SBSMRunning.compression-parameters field.

9. Possibly encrypt the ScopedPDU

   A. If the securityLevel specifies that the message is not to be protected from disclosure, then a zero-length OCTET STRING is encoded into the SBSMRunning.encryption-parameters field and the plaintext scopedPDU serves as the payload of the message being prepared.

   B. If the securityLevel specifies that the message is to be protected from disclosure, then the octet sequence representing the serialized scopedPDU is encrypted according to the store.encryptionType encryption protocol (see Section Section 6.4 for more details) . To do so a call is made to the encryption module that implements the store.encryptionType encryption algorithm using the store.outgoingEncryptionKey as the encryption key.

      If the encryption module returns failure, then the message

cannot be sent and an error indication (encryptionError) is returned to the calling module.

If the encryption module returns success, then the returned privParameters (if any, see Section Section 6.4.1) are put into the SBSMRunning.encryption-parameters field and the encryptedPDU serves as the payload of the message being prepared.

10. The message is authenticated according to the session's authentication protocol (see Section Section 6.4). To do so a call is made to the authentication module that implements the store.authenticationType algorithm using the store.outgoingAuthenticationKey as the authentication key.

    If the authentication module returns failure, then the message cannot be sent and an error indication (authenticationFailure) is returned to the calling module.

    If the authentication module returns success, then the authentication-parameters field is put into the sbsm-running and the authenticatedWholeMsg represents the serialization of the authenticated message being prepared.

11. The completed message with its length is returned to the calling module with the statusInformation set to success.

6.6.2 Incoming Messages on an open session.

This section describes the procedure followed by an SNMP engine whenever it receives a message sent through an active SBSM session with a particular securityLevel.

XXX: delete? To simplify the elements of procedure, the release of state information is not always explicitly specified. As a general rule, if state information is available when a message gets discarded, the state information should also be released. Also, an error indication can return an OID and value for an incremented counter and optionally a value for securityLevel, and values for engineID or contextID (from ScopedPDU) for the counter. In addition, the securityStateReference data is returned if any such information is available at the point where the error is detected.

1. If the received securityParameters is not the serialization (according to the conventions of [RFC3417]) of an OCTET STRING formatted according to the SBSMSecurityParameters defined in section 2.4, then the snmpInASNParseErrs counter [RFC3418] is incremented, and an error indication (parseError) is returned to the calling module. Note that we return without the OID and

value of the incremented counter, because in this case there is not enough information to generate a Report PDU.

2. The values of the SBSMRunning fields are extracted and the value of the SBSMRunning.to-identifier is used to look up a session store with a store.local-identifier equal to the SBSMRunning.to-identifier. If no such session store is found,

processing is stopped and a unknownSBSMSession is returned to the calling module.

3. If the current session is considered closed, then an unknownSBSMSession error returned to the calling module.

4. check against valid session states... must be up or just have sent init3 right?

5. If the current time minus the store.startTime is greater than the number of seconds from the store.legalSessionLength field (or any other value from a policy that restricts session time lengths), then the session MUST be immediately closed (see Section Section 6.1.1 for information on closing a session) and an unknownSBSMSession error returned to the calling module.

6. If the SNMPv3 message's securityLevel specifies that the message was not authenticated, then processing is stopped, the sbsmStatsUnsupportedSecLevels counter is incremented and an error indication (unsupportedSecurityLevel) together with the OID and value of the incremented counter is returned to the calling module. IE, message authentication is a requirement of the SBSM security model.

7. The anti-replay processing described in Section Section 6.9.2 MUST be followed at this point.

8. The store.session-status field is set to up(3) if it is not set to up(3) yet (IE, it might be set to init2(2) if no response had been received from the SBSMInit3 message yet).

9. If the SNMPv3 message's securityLevel indicates that the message was not protected from disclosure, then the scopedPDU is assumed to be in plain text format.

10. If the securityLevel indicates that the message was protected from disclosure, then the OCTET STRING representing the encryptedPDU is decrypted according to the store.encryptionType encryption protocol, the message.encryption-parameters field and the store.incomingEncryptionKey to obtain an unencrypted

serialized scopedPDU value. To do so a call is made to the encryption module that implements the store.encryptionType encryption protocol using the store.incomingEncryptionKey as the encryption secret key.

If the encryption module returns failure, then the message can not be processed, so the sbsmStatsDecryptionErrors counter is incremented and an error indication (messageEncryptionError)

together with the OID and value of the incremented counter is returned to the calling module.

If the encryption module returns success, then the decrypted scopedPDU is used as the message payload to be later returned to the calling module.

11. If the session.compression-algorithm field indicates the packet should be compressed, then the compression algorithm is passed the scopedPDU field and the message.compression-parameters field and the results are used as the new scopedPDU. If the compression algorithm fails, then the sbsmStatsCompressionErrors counter is incremented and an error condition (messageCompressionError) is returned to the calling module and processing is stopped.

12. If the PDU contained within the scopedPDU is a REPORT message of type sbsmSessionEstablished, then the message is dropped and processing is stopped. Success is returned to the calling module.

13. The maxSizeResponseScopedPDU is calculated. This is the maximum size allowed for a scopedPDU for a possible Response message. Provision is made for a message header that allows the same securityLevel as the received Request.

14. The securityName to be used when processing this message is retrieved from store.securityName.

15. The security data is stored as storedSecurityData, so that a possible response to this message can and will use the same authentication and encryption parameters. Information to be saved/stored is as follows:

    local-identifier

    sequence-number

16. The store.messageStoreList[sequence-number mod window-size].sequence-number field from the session store is set

to the SBSMRunning.sequence-number of the incoming message.

17. The statusInformation is set to success and a return is made to the calling module passing back the OUT parameters as specified in the processIncomingMsg primitive. Note that the application, especially for purposes of access control determination, should process the message as if it came through a security module equivalent to the security-model from the session store. IE, a

application should not need be aware of SBSM processing but should only be aware of the identity mechanism used instead, which maps to a real SNMP security model number.

6.7 Processing SBSMError messages.

6.7.1 Processing outgoing SBSMError messages.

Outgoing error codes are generated using a SBSMError message, used only when indicated by the elements of procedure in either Section Section 6.5 or Section Section 6.6, are combined with a REPORT PDU containing a varbind with an incremented sbsmProtocolError counter contained within it and sent in response to the previous message that triggered an error. The procedure for generating the values of the SBSMError message are as follows. These procedures assume that a session store has already been found to process the error in.

1. A SBSMError sequence is created and the SBSMError.error-code and SBSMError.error-description fields are filled according to the passed values.

2. The procedures set forth in Section Section 6.6.1 should generally be followed, although XXX

6.7.2 Processing incoming SBSMError messages.

When SBSMError mesasges are received either during session creation or during a running session, they should be processed according to the following procedures, depending on the value of the contained error-code. If no store.local-identifier matches the SBSMError.to-identifier, then the message is to be dropped and processing of the error message is ceased.

1. When the SBSMError.to-identifier matches a store.local-identifier and the store.session-status is equal to anything other than up(3), then the following error codes are all processed in the same manner:

* genErr(5)

* resourceUnavailable

* noSupportedAuthAlgorthim

* noSupportedPrivAlgorthim

* insufficientNonce

* insufficientEncryptionParameters

* noSupportedIdentityType

* incorrectIdentityType

* identificationError

* identityAuthenticationError

* unacceptableIdentity

* messageAuthenticationError

Specifically, all these errors indicate that the remote and gin was unable to process a SBSMInit1 message properly due to an unsupported value.

Most importantly, an implementation MUST NOT accept one of these error messages as authoritative. They have not been cryptographically signed and are thus untrustworthy. Even if they have been properly signed by the negotiated diffie-helman key, the identity of the remote side has yet to be proven and thus the packet may be the work of an imposter (a man-in-the-middle). They are, however, useful for informational purposes. Upon reception of one of these messages, an action SHOULD NOT be taken until a suitable time out has passed, and no other corresponding error message or SBSM message was received.

Reception of one of these messages can be dealt with and one of two ways, after the suitable timeout period has passed:

M. Cease attempting the establishment of a session, delete the local corresponding session store, and log an error. The SBSMError.error-description field might contain a message intended for human consumption and implications.

N. Attempt to renegotiate the establishment of a session after fixing the parameters associated with the error code in question. After new parameters have been generated, the SBSM message may be sent again containing the new parameters, assuming the session is in a suitable state to do so (E.G., it is known that the other side of the connection hasn't closed their side of the negotiation).

2. When the SBSMError.to-identifier matches a store.local-identifier and the store.session-status is equal to anything other than up(3) and the error-code is set to identityContinuationNeeded and the message is properly protected, then the error-description field should be passed to the identity authentication module as part of a continuing series of identification messages. The identity authentication module should then indicate what should happen next: either another SBSMInit3 message should be sent to the responder, or the session should be closed as an unrecoverable failure was hit (see Section Section 6.1.1 for information on closing a session). When further SBSMInit3 fields are sent, it is likely that new init-proof1 and a new init-proof2 field should be encoded into the new SBSMInit3 message. Other than that, the other fields can be calculated according to the procedures outlined in Section Section 6.5.3.

XXX: sequence-number and caching check

6.8 Closing an active session from either side

Although implementations must expect the case where the other side of a session may have lost its session information, when possible closing messages should be sent in order to assist in resource-freeing and other cleanup tasks.

To send a closing-session message, an empty GET PDU is constructed and sent to the opposite side with a SBSMError message where the error-code value is set to sessionClosing. This is responded to from the other side with a sessionClosed SBSMError message sent coupled with a REPORT PDU. If a REPORT message is not received by the side initiating the session closing, it SHOULD resend the close request (retrying with a suitable number of close attempts over a suitable period of time). Note that if the remote side has already closed its session, it will send a unknownSBSMSession SBSMError REPORT although it will be unable to properly authenticate it.

When closing a session, the session cache must be cleaned according to the description in Section Section 6.1.1.

6.9 Processing the SBSM messages for anti-replay support.

there are multiple points within the SBSM protocol where messages may be resent or retransmitted either intentionally by one side of the session or maliciously by an attacker. For these cases, the SBSM protocol is designed to officially take care of such messages. This section describes the processing required to support this for both incoming and outgoing messages of all types.

The most fundamental concept to understand is that once a response to an incoming message it is stored by the security model so that the application and SNMP engine does not have to regenerate the response at a future time. Each SBSM message contains a sequence-number field, which is monotonically increasing in nature, deficit this sequence number that helps provide this replay protection. The size of this outgoing cash is dependent on a number of factors: the local implementation supported maximum, the locally defined policy, and the negotiated window-size parameter.

Another important point is that the real transmission of previous requests do not result in an application reprocessing the request. Only the security model and the SNMP engine needs to be aware of the reach transmission, which should save computational cycles and used to lead to denial of service attacks. It should be noted that message indication and encryption services are still exercised again, but this competition should be lower than what would be needed to completely recompute a new response. It is recommended that engines which currently deal with retransmissions of lost requests now do so using these services when possible.

6.9.1 Processing outgoing messages

Anytime a new outgoing message is being sent which is a direct response to an incoming message, it is stored as follows:

```
/* update the caching information for the current message */
store.outgoingSequenceNumber = store.outgoingSequenceNumber + 1
outgoingMessage.sequence-number = store.outgoingSequenceNumber
```

If the outgoing message is in response to an incoming message, then the sequence-number field must be available and the following processing is performed:

```
/* put the outgoing message in the store in the right place */
tempvar.N =
   incomingMessage.sequence-number mod store.window-size

store.messageStore[tempvar.N].sequence-number =
```

```
          incomingMessage.sequence-number
     store.messageStore[tempvar.N].message =
          outgoingMessageData                    /* see below */
     store.messageStore[tempvar.N].timestamp = generated.now()
```

If the value of the store.outgoingSequenceNumber wraps in the above process, the session MUST be immediately closed (see Section Section

6.1.1 for information on closing a session)and unknownSBSMSession error returned to the calling module.

For the outgoingMessageData, whatever data is needed to reconstruct the response properly should be stored here. For messages which are specific to the negotiation portion of the SBSM protocol, this would generally be the protocol fields as well as the possibly-mostly-blank ScopedPDU. In general, sensed this specification can not mandate that the other side of the session use an identical SNMPv3 message, it must be possible to receive a new SNMPv3 message from the other side which contains a new msgID field and still be able to reauthenticate the message without regenerating any of the other SBSM fields or the enclosed ScopedPDU. For outgoing messages on a running session, the only outgoingMessageData that is to be saved should be the unencrypted ScopedPDU. Note that the unencrypted version must be saved since the retransmission of the SNMPv3 message may have dropped the encryption flag.

6.9.2 Processing Incoming Messages

When an incoming message is received, it should be subject to the following processing procedures for all message types except SBSMInit1 messages:

1. If the incomingMessage.sequence-number is less than the store.incomingMinSequenceNumber, then an error indication (authenticationFailure) is returned to the calling module and processing is stopped.

2. If the message is an SBSMInit2 or SBSMInit3 message or is a SBSMRunning message but is not a response class message, and the second three of both these statements are both true:

```
tempvar.N =
   incomingMessage.sequence-number mod store.window-size

incomingMessage.sequence-number <
       store.incomingMinSequenceNumber + store.window-size

incomingMessage.sequence-number ==
```

```
            store.messageStoreList[tempvar.N].sequence-number

         store.messageStoreList[tempvar.N].timestamp >=
                generated.now() + 300 seconds


        Then: the message stored in the
```

store.messageStoreList[tempvar.N].message field is returned as the answer to the incoming request and is returned to the calling module. An application MUST NOT process this request and the resulting response message contained within the store message MUST be used to generate a duplicate response. Processing should then continue through the outgoing processing steps for the given outgoing message type, but using the store.messageStoreList[tempvar.N].message value as the returned message.

3. If the message is a response class message, and if both of the following two statements are true:

```
 incomingMessage.sequence-number <
          store.incomingMinSequenceNumber + store.window-size

    store.messageStoreList[tempvar.N].sequence-number ==
          incomingMessage.sequence-number field
```

Then the message is dropped as it has already been previously received.

4. The message's authentication is checked according to the store.authenticationType authentication protocol and store.incomingAuthenticationKey. To do so a call is made to the authentication module that implements the store.authenticationType authentication protocol using the store.incomingAuthenticationKey as the authentication secret key.

   If the authentication module returns failure, then the message cannot be trusted, so the sbsmStatsWrongDigests counter is incremented and an error indication (sbsmIntegrityFailure) together with the OID and value of the incremented counter is returned to the calling module.

   If the authentication module returns success, then the message is authentic and can be trusted so processing continues.

5. The store.incomingMinSequenceNumber is then updated: to be the maximum of:

```
store.incomingMinSequenceNumber =
     min(store.incomingMinSequenceNumber,
         incomingMessage.sequence-number - store.window-size)
```

If the new incomingMinSequenceNumber number wraps or is set to 2^32-1-store.window-size, then the session MUST be closed after this message is processed (see Section Section 6.1.1 for

information on closing a session).

7. MIB Definitions

The MIB included below is only minimal in nature (obviously) but it is a start. Feedback on useful objects to be placed into this MIB would be highly appreciated.

```
SBSM-MIB DEFINITIONS ::= BEGIN

IMPORTS
    MODULE-IDENTITY, OBJECT-TYPE, NOTIFICATION-TYPE, Integer32,
    Unsigned32, Counter32               FROM SNMPv2-SMI

    TEXTUAL-CONVENTION
                                        FROM SNMPv2-TC

    MODULE-COMPLIANCE, OBJECT-GROUP, NOTIFICATION-GROUP
                                        FROM SNMPv2-CONF

    InetAddressType, InetAddress, InetPortNumber
                                        FROM INET-ADDRESS-MIB
;

--
-- module identity
--

sbsmMIB MODULE-IDENTITY
    LAST-UPDATED "200402150000Z"
    ORGANIZATION "IETF non-existent SBSM Working Group"
    CONTACT-INFO "Wes Hardaker
                  Sparta, Inc.
                  P.O. Box 382
                  Davis, CA  95617
                  Phone: +1 530 792 1913
                  Email: hardaker@tislabs.com"
    DESCRIPTION
     "This MIB module defines objects for managing the SNMPv3 SBSM
```

security module.

Copyright (C) The Internet Society (2004). This version of
this MIB module is part of RFC XXXX, see the RFC itself for
full legal notices."

-- Revision History

```
        REVISION     "200402150000Z"
        DESCRIPTION  "Initial version, published as RFC xxxx."
        -- RFC-editor assigns xxxx


    -- XXX: To be assigned by IANA
        ::= { XXX }


    --
    -- groups of related objects
    --


    sbsmObjects         OBJECT IDENTIFIER
         ::= { sbsmMIB 1 }
    sbsmNotificationObjects   OBJECT IDENTIFIER
         ::= { sbsmMIB 2 }
    sbsmConformanceObjects    OBJECT IDENTIFIER
         ::= { sbsmMIB 3 }


    --
    -- Textual Conventions
    --


    sbsmCounterObjects OBJECT IDENTIFIER ::= { sbsmObjects 1 }
    sbsmSessionObjects OBJECT IDENTIFIER ::= { sbsmObjects 2 }
    sbsmCompressionDefinitions
       OBJECT IDENTIFIER ::= { sbsmObjects 3 }


    --
    -- Counter objects
    --
    sbsmSessionsEstablished OBJECT-TYPE
        SYNTAX      Counter32
        MAX-ACCESS  read-only
        STATUS      current
        DESCRIPTION
            ""
        ::= { sbsmCounterObjects 1}


    sbsmStatsUnsupportedSecLevels OBJECT-TYPE
        SYNTAX      Counter32
        MAX-ACCESS  read-only
```

```
        STATUS      current
        DESCRIPTION
            ""
        ::= { sbsmCounterObjects 2}


    sbsmStatsDecryptionErrors OBJECT-TYPE
        SYNTAX      Counter32
```

```
        MAX-ACCESS  read-only
        STATUS      current
        DESCRIPTION
            ""
        ::= { sbsmCounterObjects 3}


    sbsmStatsCompressionErrors OBJECT-TYPE
        SYNTAX      Counter32
        MAX-ACCESS  read-only
        STATUS      current
        DESCRIPTION
            ""
        ::= { sbsmCounterObjects 4}


    sbsmProtocolError OBJECT-TYPE
        SYNTAX      Counter32
        MAX-ACCESS  read-only
        STATUS      current
        DESCRIPTION
            ""
        ::= { sbsmCounterObjects 5}


    sbsmStatsWrongDigests OBJECT-TYPE
        SYNTAX      Counter32
        MAX-ACCESS  read-only
        STATUS      current
        DESCRIPTION
            ""
        ::= { sbsmCounterObjects 6}


    --
    -- Established sessions
    --


    sbsmSessionTable OBJECT-TYPE
        SYNTAX      SEQUENCE OF SbsmSessionEntry
        MAX-ACCESS  not-accessible
        STATUS      current
        DESCRIPTION
            "A table describing currenly open, currently being established
            or recently closed SBSM sessions."
        ::= { sbsmSessionObjects 1 }
```

```
   sbsmSessionEntry OBJECT-TYPE
       SYNTAX      SbsmSessionEntry
       MAX-ACCESS  not-accessible
       STATUS      current
       DESCRIPTION
```

```
            ""
        INDEX   { sbsmId }
        ::= { sbsmSessionTable 1 }


    SbsmSessionEntry ::= SEQUENCE {
        sbsmId             Unsigned32
    }


    sbsmId OBJECT-TYPE
        SYNTAX      Unsigned32
        MAX-ACCESS  read-only
        STATUS      current
        DESCRIPTION
            ""
        ::= { sbsmSessionEntry 1 }



    -- remote ID, state, alg types in use, started when, misc counters, ...
    -- (suggestions welcome)


    --
    -- Compression algorithms
    --
    sbsmNullCompressionAlgorithm
        OBJECT IDENTIFIER ::= { sbsmCompressionDefinitions 1 }


    sbsmGZipCompressionAlgorithm
        OBJECT IDENTIFIER ::= { sbsmCompressionDefinitions 1 }


    sbsmBZip2CompressionAlgorithm
        OBJECT IDENTIFIER ::= { sbsmCompressionDefinitions 1 }


    --
    -- other MIB items to do:
    --
    -- o notifications
    -- o configuration of policy.  eg: user A using algorthim B/C
    --   is different than user X using Y/Z.
```

```
END
```

8. Identification Mechanisms

overall picture: TBD

8.1 Public Key Based Identities

8.1.1 Security Model assignment

This mechanism defines multiple identity types, all of which are based on public-key mechanisms for authentication. The SNMP security model numbers will be assigned by (IANA). These models include:

- BER encoded signature-based X.509 certificate.
- PGP certificate.
- ssh public key.
- PKIX certificate
- XXX

8.1.2 Format of the identity field

Certificate based identities are identities which are represented by public key based certificates. Multiple types of certificates are defined below, but not all types of certificates may be supported by all implementations.

The identity, when transmitted, will be formated according to the following definition:

```
CertificateSecurityIdentity DEFINITIONS IMPLICIT TAGS ::= BEGIN


  CertificateIdentityInformation ::=
     SEQUENCE {
        certificate-user         OCTET STRING,
        certificate-list         CertificateList
     }
```

```
  CertificateList ::=
     SEQUENCE (SIZE (0..32)) OF OCTET STRING
END
```

Where:

certificateUser: The local account the certificate is expected to be authorized to grant access for.

certificate-list The certificate itself, along with any required supporting certificates (E.G. parent certificates if required), all of which are encoded as dictated by the corresponding identity type (IE, security model number).

8.1.3 Signatures

init-proof1 and resp-proof1 are generated by creating a public key signing and the resulting signature is used as the value for the init-proof1 and resp-proof1 fields.

Checking the value of the init-proof1 and resp-proof1 fields require the following steps:

1. If the security model number must be checked and if it is not a support typed, then a authentication field error MUST be returned.

2. The user name mapping must be a legitimate mapping, as explained in Section Section 8.1.4 below.

3. The value of the signature field should be checked against the expected generated value.

8.1.4 Security Name Mapping

The certificate-user field indicates which securityName the given certificate is expected to access. Legitimate access to this securityName via the given certificate MUST be checked for authorization for the mapping to take place. If the provided certificate is not allowed to "log into" the given securityName account, an authentication error MUST result.

8.2 Local Accounts

When one side of a session wants to perform a traditional login for authentication purposes, this identification mechanism can be used to

achieve that purpose. Note that this mechanism is not recommended since the user's password is transmitted over the wire, although it is encrypted within the session. It is expected that the mechanism will be needed to match current security deployment practices, however. One such example is unix systems which do not have a copy of the user's password and must obtain a copy of it to hash and ensure that the local password database hash matches the incoming password's hash. A better identification mechanism is specified in Section XXX which should be used instead whenever possible.

It is critical for security that this mechanism MUST NOT be used to authenticate a responder to an initiator.

8.2.1 Security Model assignment

This mechanism will be assigned the security number XXX (IANA).

8.2.2 Format of the identity field

The identity, when transmitted, will be formated according to the following definition:

```
LocalAccountSecurityIdentity DEFINITIONS IMPLICIT TAGS ::= BEGIN

  LocalAccountIdentityInformation ::=
     SEQUENCE {
        userName          OCTET STRING,
        passPhrase        OCTET STRING
     }
END
```

8.2.3 Signatures

Generating init-proof1 requires that a signature be generated to sign the protocol values that have been passed over the wire. To do this, the user's passPhrase is converted into a key of an appropriate length by using the authentication algorithm, as negotiated via the authentication-algorithm field of SBSMInit2, as follows:

```
PASSHASH = ALGORITHM_HASH(passPhrase)
KEY = ALGORITHM_HMAC_HASH(PASSHASH,
                          init-engineID | resp-engineID)
```

The KEY is then used to generate a digest using the authentication algorithm and protocol indicated by the authentication-algorithm value. It is potentially truncated according to the authentication protocol specifications of the authentication-algorithm before being inserted into the init-proof1 field of the SBSMInit2Encr portion of

the message.

```
  DIGEST = ALGORITHM_HMAC_HASH(KEY, DATA_TO_DIGEST)
```

XXX: todo change the engineIDs into proper random nonce data.

8.2.4 Security Name Mapping

Mapping a local account user name into a securityName for storage in

the session store and for use in access control is done using a one-to-one mapping. IE, the userName passed in via the IdentityInformation sequence is used directly as the securityName.

8.3 EAP Authentication and Identification

Although not defined yet, the EAP identification mechanism will support a number of important identification concepts missing from the previous mechanisms, such as Generic Token Card, One Time Password, and two factor authentication support.

XXX

8.4 SSH Authentication and Identification

Although not defined yet, the SSH identification mechanism will support identifying users and hosts based on the configured ssh keys for them. It will not make use of SSH itself, just the keys to do authentication and identification. Once a SBSM running session has been established no use of the SSH identity keys will be needed. The SBSM negotiated algorithms and keys will be used for SBSM/SNMPv3 running message integrity.

XXX

9. Compression Algorithms

9.1 sbsmNullCompressionAlgorithm

The sbsmNullCompressionAlgorithm algorithm is a NULL compression algorithm. No compression occurs as a result of this algorithm and the input and output of the algorithm for both compression and decompression are identical. The compression-parameters field of all messages must be set a zero length string.

9.2 sbsmGZipCompressionAlgorithm

The sbsmGZipCompressionAlgorithm algorithm uses the GZip algorithm to

compress the SBSM messages. The compression-parameters field of all messages is unneeded and must be set to a zero length string.

9.3 sbsmBZip2CompressionAlgorithm

The sbsmBZip2CompressionAlgorithm algorithm uses the BZip2 algorithm to compress the SBSM messages. The compression-parameters field of all messages is unneeded and must be set to a zero length string.

10. Security Considerations

This document defines a security protocol to be used within the SNMP framework for providing authentication, integrity, and encryption of SNMP messages. The elements of procedure defined in this document were carefully constructed and must be followed in the proper order to ensure the security properties of the SBSM hold true.

XXX: Write more.

11. TODO list

1. discuss timeout values (don't negotiate, just deny those packets in the future? Doesn't work, necessarily, if managers want to know they have an active open session. But then why not just offer a MIB object instead of burdening the security section with more fields that won't matter)

12. History and Acknowledgments

- Comments from David:

  Back in 1999 after the updates for SNMPv3 where completed for it to be elevated to DRAFT-Standard status, feedback was gathered to determine how SNMPv3 was being used in operational networks. The feedback showed that SNMPv3 was not being as widely deployed as anticipated. Only a few vendors were supporting SNMPv3 agents, and there was little support in management platforms. Also, where it was supported, it was typically not deployed (turned on).

  This resulted in the start of interactions between the operator community and the SNMP community. Over time, it became clear that there was a gap between the needs of the operators and the technology defined by the SNMP community. There were several areas where there were gaps. In the security area, the phrase that was heard over and again was that Radius and SSH were used to manage user authentication for access to managed devices, and SNMPv3/USM created a parallel set of users that had no coordination with the existing security infrastructure.

There were several IETF working groups started to address issues unrelated to security, but none were started to address the security related issues. Thus, the investigation into security issues was uncordinated by the IETF. Several individuals attacked the security issues. Ken Hornstein (?sp) looked at creating a security model using Kerberos security. I (and others) looked at what it would take to use Radius for user authentication. After

some study, I was convinced that directly using Radius "would not work". This was due to two problems. The first was that with USM, the user identity is carried in each message, and to check the user identity with a Radius server would mean that the processing of an SNMPv3 message would be delayed until the Radius server could authenticate a user's identity. The added delay, recovery from dropped traffic to/from the Radius server, and the additional network traffic were judged to be high costs. The second problem was how to support authentication and encryption of SNMP messages using Radius technology. Several approaches were considered, but none were determined workable.

So, if per SNMP operation identity authentication, and then message authentication and optionally message encryption are not feasible using Radius, how could Radius be used? It was at that point that the idea of creating a session where identity authentication was determined, and then creation of session keys for message in the session authentication and optional encryption was determined. This fundamentally different approach to providing security to SNMPv3 had the promise that many different mechanisms could be used to provide identity authentication, and negotiated algorithms could be used for message in the session authentication and encryption. Also, I was convinced that sessions could be created the ran over UDP, and the overhead of the maintained session state was not too costly.

A sketchy description of this session-based security model was put together in June 2001, and shown to a select few during the London IETF in July 2001. The fundamental approach, which is that used in this proposal, was contained in that sketchy proposal. The problem was to work through all of the security details. That is when one after another security experts were asked to provide some assistance.

None volunteered until Wes was convinced that there might be merit in the approach. And it took some time before he could start focusing on the security details. This document has only occurred due to his dedication to the effort in spite of my delays.

- From Wes:

Much of the work in this document is directly derived from the SIGMA protocol [SIGMA]. Specifically, the protocol within this

document is derived from the SIGMA-I variation of the SIGMA protocol. Some of the design decisions made by the IPsec working group surrounding the use of SIGMA in the IKEv2 specification [IKEv2] is also reflected here where appropriate (note that IKEv2 is based on SIGMA-R though). Finally, some of the text in this

document was plagiarized directly from the User Based Security module document [RFC3414].

Discussions Wes has held with the following people over the past few years influenced the contents of this document through either generation of requirements or design ideas:

Michael Baer

Chris Elliot

Eric Fleishman

David Harrington

Ken Hornstein

Sean O'Keeffe

David and I have disagreements about where various ideas within the draft came from and whether they were derived in parallel or not. He has promised me, though, that I have thought of at least one idea in the paper and has thanked me for being an efficient pen.

13. References

13.1 Normative References

[refs.RFC3412]
Case, J., "Message Processing and Dispatching for the Simple Network Management Protocol (SNMP)", RFC 3412, STD 62, December 2002.

[refs.RFC3415]

Wijen, B., Presuhn, R. and K. McCloghrie, "View-based Access Control Model (VACM) for the Simple Network Management Protocol (SNMP)", RFC 3415, STD 62, December 2002.

[refs.RFC3414]
Wijen, B. and U. Blumenthal, "User-based Security Model (USM) for version 3 of the Simple Network Management Protocol (SNMPv3)", RFC 3414, STD 62, December 2002.

13.2 Informative References

[refs.v3overview]
Perkins, D., "An Consolidated Overview of the SNMPv3 Protocol (Internet-Draft)", February 2004.

Authors' Addresses

Wes Hardaker
Sparta
P.O. Box 382
Davis 95617
US

EMail: hardaker@tislabs.com

David T. Perkins
SNMPInfo
548 Quailbrook Ct
San Jose 95110
US

EMail: dperkins@snmpinfo.com

Appendix A. Diffie-Helman Group information

A.1 Diffie-Helman Group IKEv2-N5

This group is represented during negotiations by the OID XXX.

The Diffie-Helman properties to be used for Diffie-Helman calculations using this group is the following. (Note that this is Group 5 from the IKEv2 specification).

The prime is 2^1536 - 2^1472 - 1 + 2^64 * {[2^1406 pi] + 741804}. Its hexadecimal value is

```
FFFFFFFF FFFFFFFF C90FDAA2 2168C234 C4C6628B 80DC1CD1 29024E08
8A67CC74 020BBEA6 3B139B22 514A0879 8E3404DD EF9519B3 CD3A431B
302B0A6D F25F1437 4FE1356D 6D51C245 E485B576 625E7EC6 F44C42E9
A637ED6B 0BFF5CB6 F406B7ED EE386BFB 5A899FA5 AE9F2411 7C4B1FE6
49286651 ECE45B3D C2007CB8 A163BF05 98DA4836 1C55D39A 69163FA8
FD24CF5F 83655D23 DCA3AD96 1C62F356 208552BB 9ED52907 7096966D
670C354E 4ABC9804 F1746C08 CA237327 FFFFFFFF FFFFFFFF
```

The generator is 2.

Intellectual Property Statement

Disclaimer of Validity

Copyright Statement

Acknowledgment

Funding for the RFC Editor function is currently provided by the Internet Society.