The Extended GSS-API Negotiation Mechanism (NEGOEX)
draft-zhu-negoex-01

Abstract

This document defines the Extended Generic Security Service Application Program Interface (GSS-API) Negotiation Mechanism (NegoEx). NegoEx is a pseudo-security mechanism that logically extends the SPNEGO protocol as defined in RFC4178.

The NegoEx protocol itself is a security mechanism negotiated by SPNEGO. When selected as the common mechanism, NegoEx OPTIONALLY adds a pair of meta-data messages for each negotiated security mechanism. The meta-data exchange allows security mechanisms to exchange auxiliary information such as trust configurations, thus NegoEx provides additional flexibility than just exchanging object
 identifiers in SPNEGO.

NegoEx preserves the optimistic token semantics of SPNEGO and applies that recursively. Consequently a context establishment mechanism token can be included in the initial NegoEx message, and NegoEx does not require an extra round-trip when the initiator’s optimistic token is accepted by the target.

Similar to SPNEGO, NegoEx defines a few new GSS-API extensions that a security mechanism MUST support in order to be negotiated by NegoEx. This document defines these GSS-API extensions.

Unlike SPNEGO however, NegoEx defines its own way for signing the protocol messages in order to protect the protocol negotiation. The NegoEx message signing or verification can occur before the security context for the negotiated real security mechanism is fully established.
Table of Contents

1. Introduction .................................................. 4
2. Requirements Terminology .................................... 6
3. Presentation Language and Primitive Data Types ............... 6
   3.1. Basic Block Size ......................................... 7
   3.2. Miscellaneous ........................................... 7
   3.3. Constants ............................................... 7
   3.4. Numbers ................................................. 7
   3.5. Enum Types ............................................. 7
   3.6. Typedef Declarations .................................... 8
   3.7. Array Types ............................................ 8
   3.8. Vector Types ........................................... 8
   3.9. Constructed Types ...................................... 9
4. Cryptographic Computations .................................... 10
5. The NegoEx Protocol ........................................... 11
   5.1. Generation of the Initiator Initial Token ............... 11
   5.2. Receipt of the Initial Initiator Token and Generation
        of the Initial Acceptor Response ....................... 13
   5.3. Receipt of the Acceptor Initial Response and
        Completion of Authentication after the Negotiation
        Phrase .................................................. 14
   5.4. Finalizing Negotiation .................................. 15
   5.5. High-level NegoEx Message Flow ......................... 15
6. Supporting GSS-API Extensions ................................ 16
   6.1. GSS_Query_meta_data ..................................... 16
   6.2. GSS_Exchange_meta_data .................................. 16
   6.3. GSS_Query_mechanism_info ................................ 16
   6.4. GSS_Query_context_attr .................................. 16
7. Security Considerations ....................................... 16
8. Acknowledgements ............................................. 17
9. IANA Considerations .......................................... 17
10. Normative References ........................................ 17
Appendix A. Protocol Data Structures and Constant Values .......... 17
Authors’ Addresses ............................................. 21
Intellectual Property and Copyright Statements ................... 22
1. Introduction

If more than one GSS-API mechanism is shared between the initiator and the acceptor, the Simple and Protected (GSS-API) Negotiation Mechanism (SPNEGO) as defined in [RFC4178] can be deployed to choose a mutually preferred one. This pseudo mechanism does well in the most basic scenarios but suffers from a couple of drawbacks, notably:

- First, the SPNEGO negotiation model is inefficient when negotiating based on mechanism specific configuration information. SPNEGO negotiation is based on exchanging object identifiers only, and it does not allow exchange of auxiliary information in any other form. This is inefficient and often impractical in that one object identifier effectively conveys only one bit of information.

- Secondly, the SPNEGO negotiation model is inadequate when the choice cannot be made by the acceptor in the initial response. In SPNEGO, the negotiation information is sent one-way from the initiator for the acceptor to make a choice, and the acceptor must choose one when it makes the initial response. This negotiation model is counter intuitive. The selection of a security mechanism is typically the result of selecting one type of credentials from the available set, and the initiator typically does not wish to reveal credentials information often associated with user identities. In practice, in order to operate in this model, the Kerberos GSS-API mechanism [RFC4121] must acquire the context establishment token in the initial call to GSS_Init_sec_context(). If the initiator fails to acquire the initial Kerberos GSS-API context token, it must not offer Kerberos; otherwise the SPNEGO context negotiation will fail without being able to select the next available mechanism that could work. Obtaining the initial Kerberos GSS-API context token may require multiple round-trips of network calls and the cost of the operation can be substantial. It is suboptimal when multiple GSS-API mechanisms have to add the extra cost that would not exist if the negotiated security mechanism were selected based on configuration.

The Extended Generic Security Service Application Program Interface (GSS-API) Negotiation Mechanism (NegoEx) is defined to address these concerns. NegoEx is a pseudo security mechanism that is negotiated by SPNEGO, and when negotiated, it can recursively negotiate real security mechanisms.

Any security mechanism negotiated by NegoEx MUST support integrity protection.

The basic form of NegoEx works as follows:
1. The initiator proposes a list of mechanisms in decreasing preference order. For each of these mechanisms, NegoEx optionally includes a mechanism-specific meta-data token. GSS-API extensions are defined later in this document for NegoEx to query the meta-data token for inclusion in the NegoEx message.

2. The acceptor then passes the meta-data token from the initiator to the intended security mechanism. A meta-data token for a security mechanism not supported on the acceptor side is ignored. New GSS-API extensions are defined later in this document for a security mechanism to consume the meta-data token. When processing the received meta-data tokens, a security mechanism that reports a failure is removed from the set of mutually supported mechanisms. The acceptor then responds with the list of mutually supported mechanisms in decreasing preference order. For each of these mechanisms, NegoEx again optionally supplies a mechanism-specific meta-data token in the response. These meta-data tokens are returned to NegoEx via new GSS-API extensions as described in the initial step.

3. The initiator then passes the meta-data tokens to the intended security mechanisms by invoking the new GSS-API extensions. When processing the received meta-data token, a security mechanism that reports a failure is removed from the set of mutually supported mechanisms for this negotiation context. The initiator then selects one from the set of mutually-supported mechanisms. If more than one security mechanism is available, unless otherwise specified, the preferred one in the acceptor’s preference order SHOULD be selected. Once the common security mechanism is identified, the security mechanism may also negotiate mechanism-specific options during its context establishments. This will be inside the mechanism tokens, and invisible to the NegoEx protocol.

4. The selected security mechanism provides keying materials to NegoEx, and NegoEx then signs and verifies the negotiation NegoEx messages to protect the negotiation.

5. The initiator and the acceptor proceed to exchange tokens until the GSS-API context for selected security mechanism is established. Once the security context is established, the per-message tokens are generated and verified in accordance with the selected security mechanism.

NegoEx does not work outside of SPNEGO. When negotiated by SPNEGO, NegoEx uses the concepts developed in the GSS-API specification [RFC2743]. The negotiation data is encapsulated in context-level tokens. Therefore, callers of the GSS-API do not need to be aware of
the existence of the negotiation tokens but only of the SPENGO
pseudo-security mechanism.

In its basic form NegoEx requires at least one extra round-trip.
Network connection setup is a critical performance characteristic of
any network infrastructure and extra round trips over WAN links,
packet radio networks, etc. really make a difference. In order to
avoid such an extra round trip the initial security token of the
preferred mechanism for the initiator may be embedded in the initial
NegoEx token. The optimistic mechanism token may be accompanied by
the meta-data tokens and the optimistic mechanism token MUST be that
of the first mechanism in the list of the mechanisms proposed by the
initiator. The NegoEx message that contains signatures for
protecting the NegoEx negotiation can also be included along with the
mechanism token. If the target preferred mechanism matches the
initiator’s preferred mechanism, and when the NegoEx negotiation
protection messages are included along with the mechanism token, no
additional round trips are incurred by using the NegoEx protocol with
SPNEGO.

NegoEx does not update the ASN.1 structures of SPNEGO in that a large
deployment of SPNEGO does not have the ASN.1 extensibility marker in
the message definition. There is no change to the SPNEGO messages.

NegoEx does not use ASN.1 encoding and it uses simple C structures
encoded in little endian for all its messages.

The rest of the document is organized as follows: Section 3 defines
the encoding of NegoEx data structures and all the primitive data
types. Section 4 describes the cryptographic framework required by
the NegoEx for protecting the NegoEx negotiation. Section 5 defines
the NegoEx messages and the NegoEx protocol. Section 6 defines the
new GSS-API extensions that a security mechanism MUST support in
order to be negotiated by NegoEx. These then are followed by the
security considerations section. Lastly Appendix A contains all the
protocol constructs and constants.

2. Requirements Terminology

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

3. Presentation Language and Primitive Data Types

The following very basic and somewhat casually defined presentation
syntax will be used in all NegoEx messages. Although it resembles the programming language "C" in its syntax, it would be risky to draw too many parallels. The purpose of this presentation language is to document NegoEx only; it has no general application beyond that particular goal.

This section also defines all the primitive data types. The semantics of the data types is explained in the next section.

3.1. Basic Block Size

The representation of all data items is explicitly specified. The basic data block size is one octet. Multiple octet data items are concatenations of octets, from left to right, from top to bottom. Unless otherwise specific a multi-octet numeric is in little endian order with the least significant octet first.

3.2. Miscellaneous

Comments start with "//" and continue until the end of the line.

3.3. Constants

Constants are denoted using "#define" followed by the symbolic name and then the constant value.

3.4. Numbers

UCHAR is the data type for a one-octet number.

ULONG is the data type for a 4-octet number encoded in little endian.

USHORT is the data type for a 2-octet number encoded in little endian.

ULONG64 is the data type for a 8-octet number encoded in little endian.

GUID is the data type for a 16-octet number encoded in little endian.

3.5. Enum Types

An enum type is the data type for a number with a small number of permissible values. An instance of an enum type is a 4-octet number encoded in little endian.

The definition of an enum type follows the simple "C" convention.
MESSAGE_TYPE is an enum type defined as follows:

```c
enum
{
    MESSAGE_TYPE_INITIATOR_NEGO = 0,
    MESSAGE_TYPE_ACCEPTOR_NEGO,
    MESSAGE_TYPE_INITIATOR_META_DATA,
    MESSAGE_TYPE_ACCEPTOR_META_DATA,
    MESSAGE_TYPE_CHALLENGE,
        // an exchange message from the acceptor
    MESSAGE_TYPE_AP_REQUEST,
        // an exchange message from the initiator
    MESSAGE_TYPE_VERIFY,
    MESSAGE_TYPE_ALERT,
} MESSAGE_TYPE;
```

MESSAGE_TYPE_INITIATOR_NEGO has the value 0, and MESSAGE_TYPE_ALERT has the value 7.

3.6. Typedef Declarations

A typedef creates a synonym for the type. This is used to create more meaningful names for existing types.

The following two type synonyms are defined.

```c
typedef GUID AUTH_SCHEME;
typedef GUID CONVERSATION_ID;
```

3.7. Array Types

Arrays are a data structure which holds multiple variables of the same data type consecutively and the number of elements is fixed. An array is declared using "C" convention. For example, the following defines an array of 32 octets.

```c
UCHAR Random[32];
```

3.8. Vector Types

Vectors are a data structure which holds multiple variables of the same data type consecutively and the number of elements is not fixed. A vector contains a fixed length header followed by a variable length payload. The header of a vector structure contains the count of elements and the offset to the payload. In this document all the offset fields start from the beginning of the containing NegoEx message. The size of each element is specified by the vector type definition.
The following vector types are defined.

```c
struct
{
    ULONG ByteArrayOffset; // each element contains an octet/byte
    ULONG ByteArrayLength;
} BYTE_VECTOR;
```

BYTE_VECTOR encapsulates a variable length array of octets (or bytes) that are stored consecutively. Each element is a byte (8 bits).

```c
struct
{
    ULONG AuthSchemeArrayOffset;
    // each element contains an AUTH_SCHEME
    USHORT AuthSchemeCount;
} AUTH_SCHEME_VECTOR;
```

AUTH_SCHEME_VECTOR encapsulates a variable length array of AUTH_SCHEMES that are stored consecutively. Each element is a structure of the type AUTH_SCHEME.

```c
struct
{
    ULONG ExtensionArrayOffset;
    // each element contains an EXTENSION
    USHORT ExtensionCount;
} EXTENSION_VECTOR;
```

EXTENSION_VECTOR encapsulates a variable length array of EXTENSIONS that are stored consecutively. Each element is a structure of the type EXTENSION.

### 3.9. Constructed Types

Structure types may be constructed from primitive types for convenience. Each specification declares a new, unique type. The syntax for definition is much like that of C.

```c
struct {
    T1 f1;
    T2 f2;
    ...
    Tn fn;
} T;
```

Structure definitions may be embedded.
The following types are defined as constructed types:

```c
struct
{
    ULONG ExtensionType; // negative extensions are critical
    BYTE_VECTOR ExtensionValue;
} EXTENSION;
```

An extension has two fields. The ExtensionType field indicates how the extension data should be interpreted. The ExtensionValue field contains the extension data.

```c
//
// schemes defined for the checksum in the VERIFY message
//
struct
{
    ULONG cbHeaderLength;
    ULONG ChecksumScheme;
    ULONG ChecksumType; // in the case of RFC3961 scheme, this is
    // the RFC3961 checksum type
    BYTE_VECTOR ChecksumValue;
} CHECKSUM;
```

The CHECKSUM structure contains 4 fields. The cbHeaderLength length contains the length of the structure definition in octets, and this field has a value of 20.

The ChecksumScheme field describes how checksum is computed and verified. Currently only one value is defined.

```c
#define CHECKSUM_SCHEME_RFC3961 1
```

When the value of the ChecksumScheme field is 1 (CHECKSUM_SCHEME_RFC3961), the ChecksumValue field contains a sequence of octets computed according to [RFC3961] and the ChecksumType field contains the checksum type value defined according to [RFC3961].

4. Cryptographic Computations

The message signing and verification in NegoEx is based on [RFC3961]. [RFC3961] is used here as a generic framework and this application is not Kerberos specific.

A security mechanism MUST support [RFC3961] in order to be negotiated.
5. The NegoEx Protocol

This section describes the NegoEx protocol and it defines NegoEx messages in the order that the messages can appear on the wire. The enum type MESSAGE_TYPE defined in Section 3.5 lists all NegoEx message types. A GSS-API context token for NegoEx consists of one or more NegoEx messages. If there are more than one NegoEx message, these messages are concatenated together. The smallest data unit for NegoEx to compute the checksum for negotiation protection is a NegoEx message. Note that NegoEx is not a GSS-API mechanism itself and the initial NegoEx context establishment token does not follow the mechanism-independent token format defined in Section 3.1 of [RFC2743].

A security mechanism negotiated by NegoEx is identified by a unique identifier of the data type AUTH_SCHEME defined in Section 3.5. The value of the security mechanism is returned to NegoEx via the GSS_Query_mechanism_info() GSS-API extension as defined in Section 6.

The object identifier of the NegoEx within SPNEGO is iso(1) identified-organization(3) dod(6) internet(1) private(4) enterprise(1) microsoft (311) security(2) mechanisms(2) negoex(30). Note that NegoEx does not work outside of SPNEGO and it is not GSS-API mechanism.

5.1. Generation of the Initiator Initial Token

The GSS-API initiator makes the first call to GSS_Init_sec_context() with no input token, and the output token starts as a NEGO_MESSAGE message with the MESSAGE_TYPE_INITIATOR_NEGO message type.
struct
{
    ULONG64 Signature; // contains MESSAGE_SIGNATURE
    MESSAGE_TYPE MessageType;
    ULONG SequenceNum; // the message sequence number of this,
    // conversation, starting with 0 and sequentially
    // incremented
    ULONG cbHeaderLength; // the header length of this message,
    // including the message specific header, excluding the
    // payload
    ULONG cbMessageLength; // the length of this message
    CONVERSATION_ID ConversationId;
} MESSAGE_HEADER;

struct
{
    MESSAGE_HEADER Header;
    // MESSAGE_TYPE_INITIATOR_NEGO for the initiator,
    // MESSAGE_TYPE_ACCEPTOR_NEGO for the acceptor
    UCHAR Random[32];
    ULONG64 ProtocolVersion;
    // version of the protocol, this contains 0
    AUTH_SCHEME_VECTOR AuthSchemes;
    EXTENSION_VECTOR Extensions;
} NEGO_MESSAGE;

The initiator randomly generates a ConversationID and fills the
common header. The ConversationID in subsequent NegoEx messages MUST
remain the same. The initiator also fills the Random field using a
secure random number generator. The initiator fills the AuthSchemes
with available security mechanisms supported by the initiator in
decreasing preference order.

The extensions field contains NegoEx extensions for future
extensibility. There is no extension defined in this document. All
negative extension types (the highest bit is set to 1) are critical.
If the receiver does not understand a critical extension, the
authentication attempt must be rejected.

The initiator can OPTIONALLY include a meta-data token, one for each
available security mechanism.

A meta-data token is returned to NegoEx for a security mechanism
using GSS_Query_meta_data() extension as defined in Section 6. A
meta-data token is encapsulated in an EXCHANGE message with the
message type MESSAGE_TYPE_INITIATOR_META_DATA.
struct
{
    MESSAGE_HEADER Header;
    // MESSAGE_TYPE_CHALLENGE for the acceptor,
    // or MESSAGE_TYPE_AP_REQUEST for the initiator
    // MESSAGE_TYPE_INITIATOR_META_DATA for
    // the initiator metadata
    // MESSAGE_TYPE_ACCEPTOR_META_DATA for
    // the acceptor metadata
    AUTH_SCHEME AuthScheme;
    BYTE_VECTOR Exchange;
    // contains the opaque handshake message for the
    // authentication scheme
} EXCHANGE_MESSAGE;

The AuthScheme field signifies the security mechanism for which the
EXCHANGE message is targeted. If a security mechanism fails to
produce the metadata token, it should be removed from the list of
supported security mechanism for this negotiation context.

If there are more than one exchange messages, the order in which the
exchange message is included bears no significance. In other words,
the exchange messages are in an unordered set. The NEGO_MESSAGE MAY
be followed by a set of MESSAGE_TYPE_INITIATOR_META_DATA messages as
described above, in which case all the NegoEx messages concatenated
are returned as a single input token.

The first mechanism in the initiator proposed list can OPTIONALLY
include its initial context context in an AP_REQUEST message.

Both an AP_REQUEST(short for MESSAGE_TYPE_AP_REQUEST) message and a
INITIATOR_META_DATA(short for MESSAGE_TYPE_INITIATOR_META_DATA)
message are instances of the EXCHANGE_MESSAGE structure with
different message type values. An AP_REQUEST message contains the
type MESSAGE_TYPE_AP_REQUEST while an INITIATOR_META_DATA message
contains the type MESSAGE_TYPE_INITIATOR_META_DATA.

5.2. Receipt of the Initial Initiator Token and Generation of the
Initial Acceptor Response

Upon receipt of the NEGO_MESSAGE from the initiator, the acceptor
verifies the NEGO_MESSAGE to make sure it is well-formed. The
acceptor then computes the list of authentication schemes that are
mutually supported by examining the set of security mechanisms
proposed by the initiator and the meta-data tokens from the
initiator. The meta-data tokens are passed to the security mechanism
via GSS_Exchange_meta_data() as defined in Section 6.
The acceptor MUST examine the NegoEx extensions in the NEGO_MESSAGE. If there is an unknown critical extension, the authentication must be rejected.

The acceptor’s response starts as a NEGO_MESSAGE but with the MESSAGE_TYPE_ACCEPTOR_NEGO. The AuthSchemes field contains the list of mutually supported security mechanism in decreasing preference order of the acceptor. The acceptor does not need to honor the preference order proposed by the initiator when computing its preference list.

The acceptor can OPTIONALLY include a meta-data token, one for each available security mechanism.

A meta-data token is returned to NegoEx for a security mechanism using GSS_Query_meta_data() extension as defined in Section 6. A meta-data token is encapsulated in an EXCHANGE message with the message type MESSAGE_TYPE_ACCEPTOR_META_DATA. For a given security mechanism if a meta-token is received from the initiator, GSS_Query_meta_data() MUST be invoked on the acceptor side for that security mechanism, and the output meta-data token, if present, MUST be included in the NegoEx reply.

5.3. Receipt of the Acceptor Initial Response and Completion of Authentication after the Negotiation Phrase

Upon receipt of the initial response from the acceptor, the initial verifies the NEGO_MESSAGE to make sure it is well-formed. The initiator then computes the list of authentication schemes that are mutually supported by examining the set of security mechanisms returned by the acceptor and the meta-data tokens from the acceptor. The meta-data tokens are passed to the security mechanism via GSS_Exchange_meta_data() as defined in Section 6.

The initiator MUST examine the NegoEx extensions in the NEGO_MESSAGE. If there is an unknown critical extension, the authentication must be rejected.

After the initial exchange of NEGO_MESSAGE messages, the initiator MUST choose the negotiated security mechanism. The negotiated security mechanism cannot be changed once it is selected.

The initiator and the acceptor can then proceed to exchange handshake messages as determined by the negotiated security mechanism until its authentication context is established. The context tokens of the negotiated security mechanism are encapsulated in an EXCHANGE_MESSAGE. If the context token is from the initiator, the EXCHANGE_MESSAGE message has the message type.
MESSAGE_TYPE_AP_REQUEST; otherwise, the message type is MESSAGE_TYPE_CHALLENGE.

5.4. Finalizing Negotiation

Whenever there is a shared key established returned by GSS_Query_context_attr(NEGOEX_SESSION_KEYS) as defined in Section 6,, a VERIFY message is produced and included in the output token. The returned protocol key is used as the base key in the parlance of RFC3961 to sign all the NegoEx messages in the negotiation context.

A VERIFY message is a VERIFY_MESSAGE structure. The AuthScheme field signifies from which security mechanism the protocol key was obtained. The checksum is computed based on RFC3961 and the key usage number is 23 for the message is signed by the initiator, 25 otherwise. The checksum is performed over all the previous NegoEx messages in the context negotiation.

```c
struct
{
    MESSAGE_HEADER Header; // MESSAGE_TYPE_VERIFY
    AUTH_SCHEME AuthScheme;
    CHECKSUM Checksum;
    // contains the checksum of all the previously exchanged messages in the order they were sent.
} VERIFY_MESSAGE;
```

Note that the VERIFY_MESSAGE message can be included before the security context for the negotiated security mechanism is fully established.

5.5. High-level NegoEx Message Flow

The following text art summarizes the protocol message flow:
6. Supporting GSS-API Extensions

This section defined all the required GSS-API extensions required by NegoEx.

6.1.  GSS_Query_meta_data

TBD.

6.2.  GSS_Exchange_meta_data

TBD.

6.3.  GSS_Query_mechanism_info

TBD.

6.4.  GSS_Query_context_attr

TBD.

7. Security Considerations

TBD.
8. Acknowledgements

TBD.

9. IANA Considerations

There is no action required for IANA.

10. Normative References


Appendix A. Protocol Data Structures and Constant Values

This section complies all the protocol data structures and constant values.

```c
#define MESSAGE_SIGNATURE   0x535458454f47454ei64
// "NEGOEXTS"

struct
{
    ULONG ByteArrayOffset; // each element contains a byte
    ULONG ByteArrayLength;

```
struct
{
    ULONG AuthSchemeArrayOffset;
    // each element contains an AUTH_SCHEME
    USHORT AuthSchemeCount;
} AUTH_SCHEME_VECTOR;

struct
{
    ULONG ExtensionArrayOffset;
    // each element contains an EXTENSION
    USHORT ExtensionCount;
} EXTENSION_VECTOR;

struct
{
    ULONG ExtensionType; // negative extensions are critical
    BYTE_VECTOR ExtensionValue;
} EXTENSION;

// schemes defined for the checksum in the VERIFY message
#
#define CHECKSUM_SCHEME_RFC3961 1

struct
{
    ULONG cbHeaderLength;
    ULONG ChecksumScheme;
    ULONG ChecksumType; // in the case of RFC3961 scheme, this is
    // the RFC3961 checksum type
    BYTE_VECTOR ChecksumValue;
} CHECKSUM;

typedef GUID AUTH_SCHEME;
typedef GUID CONVERSATION_ID;

enum
{
    MESSAGE_TYPE_INITIATOR_NEGO = 0,
    MESSAGE_TYPE_ACCEPTOR_NEGO,
    MESSAGE_TYPE_INITIATOR_META_DATA,
    MESSAGE_TYPE_ACCEPTOR_META_DATA,
    MESSAGE_TYPE_CHALLENGE,
    // an exchange message from the acceptor
MESSAGE_TYPE_AP_REQUEST,
    // an exchange message from the initiator
MESSAGE_TYPE_VERIFY,
MESSAGE_TYPE_ALERT,
} MESSAGE_TYPE;

struct
{
    ULONG64 Signature; // contains MESSAGE_SIGNATURE
    MESSAGE_TYPE MessageType;
    ULONG SequenceNum; // the message sequence number of this,
        // conversation, starting with 0 and sequentially
        // incremented
    ULONG cbHeaderLength; // the header length of this message,
        // including the message specific header, excluding the
        // payload
    ULONG cbMessageLength; // the length of this message
    CONVERSATION_ID ConversationId;
} MESSAGE_HEADER;

struct
{
    MESSAGE_HEADER Header;
    // MESSAGE_TYPE_INITIATOR_NEGO for the initiator,
    // MESSAGE_TYPE_ACCEPTOR_NEGO for the acceptor
    UCHAR Random[32];
    ULONG64 ProtocolVersion;
    // version of the protocol, this contains 0
    AUTH_SCHEME_VECTOR AuthSchemes;
    EXTENSION_VECTOR Extensions;
} NEGO_MESSAGE;

struct
{
    MESSAGE_HEADER Header;
    // MESSAGE_TYPE_CHALLENGE for the acceptor,
    // or MESSAGE_TYPE_AP_REQUEST for the initiator
    // MESSAGE_TYPE_INITIATOR_METADATA for
    // the initiator metadata
    // MESSAGE_TYPE_ACCEPTOR_METADATA for
    // the acceptor metadata
    AUTH_SCHEME AuthScheme;
    BYTE_VECTOR Exchange;
        // contains the opaque handshake message for the
        // authentication scheme
} EXCHANGE_MESSAGE;

struct
{ MESSAGE_HEADER Header; // MESSAGE_TYPE_VERIFY
  AUTH_SCHEME AuthScheme;
  CHECKSUM Checksum;
      // contains the checksum of all the previously
      // exchanged messages in the order they were sent.
} VERIFY_MESSAGE;

struct
{
    ULONG AlertType;
    BYTE_VECTOR AlertValue;
} ALERT;

// alert types

#define ALERT_TYPE_PULSE             1

// reason codes for the heartbeat message

#define ALERT_VERIFY_NO_KEY          1

struct
{
    ULONG cbHeaderLength;
    ULONG Reason;
} ALERT_PULSE;

struct
{
    ULONG AlertArrayOffset; // the element is an ALERT
    USHORT AlertCount; // contains the number of alerts
} ALERT_VECTOR;

struct
{
    MESSAGE_HEADER Header;
    AUTH_SCHEME AuthScheme;
    ULONG ErrorCode; // an NTSTATUS code
    ALERT_VECTOR Alerts;
} ALERT_MESSAGE;
Authors’ Addresses

Larry Zhu
Microsoft Corporation
One Microsoft Way
Redmond, WA  98052
US

Email: lzhu@microsoft.com

Kevin Damour
Microsoft Corporation
One Microsoft Way
Redmond, WA  98052
US

Email: kdamour@microsoft.com

Dave McPherson
Microsoft Corporation
One Microsoft Way
Redmond, WA  98052
US

Email: davemm@microsoft.com
Full Copyright Statement

Copyright (C) The IETF Trust (2008).

This document is subject to the rights, licenses and restrictions contained in BCP 78, and except as set forth therein, the authors retain all their rights.

This document and the information contained herein are provided on an "AS IS" basis and THE CONTRIBUTOR, THE ORGANIZATION HE/SHE REPRESENTS OR IS SPONSORED BY (IF ANY), THE INTERNET SOCIETY, THE IETF TRUST AND THE INTERNET ENGINEERING TASK FORCE DISCLAIM ALL WARRANTIES, EXPRESS OR IMPLIED, INCLUDING BUT NOT LIMITED TO ANY WARRANTY THAT THE USE OF THE INFORMATION HEREIN WILL NOT INFRINGE ANY RIGHTS OR ANY IMPLIED WARRANTIES OF MERCHANTABILITY OR FITNESS FOR A PARTICULAR PURPOSE.

Intellectual Property

The IETF takes no position regarding the validity or scope of any Intellectual Property Rights or other rights that might be claimed to pertain to the implementation or use of the technology described in this document or the extent to which any license under such rights might or might not be available; nor does it represent that it has made any independent effort to identify any such rights. Information on the procedures with respect to rights in RFC documents can be found in BCP 78 and BCP 79.

Copies of IPR disclosures made to the IETF Secretariat and any assurances of licenses to be made available, or the result of an attempt made to obtain a general license or permission for the use of such proprietary rights by implementers or users of this specification can be obtained from the IETF on-line IPR repository at http://www.ietf.org/ipr.

The IETF invites any interested party to bring to its attention any copyrights, patents or patent applications, or other proprietary rights that may cover technology that may be required to implement this standard. Please address the information to the IETF at ietf-ipr@ietf.org.