

[MS-NLMP]: NT LAN Manager (NTLM) Authentication Protocol Specification

Intellectual Property Rights Notice for Protocol Documentation

- This protocol documentation is covered by Microsoft copyrights. Regardless of any other terms that are contained in the terms of use for the Microsoft website that hosts this documentation, you may make copies of it in order to develop implementations of the protocols, and may distribute portions of it in your implementations of the protocols or your documentation as necessary to properly document the implementation. This permission also applies to any documents that are referenced in the protocol documentation.
- Microsoft does not claim any trade secret rights in this documentation.
- Microsoft has patents that may cover your implementations of the protocols. Neither this notice nor Microsoft's delivery of the documentation grants any licenses under those or any other Microsoft patents. If you are interested in obtaining a patent license, please contact protocol@microsoft.com.
- The names of companies and products contained in this documentation may be covered by trademarks or similar intellectual property rights. This notice does not grant any licenses under those rights.
- All other rights are reserved, and this notice does not grant any rights other than specifically described above, whether by implication, estoppel, or otherwise.

This protocol documentation is intended for use in conjunction with publicly available standard specifications, network programming art, and Microsoft Windows distributed systems concepts, and assumes that the reader either is familiar with the aforementioned material or has immediate access to it.

A protocol specification does not require the use of Microsoft programming tools or programming environments in order for you to develop an implementation. If you have access to Microsoft programming tools and environments you are free to take advantage of them.

Revision Summary

Date	Revision History	Revision Class	Comments
02/22/2007	0.01		MCPP Milestone 3 Initial Availability
06/01/2007	1.0	Major	Updated and revised the technical content.
07/03/2007	1.0.1	Editorial	Revised and edited the technical content.
07/20/2007	2.0	Major	Updated and revised the technical content.

Date	Revision History	Revision Class	Comments
08/10/2007	3.0	Major	Updated and revised the technical content.
09/28/2007	4.0	Major	Updated and revised the technical content.
10/23/2007	5.0	Major	Updated and revised the technical content.
11/30/2007	6.0	Major	Updated and revised the technical content.
01/25/2008	6.0.1	Editorial	Revised and edited the technical content.

Table of Contents

1	Introduction	5
1.1	Glossary	5
1.2	References	6
1.2.1	Normative References	6
1.2.2	Informative References.....	7
1.3	Protocol Overview (Synopsis).....	7
1.3.1	NTLM Authentication Call Flow	8
1.3.1.1	NTLM Connection-Oriented Call Flow	9
1.3.1.2	NTLM Connectionless (Datagram-Oriented) Call Flow	10
1.4	Relationship to Other Protocols.....	11
1.5	Prerequisites/Preconditions.....	11
1.6	Applicability Statement	11
1.7	Versioning and Capability Negotiation.....	12
1.8	Vendor-Extensible Fields	12
1.9	Standards Assignments.....	12
2	Messages	13
2.1	Transport.....	13
2.2	Message Syntax	13
2.2.1	NTLM Messages	15
2.2.1.1	NEGOTIATE_MESSAGE	15
2.2.1.2	CHALLENGE_MESSAGE	17
2.2.1.3	AUTHENTICATE_MESSAGE.....	20
2.2.1.4	NTLMSSP_MESSAGE_SIGNATURE	26
2.2.2	NTLM Structures.....	26
2.2.2.1	AV_PAIR	26
2.2.2.2	Restriction Encoding	28
2.2.2.3	LM_RESPONSE.....	29
2.2.2.4	LMv2_RESPONSE	29
2.2.2.5	NEGOTIATE	30
2.2.2.6	NTLM v1 Response: NTLM_RESPONSE	33
2.2.2.7	NTLM v2: NTLMv2_CLIENT_CHALLENGE	33
2.2.2.8	NTLM2 V2 Response: NTLMv2_RESPONSE.....	35
2.2.2.9	VERSION	35
3	Protocol Details	37
3.1	Client Details	37
3.1.1	Abstract Data Model	37
3.1.1.1	Variables Internal to the Protocol.....	37
3.1.1.2	Variables Exposed to the Application	38
3.1.2	Timers	38
3.1.3	Initialization.....	39
3.1.4	Higher-Layer Triggered Events.....	39
3.1.5	Message Processing Events and Sequencing Rules	39
3.1.5.1	Connection-Oriented	40
3.1.5.1.1	Client Initiates the NEGOTIATE_MESSAGE Packet	40
3.1.5.1.2	Client Receives a CHALLENGE_MESSAGE from the Server	40
3.1.5.2	Connectionless.....	42
3.1.5.2.1	Client Receives a CHALLENGE_MESSAGE	42
3.1.5.2.2	Server Receives an AUTHENTICATE_MESSAGE	42
3.1.6	Timer Events.....	43
3.1.7	Other Local Events.....	43

3.2	Server Details.....	43
3.2.1	Abstract Data Model.....	43
3.2.1.1	Variables Internal to the Protocol.....	43
3.2.1.2	Variables Exposed to the Application	43
3.2.2	Timers	43
3.2.3	Initialization.....	43
3.2.4	Higher-Layer Triggered Events.....	43
3.2.5	Message Processing Events and Sequencing Rules	44
3.2.5.1	Connection-Oriented	44
3.2.5.1.1	Server Receives a NEGOTIATE_MESSAGE from the Client	45
3.2.5.1.2	Server Receives an AUTHENTICATE_MESSAGE from the Client	46
3.2.5.2	Connectionless NTLM	48
3.2.5.2.1	Server Sends the Client an Initial CHALLENGE_MESSAGE Packet	48
3.2.5.2.2	Server Response Checking	48
3.2.6	Timer Events.....	48
3.2.7	Other Local Events	48
3.3	NTLM v1 and NTLM v2 Messages	48
3.3.1	NTLM v1 Authentication	48
3.3.2	NTLM v2 Authentication	50
3.4	Session Security Details	51
3.4.1	Abstract Data Model.....	52
3.4.2	Message Integrity	52
3.4.3	Message Confidentiality	53
3.4.4	Message Signature Functions.....	54
3.4.5	KXKEY, SIGNKEY, and SEALKEY	55
3.4.6	SKXKEY.....	55
3.4.7	SIGNKEY	56
3.4.8	SEALKEY	56
3.4.9	GSS_WrapEx() Call	57
3.4.9.1	Signature Creation for GSS_WrapEx()	58
3.4.10	GSS_UnwrapEx() Call.....	58
3.4.11	GSS_GetMICEx() Call	59
3.4.11.1	Signature Creation for GSS_GetMICEx()	59
3.4.12	GSS_VerifyMICEx() Call.....	59
4	Protocol Examples	61
5	Security	63
5.1	Security Considerations for Implementers.....	63
5.2	Index of Security Parameters	63
6	Appendix A: Cryptographic Operations Reference	64
7	Appendix B: Windows Behavior	67
8	Index.....	71

1 Introduction

The NT LAN Manager (NTLM) Authentication Protocol is used in Windows for authentication between clients and servers. For Windows 2000 Server, Windows XP, Windows Server 2003, Windows Vista and Windows Server 2008, Kerberos replaces NTLM as the preferred authentication protocol. However, NTLM is still used in cases where Kerberos does not work (for example, if one of the machines is not Kerberos-capable, if the server is not joined to a **domain**, or if the Kerberos configuration is not set up correctly).

1.1 Glossary

The following terms are defined in [\[MS-GLOS\]](#):

Active Directory (AD)
Checksum
Connection-Oriented RPC
Directory
Domain
Domain Controller (DC)
Domain Name
Forest
Guest Account
Kerberos
Key
Message Authentication Code (MAC)
Nonce
Original Equipment Manufacturer (OEM) Character Set
Remote Procedure Call (RPC)
Security Support Provider Interface (SSPI)
Service
Session
Session Key
Share
Unicode

The following terms are specific to this document:

AV Pair: A term for "attribute/value pair." An attribute/value pair is the name of some attribute, along with its value. AV pairs in NTLM have a structure specifying the encoding of the information stored in them.

Challenge: A piece of data used to authenticate a user. A **challenge** typically takes the form of a **nonce**.

Codepage: The traditional IBM term used for a specific character encoding table: a mapping in which a sequence of bits, usually a single octet representing integer values 0 through 255, is associated with a specific character. IBM and Microsoft often allocate a code page number to a charset even if that charset is better known by another name. Although the term code page originated from IBM's EBCDIC-based mainframe systems, the term is most commonly associated with the IBM PC code pages. Microsoft, a maker of PC operating systems, refers to these code pages as OEM code pages, and supplements them with its own "ANSI" code pages.

Connection Oriented NTLM: A particular variant of NTLM designed to be used with connection oriented RPC.

Cyclic Redundancy Check (CRC): An algorithm used to produce a checksum (that is, a small, fixed number of bits) against a block of data, such as a packet of network traffic or a block of a computer file. The **CRC** is used to detect errors after transmission or storage. A **CRC** is designed to catch stochastic errors, as opposed to intentional errors. If errors might be introduced by a motivated and intelligent adversary, a cryptographic hash function should be used instead.

FILETIME: The date and time as a 64-bit value in little-endian order representing the number of 100-nanosecond intervals elapsed since January 1, 1601.

Forest Tree Name: A **forest tree name** is the first **domain name** in a Microsoft **Active Directory forest** when the **forest** was created.

Identify Level Token: A security token resulting from authentication that represents the authenticated user but does not allow the **service** holding the token to impersonate that user to other resources.

Key Exchange Key: The **key** used to protect the **session key** that is generated by the client. The **key exchange key** is derived from the **response key** during authentication.

LMOWF: A one-way function used to generate a **key** based on the user's password. The term is also used to refer to the **key** generated by the function.

NTOWF: A one-way function (similar to the **LMOWF** function) used to generate a **key** based on the user's password. The term is also used to refer to the **key** generated by the function.

Response Key: A **key** computed via a one-way function of the password. The function depends on what NTLM version is being used. The **response key** is used to derive the **key exchange key**.

Session Security: The provision of message integrity and/or confidentiality to a **session key**.

MAY, SHOULD, MUST, SHOULD NOT, MUST NOT: These terms (in all caps) are used as described in [\[RFC2119\]](#). All statements of optional behavior use either MAY, SHOULD, or SHOULD NOT.

1.2 References

1.2.1 Normative References

We conduct frequent surveys of the normative references to assure their continued availability. If you have any issue with finding a normative reference, please contact dochelp@microsoft.com. We will assist you in finding the relevant information. Please check the archive site, <http://msdn2.microsoft.com/en-us/library/E4BD6494-06AD-4aed-9823-445E921C9624>, as an additional source.

[FIPS46-2] National Institute of Standards and Technology, "Federal Information Processing Standards Publication 46-2: Data Encryption Standard (DES)", December 1993, <http://www.itl.nist.gov/fipspubs/fip46-2.htm>

[MS-APDS] Microsoft Corporation, "[Authentication Protocol Domain Support Specification](#)", June 2007.

[MS-DTYP] Microsoft Corporation, "[Windows Data Types](#)", January 2007.

[MS-GLOS] Microsoft Corporation, "[Windows Protocols Master Glossary](#)", March 2007.

- [MS-RPCE] Microsoft Corporation, "[Remote Procedure Call Protocol Extensions](#)", January 2007.
- [MS-SMB] Microsoft Corporation, "[Server Message Block \(SMB\) Protocol Specification](#)", July 2007.
- [MS-SPNG] Microsoft Corporation, "[Simple and Protected Generic Security Service Application Program Interface Negotiation Mechanism \(SPNEGO\) Protocol Extensions](#)", January 2007.
- [MSDN-DecryptMsg] Microsoft Corporation, "DecryptMessage (General) Function", <http://msdn2.microsoft.com/en-us/library/aa375211.aspx>
- [MSDN-EncryptMsg] Microsoft Corporation, "EncryptMessage (General)", <http://msdn2.microsoft.com/en-us/library/aa375378.aspx>
- [RFC1320] Rivest, R. "The MD4 Message-Digest Algorithm", RFC 1320, April 1992, <http://www.ietf.org/rfc/rfc1320.txt>
- [RFC1321] Rivest, R. "The MD5 Message-Digest Algorithm", RFC 1321, April 1992, <http://www.ietf.org/rfc/rfc1321.txt>
- [RFC1964] Linn, J., "The Kerberos Version 5 GSS-API Mechanism", RFC 1964, June 1996, <http://www.ietf.org/rfc/rfc1964.txt>
- [RFC2104] Krawczyk, H., Bellare, M., and Canetti, R., "HMAC: Keyed-Hashing for Message Authentication", RFC 2104, February 1997, <http://www.ietf.org/rfc/rfc2104.txt>
- [RFC2119] Bradner, S., "Key words for use in RFCs to Indicate Requirement Levels", BCP 14, RFC 2119, March 1997, <http://www.ietf.org/rfc/rfc2119.txt>
- [RFC2743] Linn, J., "Generic Security Service Application Program Interface Version 2, Update 1", RFC 2743, January 2000, <http://www.ietf.org/rfc/rfc2743.txt>
- [RFC4757] Jaganathan, K., Zhu, L., and Brezak, J., "The RC4-HMAC Kerberos Encryption Types Used by Microsoft Windows", RFC 4757, December 2006, <http://www.ietf.org/rfc/rfc4757.txt>
- [SCHNEIER] Schneier, B., "Applied Cryptography, Second Edition", John Wiley and Sons, 1996, ISBN: 0471117099.

If you have any trouble finding [SCHNEIER], please check [here](#).

1.2.2 Informative References

- [MSDN-ENCRYPTMESSAGE] Microsoft Corporation, "EncryptMessage (General)", <http://msdn2.microsoft.com/en-us/library/aa375378.aspx>
- [MSDN-SSPIKIG] Microsoft Corporation, "SSPI/Kerberos Interoperability with GSSAPI", <http://msdn2.microsoft.com/en-us/library/ms995352.aspx>

1.3 Protocol Overview (Synopsis)

NT LAN Manager (NTLM) is the name of a family of security protocols in Windows. NTLM is used by application protocols to authenticate remote users and, optionally, to provide **session security** when requested by the application.

NTLM is a **challenge**-response style authentication protocol. This means that to authenticate a user, the server sends a challenge to the client. The client then sends back a response that is a function of the challenge, the user's password, and possibly other information. Computing the correct response requires knowledge of the user's password. The server (or another party trusted by the server) can

validate the response by consulting an account database to get the user's password and computing the proper response for that challenge.

The NTLM protocols are embedded protocols. Unlike stand-alone application protocols such as [\[MS-SMB\]](#) or HTTP, NTLM packets are embedded in the packets of an application protocol that needs to authenticate a user. The application protocol semantics determine how and when the NTLM packets are encoded, framed, and transported from the client to the server and vice versa. See section [4](#) for an example of how NTLM messages are embedded in [\[MS-SMB\]](#). The NTLM implementation also differs from normal protocol implementations, in that the best way to implement it is as a function library called by some other protocol implementation (the application protocol), rather than as a layer in a network protocol stack. For more information about GSS-API calls, see section [3.4.9](#). The NTLM function library receives parameters from the application protocol caller and returns packet fragments that the caller places into fields of its own messages as it chooses. Nevertheless, if one looks at just the NTLM packet elements apart from the application protocol in which they are embedded, there is an NTLM protocol and that is what is specified by this document.

There are two major variants of the NTLM authentication protocol: the "**connection-oriented**" variant and the "connectionless" variant (the names are historic and reflect the kind of RPC protocol that was expected to be the primary consumer of the variant). In the connectionless (datagram) variant:

- NTLM does not use the internal sequence number maintained by the NTLM implementation. Instead, it uses a sequence number passed in by the protocol implementation in which NTLM is embedded.
- **Keys** for session security are established at client initialization time (while in connection-oriented mode they are established only at the end of authentication exchange), and session security can be used as soon as the **session keys** are established.
- It is not possible to send a NEGOTIATE message (see section [2.2.1.1](#)).

Each of these variants has two versions: NTLMv1, and NTLMv2. The message flow for NTLMv1 and NTLMv2 is the same; the only difference is the function used to compute the response from the challenge.

In addition to authentication, the NTLM protocol optionally provides for session security—specifically message integrity and confidentiality through signing and sealing functions in NTLM.

1.3.1 NTLM Authentication Call Flow

This section provides an overview of the end-to-end message flow when application protocols use NTLM to authenticate a user to a server.

The following diagram shows a typical connection-oriented message flow when an application uses NTLM. The message flow typically consists of a number of application messages, followed by NTLM authentication messages, (which are embedded in the application protocol and transported by the application from the client to the server) and then additional application messages as specified in the application protocol.

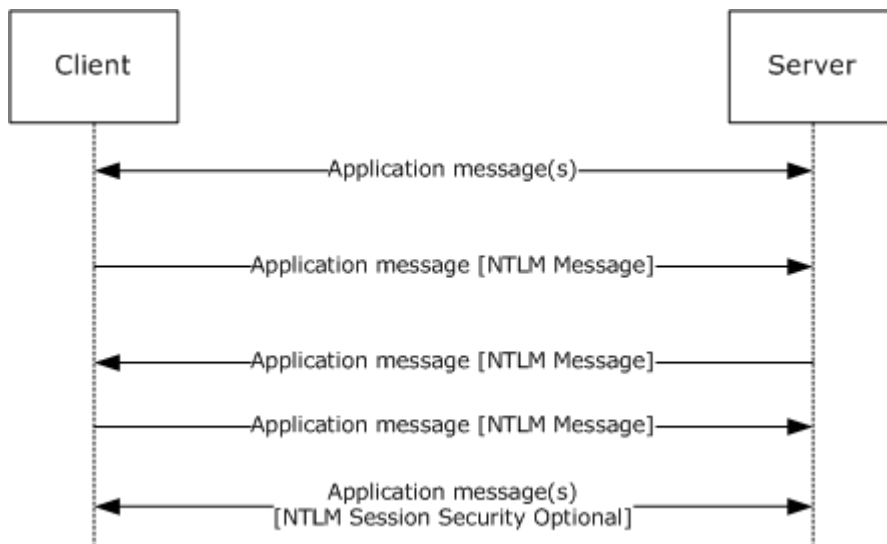


Figure 1: Typical NTLM authentication message flow

Note In the diagram above, the embedding of NTLM messages in the application protocol is shown by placing the NTLM messages within [] brackets. NTLM messages for both connection-oriented and datagram-oriented authentication are embedded in the application protocol as shown. Variations between the connection-oriented and datagram-oriented NTLM protocol sequence are documented in sections [1.3.1.1](#) and [1.3.1.2](#).

After an authenticated NTLM **session** is established, the subsequent application messages may optionally be protected with NTLM session security. This is done by the application, which specifies what options (such as message integrity or confidentiality, as specified in the Abstract Data Model) it needs, before the NTLM authentication packet sequence begins. [<1>](#)

Success and failure messages that are sent after the NTLM authentication packet sequence are specific to the application protocol invoking NTLM authentication and are not part of the NTLM Authentication Protocol.

Note In subsequent message flows, only the NTLM message flows are shown since they are the focus of this document. Keep in mind that the NTLM messages in this section are embedded in the application protocol and transported by that protocol.

Below is an overview of the connection-oriented and connectionless variants of NTLM.

1.3.1.1 NTLM Connection-Oriented Call Flow

The following illustration shows a typical NTLM connection-oriented call flow when an application protocol creates an authenticated session. For detailed message specifications, see section [2](#). The messages are processed as specified in section [3](#).

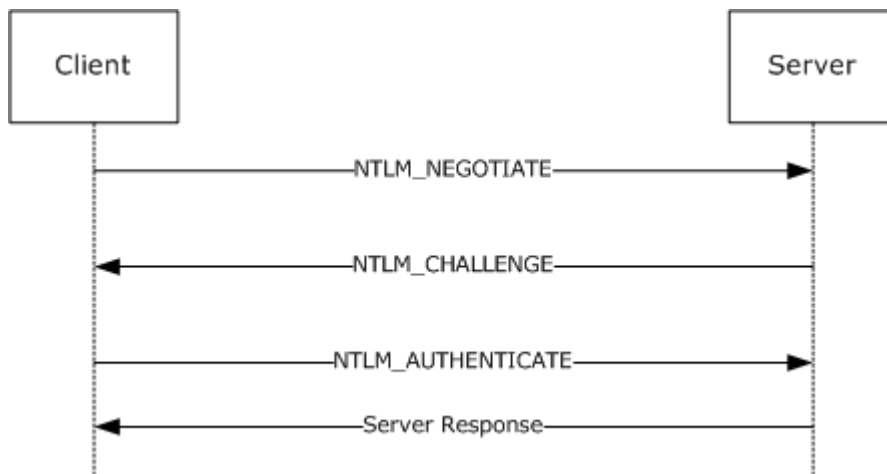


Figure 2: Connection-Oriented NTLM Message Flow

1. The client sends an NTLM NEGOTIATE_MESSAGE packet to the server. This packet specifies the desired security features of the session.
2. The server sends an NTLM CHALLENGE_MESSAGE packet to the client. The packet includes agreed upon security features, and a **nonce** that the server generates.
3. The client sends an NTLM AUTHENTICATE_MESSAGE packet to the server. The packet contains the name of a user and a response that proves that the client knows the user's password. The server validates the response sent by the client. If the user name is for a local account, it can validate the response by using information in its local account database. If the user name is for a domain account, it can validate the response by sending the user authentication information (the user name, the challenge sent to the client, and the response received from the client) to a **domain controller (DC)** that can validate the response.
4. If the challenge and the response prove that the client knows the user's password, the authentication succeeds and the application protocol continues according to its specification. If the authentication fails, the server may send the status in an application protocol-specified way, or it may simply terminate the connection.

1.3.1.2 NTLM Connectionless (Datagram-Oriented) Call Flow

The following illustration shows a typical NTLM connectionless (datagram-oriented) call flow.

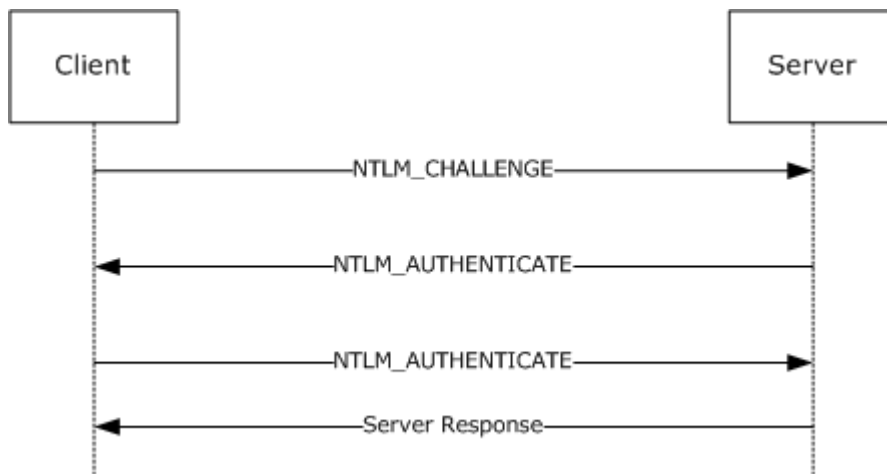


Figure 3: Connectionless NTLM message flow

Although it appears that the server is initiating the request, the client initiates the sequence by sending a packet specified by the application protocol in use.

1. The server sends the client an NTLM CHALLENGE_MESSAGE packet. The packet includes an indication of the security features desired by the server, and a nonce that the server generates.
2. The client sends an NTLM AUTHENTICATE_MESSAGE packet to the server. The packet contains the name of a user and a response that proves that the client has the user's password. The server validates the response sent by the client. If the user name is for a local account, it can validate the response by using information in its local account database. If the user name is for a domain account, it validates the response by sending the user authentication information (the user name, the challenge sent to the client, and the response received from the client) to a domain controller (DC) that can validate the response.
3. If the challenge and the response prove that the client has the user's password, the authentication succeeds and the application protocol continues according to its specification. If the authentication fails, the server sends the status in an application protocol-specified way.

1.4 Relationship to Other Protocols

Because NTLM is embedded in the application protocol, it does not have transport dependencies of its own.

NTLM is used for authentication by several application protocols including SMB, POP3, Telnet, and HTTP. For an example of how NTLM is used in SMB, see section 4.

Other protocols invoke NTLM as a function library. The interface to that library is specified in GSS-API [\[RFC2743\]](#). The NTLM implementation of GSS-API calls is specified in section [3.4.9.<2>](#)

1.5 Prerequisites/Preconditions

None.

1.6 Applicability Statement

An implementer may use the NTLM Authentication Protocol to provide for client authentication (where the server verifies the client's identity) for applications. Because NTLM does not provide for

server authentication, applications that use NTLM are susceptible to attacks from spoofed servers. Applications are therefore discouraged from using NTLM directly. If it is an option, authentication via Kerberos is preferred. <3>

1.7 Versioning and Capability Negotiation

The NTLM authentication version is not negotiated by the protocol. It must be configured on both the client and the server prior to authentication. The version is selected by the client, and requested during the protocol negotiation. If the server does not support the version selected by the client, authentication fails.

NTLM implements capability negotiation by using the flags described in section [2.2.2.5](#). The protocol messages used for negotiation depend on the mode of NTLM being used:

- In connection-oriented NTLM, negotiation starts with a NEGOTIATE_MESSAGE, carrying the client's preferences, and the server replies with NegotiateFlags in the subsequent CHALLENGE_MESSAGE.
- In connectionless NTLM, the server starts the negotiation with the CHALLENGE_MESSAGE and the client replies with NegotiateFlags in the subsequent AUTHENTICATE_MESSAGE.

1.8 Vendor-Extensible Fields

There are no vendor-extensible fields.

1.9 Standards Assignments

The NTLM Authentication Protocol does not use any standards assignments.

2 Messages

The following sections specify how NT LAN Manager (NTLM) Authentication Protocol messages are transported and NTLM Authentication Protocol message syntax.

2.1 Transport

NTLM messages are passed between the client and server. The NTLM messages **MUST** be embedded within the application protocol that is using NTLM authentication. NTLM itself does not establish any transport connections.

2.2 Message Syntax

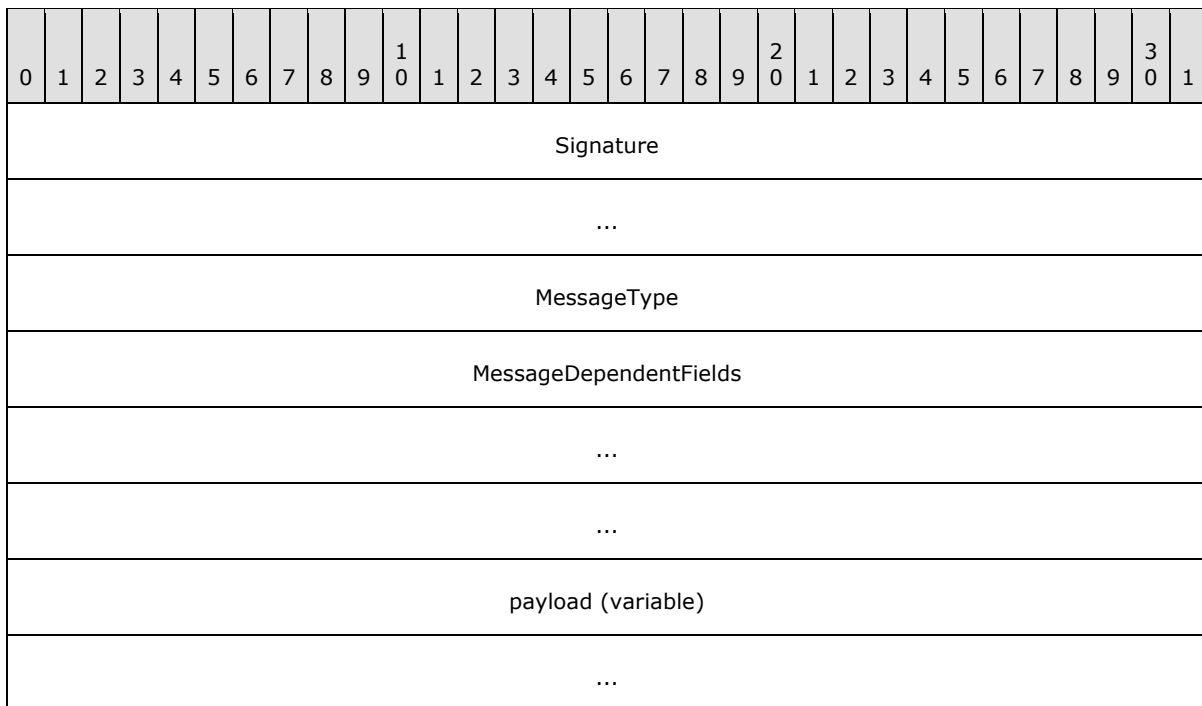
The NTLM Authentication Protocol consists of three message packet types used during authentication and one message type used for message integrity after authentication has occurred.

The authentication messages:

- [NEGOTIATE MESSAGE \(2.2.1.1\)](#)
- [CHALLENGE MESSAGE \(2.2.1.2\)](#)
- [AUTHENTICATE MESSAGE \(2.2.1.3\)](#)

are variable-length messages containing a fixed-length header and a variable-sized message payload. The fixed-length header always starts as shown below with a **Signature** and **MessageType** field.

Depending on the **MessageType** field, the message may have other message-dependent fixed-length fields. The fixed-length fields are then followed by a variable-length message payload.



Signature (8 bytes): An 8-byte character array that MUST contain the ASCII string ('N', 'T', 'L', 'M', 'S', 'S', 'P', '\0').

MessageType (4 bytes): The **MessageType** field MUST take one of the following values from the list below:

Value	Meaning
NtLmNegotiate 0x00000001	The message is a NEGOTIATE_MESSAGE.
NtLmChallenge 0x00000002	The message is a CHALLENGE_MESSAGE.
NtLmAuthenticate 0x00000003	The message is an AUTHENTICATE_MESSAGE

MessageDependentFields (12 bytes): The NTLM message contents, as specified in section [2.2.1](#).

payload (variable): The payload data contains a message-dependent number of individual payload messages. This payload data is referenced by byte offsets located in the **MessageDependentFields**.

The message integrity message, NTLMSSP_MESSAGE_SIGNATURE (as specified in section [2.2.1.4](#)) is fixed length and is appended to the calling application's messages. This message packet type is used only when an application wishes to perform message integrity or confidentiality operations, based on the session key negotiated during a successful authentication.

All multiple-byte values are encoded in little-endian byte order. Unless specified otherwise, 16-bit value fields are of type unsigned short, while 32-bit value fields are of type unsigned long.

All character string fields in NEGOTIATE_MESSAGE contain characters in the **OEM character set**. As specified in section 2.2.2.5, the client and server negotiate if they both support UNICODE characters—in which case, all character string fields in the CHALLENGE_MESSAGE and AUTHENTICATE_MESSAGE contain UNICODE_STRING unless otherwise specified. Otherwise, the OEM character set is used. Agreement between client and server on the choice of OEM character set is not covered by the protocol and MUST occur out-of-band.

2.2.1 NTLM Messages

2.2.1.1 NEGOTIATE_MESSAGE

The NEGOTIATE_MESSAGE packet defines an NTLM Negotiate message that is sent from the client to the server. This message allows the client to specify its supported NTLM options to the server. <4>

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31
Signature																															
...																															
MessageType																															
NegotiateFlags																															
DomainNameFields																															
...																															
WorkstationFields																															
...																															
Version																															
...																															
Payload (variable)																															
...																															

Signature (8 bytes): An 8-byte character array that MUST contain the ASCII string ('N', 'T', 'L', 'M', 'S', 'S', 'P', '\0').

MessageType (4 bytes): A 32-bit unsigned integer that indicates the message type. This field MUST be set to 0x00000001.

NegotiateFlags (4 bytes): A [NEGOTIATE](#) structure that contains a set of bit flags, as defined in section [2.2.2.5](#). The client sets flags to indicate options it supports.

DomainNameFields (8 bytes): If the NTLMSSP_NEGOTIATE_OEM_DOMAIN_SUPPLIED flag is not set in **NegotiateFlags**, indicating that no **DomainName** is supplied in **Payload**, then the **DomainNameLen**, **DomainNameMaxLen**, and **DomainNameBufferOffset** fields SHOULD be set to zero and MUST be ignored on receipt and treated as if they were zero. Otherwise, these fields are defined as:

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31
DomainNameLen																DomainNameMaxLen															
DomainNameBufferOffset																															

DomainNameLen (2 bytes): A 16-bit unsigned integer that defines the size, in bytes, of **DomainName** in **Payload**.

DomainNameMaxLen (2 bytes): A 16-bit unsigned integer that SHOULD be set to 0x0000 and MUST be ignored on receipt.

DomainNameBufferOffset (4 bytes): A 32-bit unsigned integer that defines the offset, in bytes, from the beginning of the NEGOTIATE_MESSAGE to **DomainName** in **Payload**.

WorkstationFields (8 bytes): If the NTLMSSP_NEGOTIATE_OEM_WORKSTATION_SUPPLIED flag is not set in **NegotiateFlags**, indicating that no **WorkstationName** is supplied in **Payload**, then the **WorkstationLen**, **WorkstationMaxLen**, and **WorkstationBufferOffset** fields SHOULD be set to zero and MUST be ignored on receipt and treated as if they were zero. Otherwise, these fields are defined as:

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31
WorkstationLen																WorkstationMaxLen															
WorkstationBufferOffset																															

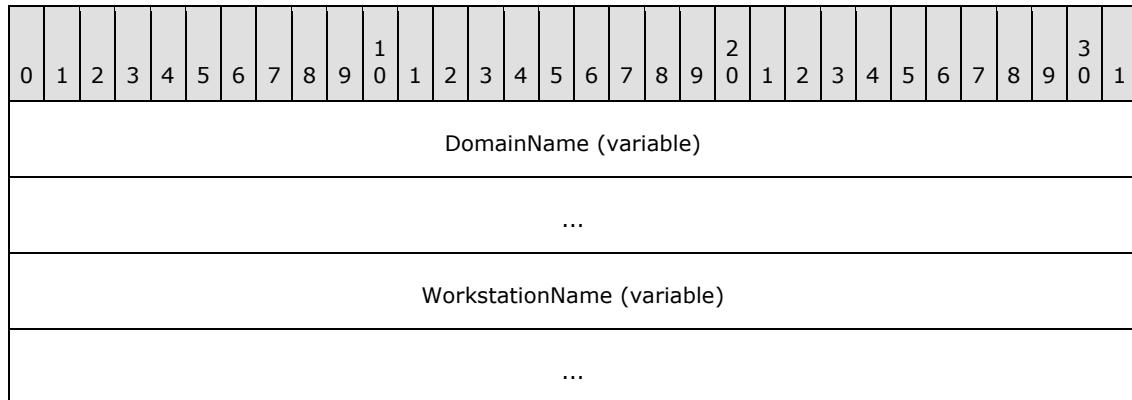
WorkstationLen (2 bytes): A 16-bit unsigned integer that defines the size, in bytes, of **WorkStationName** in **Payload**.

WorkstationMaxLen (2 bytes): A 16-bit unsigned integer that SHOULD be set to 0x0000 and MUST be ignored on receipt.

WorkstationBufferOffset (4 bytes): A 32-bit unsigned integer that defines the offset, in bytes, from the beginning of the NEGOTIATE_MESSAGE to **WorkstationName** in **Payload**.

Version (8 bytes): A [VERSION](#) structure (as defined in section [2.2.2.9](#)) that is present only when the NTLMSSP_NEGOTIATE_VERSION flag is set in the **NegotiateFlags** field. This structure is used for debugging purposes only. In normal (non-debugging) protocol messages, it is ignored and does not affect the NTLM message processing. [<5>](#)

Payload (variable): A byte-array that contains the data referred to by the **DomainNameBufferOffset** and **WorkstationBufferOffset** message fields. **Payload** data MAY be present in any order within the **Payload** field, with variable length padding before or after the datum. The data that MAY be present in the **Payload** field of this message, in no particular order, are:

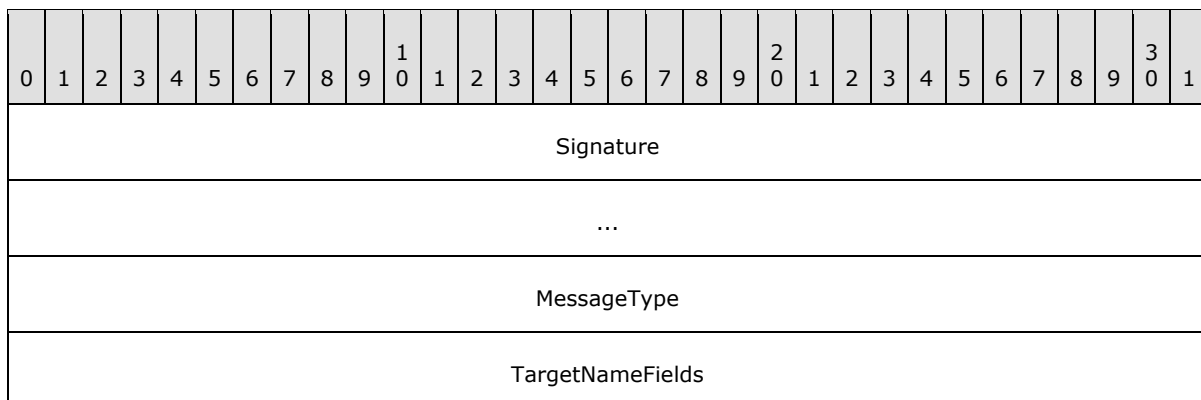


DomainName (variable): If **DomainNameLen** does not equal 0x0000, **DomainName** MUST be a byte-array that contains the name of the client authentication domain that MUST be encoded using the original equipment manufacturer (OEM) character set. Otherwise, this data is not present. [<6>](#)

WorkstationName (variable): If **WorkstationLen** does not equal 0x0000, **WorkstationName** MUST be a byte array that contains the name of the client machine that MUST be encoded using the OEM character set. Otherwise, this data is not present.

2.2.1.2 CHALLENGE_MESSAGE

The CHALLENGE_MESSAGE packet defines an NTLM challenge message that is sent from the server to the client. The CHALLENGE_MESSAGE is used by the server to challenge the client to prove its identity. For connection-oriented requests, the CHALLENGE_MESSAGE generated by the server is in response to the [NEGOTIATE_MESSAGE \(section 2.2.1.1\)](#) from the client. [<7>](#)



...
NegotiateFlags
ServerChallenge
...
Reserved
...
TargetInfoFields
...
Version
...
Payload (variable)
...

Signature (8 bytes): An 8-byte character array that MUST contain the ASCII string ('N', 'T', 'L', 'M', 'S', 'S', 'P', '\0').

MessageType (4 bytes): A 32-bit unsigned integer that indicates the message type. This field MUST be set to 0x00000002.

TargetNameFields (8 bytes): If the NTLMSSP_REQUEST_TARGET flag is not set in **NegotiateFlags**, indicating that no **TargetName** is required, then **TargetNameLen**, **TargetNameMaxLen** and **TargetNameBufferOffset** SHOULD be set to zero on transmission and MUST be ignored on receipt and treated as if they were zero. Otherwise, these fields are defined as:

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31
TargetNameLen																TargetNameMaxLen															
TargetNameBufferOffset																															

TargetNameLen (2 bytes): A 16-bit unsigned integer that defines the size, in bytes, of **TargetName** in **Payload**.

TargetNameMaxLen (2 bytes): A 16-bit unsigned integer that SHOULD be set to 0x0000 and MUST be ignored on receipt.

TargetNameBufferOffset (4 bytes): A 32-bit unsigned integer that defines the offset, in bytes, from the beginning of the CHALLENGE_MESSAGE to **TargetName** in **Payload**. If **TargetName** is a Unicode string, the values of **TargetNameBufferOffset** and **TargetNameLen** MUST be multiples of 2.

NegotiateFlags (4 bytes): A [NEGOTIATE](#) structure that contains a set of bit flags, as defined by section [2.2.2.5](#). The server sets flags to indicate options it supports or, if there has been a NEGOTIATE_MESSAGE (section 2.2.1.1), the choices it has made from the options offered by the client.

ServerChallenge (8 bytes): A 64-bit value that contains the NTLM challenge. The challenge is a 64-bit nonce. The processing of the ServerChallenge is specified in sections [3.1.5](#) and [3.2.5](#).

Reserved (8 bytes): An 8-byte array whose elements MUST be 0x00 and MUST be ignored on receipt.

TargetInfoFields (8 bytes): If the NTLMSSP_NEGOTIATE_TARGET_INFO flag of **NegotiateFlags** is clear, indicating that no **TargetInfo** is required, then **TargetInfoLen**, **TargetInfoMaxLen**, and **TargetInfoBufferOffset** SHOULD be set to zero on transmission and MUST be ignored on receipt and treated as if they were zero. Otherwise, these fields are defined as:

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31
TargetInfoLen																TargetInfoMaxLen															
TargetInfoBufferOffset																															

TargetInfoLen (2 bytes): A 16-bit unsigned integer that defines the size, in bytes, of **TargetInfo** in **Payload**.

TargetInfoMaxLen (2 bytes): A 16-bit unsigned integer that SHOULD be set to 0x0000 and MUST be ignored on receipt.

TargetInfoBufferOffset (4 bytes): A 32-bit unsigned integer that defines the offset, in bytes, from the beginning of the CHALLENGE_MESSAGE to **TargetInfo** in **Payload**.

Version (8 bytes): A [VERSION](#) structure (as defined in section [2.2.2.9](#)) that is present only when the NTLMSSP_NEGOTIATE_VERSION flag is set in the **NegotiateFlags** field. This structure is used for debugging purposes only. In normal (non-debugging) protocol messages, it is ignored and does not affect the NTLM message processing. [<8>](#)

Payload (variable): A byte array that contains the data referred to by the **TargetNameBufferOffset** and **TargetInfoBufferOffset** message fields. Payload data MAY be present in any order within the **Payload** field, with variable-length padding before or after the datum. The data that MAY be present in the **Payload** field of this message, in no particular order, are:

0	1	2	3	4	5	6	7	8	9	0	1	2	3	4	5	6	7	8	9	0	1	2	3	4	5	6	7	8	9	0	1
TargetName (variable)																															
...																															
TargetInfo (variable)																															
...																															

TargetName (variable): If **TargetNameLen** does not equal 0x0000, **TargetName** MUST be a byte array that contains the name of the server authentication realm, and MUST be expressed in the negotiated character set. A server that is a member of a domain returns the domain of which it is a member and a server that is not a member of a domain returns the server name.

TargetInfo (variable): If **TargetInfoLen** does not equal 0x0000, **TargetInfo** MUST be a byte array that contains a sequence of AV_PAIR structures. The AV_PAIR structure is defined in section [2.2.2.1](#). The length of each AV_PAIR is determined by its **AvLen** field (plus 4 bytes).

Note An AV_PAIR structure can start on any byte alignment and the sequence of AV_PAIRs has no padding between structures.

The sequence MUST be terminated by an AV_PAIR structure with an **AvId** field of MsvAvEOL. The total length of the **TargetInfo** byte array is the sum of the lengths, in bytes, of the AV_PAIR structures it contains.

Note If a **TargetInfo** AV_PAIR Value is textual, it MUST be encoded in Unicode irrespective of what character set was negotiated, as specified in section [2.2.2.1](#).

2.2.1.3 AUTHENTICATE_MESSAGE

The AUTHENTICATE_MESSAGE packet defines an NTLM authenticate message that is sent from the client to the server after the [CHALLENGE_MESSAGE \(section 2.2.1.2\)](#) is processed by the client.

<9>

0	1	2	3	4	5	6	7	8	9	0	1	2	3	4	5	6	7	8	9	0	1	2	3	4	5	6	7	8	9	0	1
Signature																															
...																															
MessageType																															
LmChallengeResponseFields																															

...
NtChallengeResponseFields
...
DomainNameFields
...
UserNameFields
...
WorkstationFields
...
EncryptedRandomSessionKeyFields
...
NegotiateFlags
Version
...
Payload (variable)
...

Signature (8 bytes): An 8-byte character array that MUST contain the ASCII string ('N', 'T', 'L', 'M', 'S', 'S', 'P', '\0').

MessageType (4 bytes): A 32-bit unsigned integer that indicates the message type. This field MUST be set to 0x00000003.

LmChallengeResponseFields (8 bytes): If the client chooses not to send an **LmChallengeResponse** to the server, then **LmChallengeResponseLen**, **LmChallengeResponseMaxLen**, and **LmChallengeResponseBufferOffset** MUST be set to zero on transmission. Otherwise, these fields are defined as:

0	1	2	3	4	5	6	7	8	9	0	1	2	3	4	5	6	7	8	9	0	1	2	3	4	5	6	7	8	9	0	1
LmChallengeResponseLen																LmChallengeResponseMaxLen															
LmChallengeResponseBufferOffset																															

LmChallengeResponseLen (2 bytes): A 16-bit unsigned integer that defines the size, in bytes, of **LmChallengeResponse** in **Payload**.

LmChallengeResponseMaxLen (2 bytes): A 16-bit unsigned integer that SHOULD be set to 0x0000 and MUST be ignored on receipt.

LmChallengeResponseBufferOffset (4 bytes): A 32-bit unsigned integer that defines the offset, in bytes, from the beginning of the AUTHENTICATE_MESSAGE to **LmChallengeResponse** in **Payload**.

NtChallengeResponseFields (8 bytes): If the client chooses not to send an **NtChallengeResponse** to the server, then **NtChallengeResponseLen**, **NtChallengeResponseMaxLen**, and **NtChallengeResponseBufferOffset** MUST be set to zero on transmission. Otherwise, these fields are defined as:

0	1	2	3	4	5	6	7	8	9	0	1	2	3	4	5	6	7	8	9	0	1	2	3	4	5	6	7	8	9	0	1
NtChallengeResponseLen																NtChallengeResponseMaxLen															
NtChallengeResponseBufferOffset																															

NtChallengeResponseLen (2 bytes): A 16-bit unsigned integer that defines the size, in bytes, of **NtChallengeResponse** in **Payload**.

NtChallengeResponseMaxLen (2 bytes): A 16-bit unsigned integer that SHOULD be set to 0x0000 and MUST be ignored on receipt.

NtChallengeResponseBufferOffset (4 bytes): A 32-bit unsigned integer that defines the offset, in bytes, from the beginning of the AUTHENTICATE_MESSAGE to **NtChallengeResponse** in **Payload**.<10>

DomainNameFields (8 bytes): If the client chooses not to send a **DomainName** to the server, then **DomainNameLen**, **DomainNameMaxLen**, and **DomainNameBufferOffset** MUST be set to zero on transmission. Otherwise, these fields are defined as:

0	1	2	3	4	5	6	7	8	9	0	1	2	3	4	5	6	7	8	9	0	1	2	3	4	5	6	7	8	9	0	1
DomainNameLen																DomainNameMaxLen															
DomainNameBufferOffset																															

DomainNameLen (2 bytes): A 16-bit unsigned integer that defines the size, in bytes, of **DomainName** in **Payload**, not including a NULL terminator.

DomainNameMaxLen (2 bytes): A 16-bit unsigned integer that SHOULD be set to 0x0000 and MUST be ignored on receipt.

DomainNameBufferOffset (4 bytes): A 32-bit unsigned integer that defines the offset, in bytes, from the beginning of the AUTHENTICATE_MESSAGE to **DomainName** in **Payload**. If **DomainName** is a Unicode string, the values of **DomainNameBufferOffset** and **DomainNameLen** MUST be multiples of 2.

UserNameFields (8 bytes): If the client chooses not to send a **UserName** to the server, then **UserNameLen**, **UserNameMaxLen**, and **UserNameBufferOffset** MUST be set to zero on transmission. Otherwise, these fields are defined as:

0	1	2	3	4	5	6	7	8	9	0	1	2	3	4	5	6	7	8	9	0	1	2	3	4	5	6	7	8	9	0	1
UserNameLen																UserNameMaxLen															
UserNameBufferOffset																															

UserNameLen (2 bytes): A 16-bit unsigned integer that defines the size, in bytes, of **UserName** in **Payload**, not including a NULL terminator.

UserNameMaxLen (2 bytes): A 16-bit unsigned integer that SHOULD be set to 0x0000 and MUST be ignored on receipt.

UserNameBufferOffset (4 bytes): A 32-bit unsigned integer that defines the offset, in bytes, from the beginning of the AUTHENTICATE_MESSAGE to **UserName** in **Payload**. If **UserName** to be sent contains a Unicode string, the values of **UserNameBufferOffset** and **UserNameLen** MUST be multiples of 2.

WorkstationFields (8 bytes): If the client chooses not to send **Workstation** to the server, then **WorkstationLen**, **WorkstationMaxLen**, and **WorkstationBufferOffset** MUST be set to zero on transmission. Otherwise, these fields are defined as:

0	1	2	3	4	5	6	7	8	9	0	1	2	3	4	5	6	7	8	9	0	1	2	3	4	5	6	7	8	9	0	1
WorkstationLen																WorkstationMaxLen															
WorkstationBufferOffset																															

WorkstationLen (2 bytes): A 16-bit unsigned integer that defines the size, in bytes, of **Workstation** in **Payload**, not including a NULL terminator.

WorkstationMaxLen (2 bytes): A 16-bit unsigned integer that SHOULD be set to 0x0000 and MUST be ignored on receipt.

WorkstationBufferOffset (4 bytes): A 32-bit unsigned integer that defines the offset, in bytes, from the beginning of the AUTHENTICATE_MESSAGE to **Workstation** in **Payload**. If **Workstation** contains a Unicode string, the values of **WorkstationBufferOffset** and **WorkstationLen** MUST be multiples of 2.

EncryptedRandomSessionKeyFields (8 bytes): If the NTLMSSP_NEGOTIATE_KEY_EXCH flag is not set in **NegotiateFlags**, indicating that no **EncryptedRandomSessionKey** is supplied, then **EncryptedRandomSessionKeyLen**, **EncryptedRandomSessionKeyMaxLen**, and **EncryptedRandomSessionKeyBufferOffset** SHOULD be set to zero on transmission and MUST be ignored on receipt and treated as if they were zero. Otherwise, these fields are defined as:

0	1	2	3	4	5	6	7	8	9	0	1	2	3	4	5	6	7	8	9	0	1	2	3	4	5	6	7	8	9	0	1
EncryptedRandomSessionKeyLen																EncryptedRandomSessionKeyMaxLen															
EncryptedRandomSessionKeyBufferOffset																															

EncryptedRandomSessionKeyLen (2 bytes): A 16-bit unsigned integer that defines the size, in bytes, of **EncryptedRandomSessionKey** in **Payload**.

EncryptedRandomSessionKeyMaxLen (2 bytes): A 16-bit unsigned integer that SHOULD be set to 0x0000 and MUST be ignored on receipt.

EncryptedRandomSessionKeyBufferOffset (4 bytes): A 32-bit unsigned integer that defines the offset, in bytes, from the beginning of the AUTHENTICATE_MESSAGE to **EncryptedRandomSessionKey** in **Payload**.

NegotiateFlags (4 bytes): In connectionless mode, a [NEGOTIATE](#) structure that contains a set of bit flags (as defined in section [2.2.2.5](#)) and represents the conclusion of negotiation—the choices the client has made from the options the server offered in the CHALLENGE_MESSAGE. In connection-oriented mode, this field of the message MUST be ignored (as specified in section [1.7](#)).

Version (8 bytes): A [VERSION](#) structure (as specified in section [2.2.2.9](#)) that is present only when the NTLMSSP_NEGOTIATE_VERSION flag is set in the **NegotiateFlags** field. This

structure is used for debugging purposes only. In normal protocol messages, it is ignored and does not affect the NTLM message processing. <11>

Payload (variable): A byte array that contains the data referred to by the **LmChallengeResponseBufferOffset**, **NtChallengeResponseBufferOffset**, **DomainNameBufferOffset**, **UserNameBufferOffset**, **WorkstationBufferOffset**, and **EncryptedRandomSessionKeyBufferOffset** message fields. Payload data MAY be present in any order within the **Payload** field, with variable length padding before or after the datum. The data that MAY be present in the **Payload** field of this message, in no particular order, are:

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31
LmChallengeResponse (variable)																															
...																															
NtChallengeResponse (variable)																															
...																															
DomainName (variable)																															
...																															
UserName (variable)																															
...																															
Workstation (variable)																															
...																															
EncryptedRandomSessionKey (variable)																															
...																															

LmChallengeResponse (variable): An [LM_RESPONSE](#) or [LMv2_RESPONSE](#) structure that contains the computed LM response to the challenge. If NTLM v2 authentication is configured (via bit P of **NegotiateFlags**, as specified in section [2.2.2.5](#)), **LmChallengeResponse** MUST be an LMv2_RESPONSE structure, as defined in section [2.2.2.4](#). Otherwise, it MUST be an LM_RESPONSE structure, as defined in section [2.2.2.3](#).

NtChallengeResponse (variable): An [NTLM_RESPONSE](#) or [NTLMv2_RESPONSE](#) structure that contains the computed NT response to the challenge. If NTLM v2 authentication is configured (via bit P of **NegotiateFlags**, section [2.2.2.5](#)),

NtChallengeResponse MUST be an NTLMv2_RESPONSE as defined in section [2.2.2.8](#). Otherwise, it MUST be an NTLM_RESPONSE structure as defined in section [2.2.2.6](#).

DomainName (variable): The domain or computer name hosting the user account. **DomainName** MUST be encoded in the negotiated character set.

UserName (variable): The name of the user to be authenticated. **UserName** MUST be encoded in the negotiated character set.

Workstation (variable): The name of the computer that the user is logged onto. **Workstation** MUST be encoded in the negotiated character set.

EncryptedRandomSessionKey (variable): The client's encrypted random session key. **EncryptedRandomSessionKey** and its usage are defined in sections [3.1.5](#) and [3.2.5](#).

2.2.1.4 NTLMSSP_MESSAGE_SIGNATURE

The NTLMSSP_MESSAGE_SIGNATURE structure, computed as specified in section [3.4.4](#), specifies the signature block used for application message integrity and confidentiality when the NTLMSSP_NEGOTIATE_NTLM2 flag is negotiated. This structure is then passed back to the application which embeds it within the application protocol messages, along with the NTLM-encrypted or integrity-protected application message data.

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31
Version																															
Checksum																															
...																															
SeqNum																															

Version (4 bytes): A 32-bit unsigned integer that contains the signature version. This field MUST be 0x00000001.

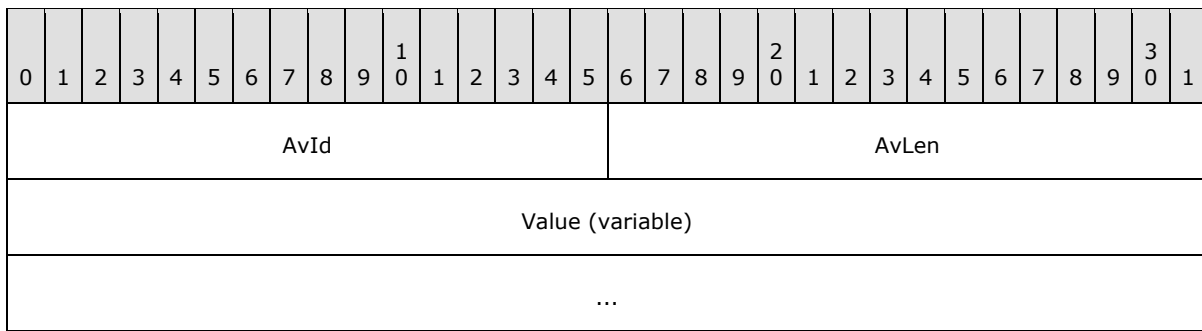
Checksum (8 bytes): An 8-byte array that contains the checksum for the message.

SeqNum (4 bytes): A 32-bit unsigned integer that contains the NTLM sequence number for this application message.

2.2.2 NTLM Structures

2.2.2.1 AV_PAIR

The AV_PAIR structure defines an attribute/value pair. Sequences of AV_PAIR structures are used in the [CHALLENGE MESSAGE](#) and [AUTHENTICATE MESSAGE](#) structures. **Note** Although the following figure suggests that the most significant bit (MSB) of **AvId** is aligned with the MSB of a 32-bit word, an AV_PAIR can be aligned on any byte boundary and can be 4+N bytes long for arbitrary N (N = the contents of **AvLen**).



AvId (2 bytes): A 16-bit unsigned integer that defines the information type in the **Value** field. This field **MUST** be one of the values from the table below.

Value	Meaning
MsvAvEOL 0	NONE; AvLen MUST be 0.
MsvAvNbComputerName 1	The server's NetBIOS computer name. The name MUST be in Unicode, and is not null-terminated.
MsvAvNbDomainName 2	The server's NetBIOS domain name . The name MUST be in Unicode, and is not null-terminated.
MsvAvDnsComputerName 3	The server's Active Directory (AD) DNS computer name. The name MUST be in Unicode, and is not null-terminated.
MsvAvDnsDomainName 4	The server's Active Directory (AD) DNS domain name. The name MUST be in Unicode, and is not null-terminated.
MsvAvDnsTreeName 5	The server's Active Directory (AD) DNS forest tree name . The name MUST be in Unicode, and is not null-terminated.
MsvAvFlags 6	An optional field containing a 32-bit value indicating server configuration. The only defined value is 0x00000001, which indicates the server forces all authentication to the guest account . This indicator MAY be used by a client application to indicate forced authentication to the guest account.
MsvAvTimestamp 7	A FILETIME structure (as specified in [MS-DTYP] section 2.3.1) in little-endian byte order that contains the server local time. <12>
MsAvRestrictions 8	A Restrictions Encoding structure, as defined in section 2.2.2.2 . The Restrictions field contains a structure representing the integrity level of the security principal, as well as a MachineID used to detect when the authentication attempt is looping back to the same machine.

AvLen (2 bytes): A 16-bit unsigned integer that defines the length, in bytes, of **Value**.

Value (variable): A variable length byte-array that contains the value defined for this AV pair entry. The contents of this field depend on the type expressed in the **AvId** field. The available types and resulting format and contents of this field are specified in the table within the AvId field description in this topic.

2.2.2.2 Restriction Encoding

Restriction Encoding in NTLM allows platform-specific restrictions to be encoded within an authentication exchange. The client produces additional restrictions to be applied to the server when authorization decisions are to be made. If the server does not support the restrictions, then the client's authorization on the server is unchanged. <13>

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31
Integrity Level																															
Subject Integrity Level																															
MachineID																															
...																															
...																															
...																															
...																															
...																															
...																															
...																															
...																															

Integrity Level (4 bytes): Indicates an integrity level is present in **Subject Integrity Level**.

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	I

Where the bits are defined as:

Value	Description
I	If set, indicates that the recipient should apply the integrity level encoded below. When clear, no integrity is present.

Subject Integrity Level (4 bytes): A 32-bit integer value indicating an integrity level of the client. <14>

MachineID (32 bytes): A 256-bit random number that is used to detect when the client and server of the NTLM authentication exchange are on the same machine. <15>

2.2.2.3 LM_RESPONSE

The LM_RESPONSE structure defines the NTLM v1 authentication **LmChallengeResponse** in the [AUTHENTICATE_MESSAGE](#). This response is used only when NTLM v1 authentication is configured.

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31
Response																															
...																															
...																															
...																															
...																															
...																															

Response (24 bytes): A 24-byte array of unsigned char that contains the client's **LmChallengeResponse** as defined in section [3.3.1](#).

2.2.2.4 LMv2_RESPONSE

The LMv2_RESPONSE structure defines the NTLM v2 authentication **LmChallengeResponse** in the [AUTHENTICATE_MESSAGE](#). This response is used only when NTLM v2 authentication is configured.

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31
Response																															
...																															
...																															
...																															
ChallengeFromClient																															
...																															

Response (16 bytes): A 16-byte array of unsigned char that contains the client's LM challenge-response. This is the portion of the **LmChallengeResponse** field to which the HMAC_MD5 algorithm has been applied, as defined in section [3.3.2](#). Specifically, **Response** corresponds to the result of applying the HMAC_MD5 algorithm, using the key ResponseKeyLM, to a message consisting of the concatenation of the ResponseKeyLM, ServerChallenge and ClientChallenge.

ChallengeFromClient (8 bytes): An 8-byte array of unsigned char that contains the client's **ClientChallenge**, as defined in section [3.1.5.1.2](#).

2.2.2.5 NEGOTIATE

During NTLM authentication, each of the following flags is a possible value of the **NegotiateFlags** field of the [NEGOTIATE MESSAGE](#), [CHALLENGE MESSAGE](#), and [AUTHENTICATE MESSAGE](#), unless otherwise noted. These flags define client or server NTLM capabilities supported by the sender.

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31
X	V	U	r1	r2	r3	T	r4	S	R	r5	Q	P	O	N	M	L	r6	K	J	r7	I	H	r8	G	F	E	D	r9	C	B	A

X (1 bit): If set, requests 56-bit encryption. If the client sends NTLMSPP_NEGOTIATE_56 to the server in the NEGOTIATE_MESSAGE, the server MUST return NTLMSPP_NEGOTIATE_56 to the client in the CHALLENGE_MESSAGE only if the client sets NTLMSPP_NEGOTIATE_SEAL or NTLMSPP_NEGOTIATE_SIGN. Otherwise it is ignored. If both NTLMSPP_NEGOTIATE_56 and NTLMSPP_NEGOTIATE_128 are requested and supported by the client and server, NTLMSPP_NEGOTIATE_56 and NTLMSPP_NEGOTIATE_128 will both be returned to the client. It is recommended that clients and servers that set NTLMSPP_NEGOTIATE_SEAL always set NTLMSPP_NEGOTIATE_56 if it is supported. An alternate name for this field is **NTLMSPP_NEGOTIATE_56**.

V (1 bit): If set, requests an explicit key exchange. If the client sends NTLMSPP_NEGOTIATE_KEY_EXCH to the server in the NEGOTIATE_MESSAGE, the server MUST return NTLMSPP_NEGOTIATE_KEY_EXCH to the client in the CHALLENGE_MESSAGE and use key exchange only if the client sets NTLMSPP_NEGOTIATE_SIGN or

NTLMSSP_NEGOTIATE_SEAL. Otherwise it is ignored. Use of this capability is recommended because message integrity or confidentiality can be provided only when this flag is negotiated and a key exchange key is created. See sections [3.1.5.1](#) and [3.1.5.2](#) for details. An alternate name for this field is **NTLMSSP_NEGOTIATE_KEY_EXCH**.

U (1 bit): If set, requests 128-bit session key negotiation. If the client sends NTLMSSP_NEGOTIATE_128 to the server in the NEGOTIATE_MESSAGE, the server MUST return NTLMSSP_NEGOTIATE_128 to the client in the CHALLENGE_MESSAGE only if the client sets NTLMSSP_NEGOTIATE_SEAL or NTLMSSP_NEGOTIATE_SIGN. Otherwise it is ignored. If both NTLMSSP_NEGOTIATE_56 and NTLMSSP_NEGOTIATE_128 are requested and supported by the client and server, NTLMSSP_NEGOTIATE_56 and NTLMSSP_NEGOTIATE_128 will both be returned to the client. It is recommended that clients and servers that set NTLMSSP_NEGOTIATE_SEAL always set NTLMSSP_NEGOTIATE_128 if it is supported. An alternate name for this field is **NTLMSSP_NEGOTIATE_128**.

r1 (1 bit): This bit is unused and MUST be zero.

r2 (1 bit): This bit is unused and MUST be zero.

r3 (1 bit): This bit is unused and MUST be zero.

T (1 bit): If set, requests the protocol version number. The data corresponding to this flag is provided in the **Version** field of the NEGOTIATE_MESSAGE, the CHALLENGE_MESSAGE, and the AUTHENTICATE_MESSAGE. [<16>](#) An alternate name for this field is **NTLMSSP_NEGOTIATE_VERSION**.

r4 (1 bit): This bit is unused and MUST be zero.

S (1 bit): If set, requests extended information about the server authentication realm to be sent as [AV_PAIR](#) in the **TargetInfo** payload, as specified in section [2.2.2.7](#). If the client sends NTLMSSP_NEGOTIATE_TARGET_INFO to the server in the NEGOTIATE_MESSAGE, the server MUST support the request and return NTLMSSP_NEGOTIATE_TARGET_INFO to the client in the CHALLENGE_MESSAGE. In that case, the data corresponding to this flag is provided by the server in the **TargetInfo** field of the CHALLENGE_MESSAGE. An alternate name for this field is **NTLMSSP_NEGOTIATE_TARGET_INFO**.

R (1 bit): If set, requests the usage of the **LMOWF**, as specified in section [3.3](#). An alternate name for this field is **NTLMSSP_REQUEST_NON_NT_SESSION_KEY**.

r5 (1 bit): This bit is unused and MUST be zero.

Q (1 bit): If set, requests an **identify level token**. An alternate name for this field is **NTLMSSP_NEGOTIATE_IDENTIFY**.

P (1 bit): If set, requests usage of the NTLM v2 session security. NTLMSSP_NEGOTIATE_LM_KEY and NTLMSSP_NEGOTIATE_NTLM2 are mutually exclusive. If both NTLMSSP_NEGOTIATE_NTLM2 and NTLMSSP_NEGOTIATE_LM_KEY are requested, NTLMSSP_NEGOTIATE_NTLM2 alone MUST be returned to the client. NTLM v2 authentication session key generation must be supported by both the client and the DC in order to be used, and NTLM v2 session security signing and sealing requires support from the client and the server in order to be used. [<17>](#) An alternate name for this field is **NTLMSSP_NEGOTIATE_NTLM2**.

O (1 bit): If set, **TargetName** MUST be a **share** name. The data corresponding to this flag is provided by the server in the **TargetName** field of the CHALLENGE_MESSAGE. This flag MUST be ignored in the NEGOTIATE_MESSAGE and the AUTHENTICATE_MESSAGE. An alternate name for this field is **NTLMSSP_TARGET_TYPE_SHARE**.

N (1 bit): If set, **TargetName** MUST be a server name. The data corresponding to this flag is provided by the server in the **TargetName** field of the CHALLENGE_MESSAGE. This flag MUST be ignored in the NEGOTIATE_MESSAGE and the AUTHENTICATE_MESSAGE. An alternate name for this field is **NTLMSSP_TARGET_TYPE_SERVER**.

M (1 bit): If set, **TargetName** MUST be a domain name. The data corresponding to this flag is provided by the server in the **TargetName** field of the CHALLENGE_MESSAGE. This flag MUST be ignored in the NEGOTIATE_MESSAGE and the AUTHENTICATE_MESSAGE. An alternate name for this field is **NTLMSSP_TARGET_TYPE_DOMAIN**.

L (1 bit): If set, requests the presence of a signature block on all messages. NTLMSSP_NEGOTIATE_NTLM and NTLMSSP_NEGOTIATE_ALWAYS_SIGN MUST be set in the NEGOTIATE_MESSAGE to the server and the CHALLENGE_MESSAGE to the client. NTLMSSP_NEGOTIATE_ALWAYS_SIGN is overridden by NTLMSSP_NEGOTIATE_SIGN and NTLMSSP_NEGOTIATE_SEAL, if they are supported. An alternate name for this field is **NTLMSSP_NEGOTIATE_ALWAYS_SIGN**.

r6 (1 bit): This bit is unused and MUST be zero.

K (1 bit): This flag indicates whether the **Workstation** field is present. If this flag is not set, the **Workstation** field MUST be ignored. If this flag is set, the **length** field of the **Workstation** field specifies whether the workstation name is non-empty or not. <18> An alternate name for this field is **NTLMSSP_NEGOTIATE_OEM_WORKSTATION_SUPPLIED**.

J (1 bit): If set, the domain name is provided, as specified in section 2.2.1.1. <19> An alternate name for this field is **NTLMSSP_NEGOTIATE_OEM_DOMAIN_SUPPLIED**.

r7 (1 bit): This bit is unused and MUST be zero.

I (1 bit): If set, LM authentication is not allowed and only NT authentication is used. An alternate name for this field is **NTLMSSP_NEGOTIATE_NT_ONLY**.

H (1 bit): If set, requests usage of the NTLM v1 session security protocol. NTLMSSP_NEGOTIATE_NTLM and NTLMSSP_NEGOTIATE_ALWAYS_SIGN MUST be set in the NEGOTIATE_MESSAGE to the server and the CHALLENGE_MESSAGE to the client. An alternate name for this field is **NTLMSSP_NEGOTIATE_NTLM**.

r8 (1 bit): This bit is unused and MUST be zero.

G (1 bit): If set, requests LAN Manager (LM) session key computation. NTLMSSP_NEGOTIATE_LM_KEY and NTLMSSP_NEGOTIATE_NTLM2 are mutually exclusive. If both NTLMSSP_NEGOTIATE_LM_KEY and NTLMSSP_NEGOTIATE_NTLM2 are requested, NTLMSSP_NEGOTIATE_NTLM2 alone MUST be returned to the client. NTLM v2 authentication session key generation MUST be supported by both the client and the domain controller (DC) in order to be used, and NTLM v2 session security signing and sealing requires support from the client and the server to be used. <20> An alternate name for this field is **NTLMSSP_NEGOTIATE_LM_KEY**.

F (1 bit): If set, requests datagram-oriented (connectionless) authentication. If NTLMSSP_NEGOTIATE_DATAGRAM is set, then NTLMSSP_NEGOTIATE_KEY_EXCH MUST always be set in the AUTHENTICATE_MESSAGE to the server and the CHALLENGE_MESSAGE to the client. An alternate name for this field is **NTLMSSP_NEGOTIATE_DATAGRAM**.

E (1 bit): If set, requests session key negotiation for message confidentiality. If the client sends NTLMSSP_NEGOTIATE_SEAL to the server in the NEGOTIATE_MESSAGE, the server MUST return NTLMSSP_NEGOTIATE_SEAL to the client in the CHALLENGE_MESSAGE. Clients and servers that set NTLMSSP_NEGOTIATE_SEAL SHOULD always set NTLMSSP_NEGOTIATE_56

and NTLMSSP_NEGOTIATE_128, if they are supported. <21> An alternate name for this field is **NTLMSSP_NEGOTIATE_SEAL**.

D (1 bit): If set, requests session key negotiation for message signatures. If the client sends NTLMSSP_NEGOTIATE_SIGN to the server in the NEGOTIATE_MESSAGE, the server MUST return NTLMSSP_NEGOTIATE_SIGN to the client in the CHALLENGE_MESSAGE. An alternate name for this field is **NTLMSSP_NEGOTIATE_SIGN**.

r9 (1 bit): This bit is unused and MUST be zero.

C (1 bit): If set, a **TargetName** field of the **CHALLENGE_MESSAGE** (section 2.2.1.2) MUST be supplied. An alternate name for this field is **NTLMSSP_REQUEST_TARGET**.

B (1 bit): If A==1, the choice of character set encoding MUST be UNICODE. If A==0 and B==1, the choice of character set MUST be OEM. If A==0 and B==0, the protocol MUST terminate. An alternate name for this field is **NTLM_NEGOTIATE_OEM**.

A (1 bit): If A==1, the choice of character set encoding MUST be UNICODE. If A==0 and B==1, the choice of character set MUST be OEM. If A==0 and B==0, the protocol MUST terminate. An alternate name for this field is **NTLMSSP_NEGOTIATE_UNICODE**.

2.2.2.6 NTLM v1 Response: NTLM_RESPONSE

The NTLM_RESPONSE structure defines the NTLM v1 authentication **NtChallengeResponse** in the [AUTHENTICATE_MESSAGE](#). This response is only used when NTLM v1 authentication is configured.

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31
Response																															
...																															
...																															
...																															
...																															
...																															

Response (24 bytes): A 24-byte array of unsigned char that contains the client's **NtChallengeResponse**, as specified in section 3.3.1.

2.2.2.7 NTLM v2: NTLMv2_CLIENT_CHALLENGE

The NTLMv2_CLIENT_CHALLENGE structure defines the client challenge in the [AUTHENTICATE_MESSAGE](#). This structure is used only when NTLM v2 authentication is configured. <22>

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31
RespType								HiRespType								Reserved1															
Reserved2																															
TimeStamp																															
...																															
ChallengeFromClient																															
...																															
Reserved3																															
AvPairs (variable)																															
...																															

RespType (1 byte): An 8-bit unsigned char that contains the current version of the challenge response type. This field MUST be 0x01.

HiRespType (1 byte): An 8-bit unsigned char that contains the maximum supported version of the challenge response type. This field MUST be 0x01.

Reserved1 (2 bytes): A 16-bit unsigned integer that SHOULD be 0x0000 and MUST be ignored on receipt.

Reserved2 (4 bytes): A 32-bit unsigned integer that SHOULD be 0x00000000 and MUST be ignored on receipt.

TimeStamp (8 bytes): A 64-bit unsigned integer that contains the current system time, represented as the number of 100 nanosecond ticks elapsed since midnight of January 1, 1601.

ChallengeFromClient (8 bytes): An 8-byte array of unsigned char that contains the client's **ClientChallenge**, which is set as specified in section [3.1.5.1.2](#).

Reserved3 (4 bytes): A 32-bit unsigned integer that SHOULD be 0x00000000 and MUST be ignored on receipt.

AvPairs (variable): A byte array that contains a sequence of [AV_PAIR](#) structures, as specified in section [2.2.2.1](#). The sequence contains the server naming context and is terminated by an AV_PAIR structure with an **AvId** field of MsvAvEOL.

2.2.2.8 NTLM2 V2 Response: NTLMv2_RESPONSE

The NTLMv2_RESPONSE structure defines the NTLMv2 authentication NtChallengeResponse in the [AUTHENTICATE MESSAGE](#). This response is only used when NTLMv2 authentication is configured.

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31
Response																															
...																															
...																															
...																															
NTLMv2_CLIENT_CHALLENGE (variable)																															
...																															

Response (16 bytes): A 16-byte array of unsigned char that contains the client's NT challenge-response as defined in section [3.3.2](#). Response corresponds to the NTProofStr variable from section [3.3.2](#).

NTLMv2_CLIENT_CHALLENGE (variable): A variable-length byte array that contains the ClientChallenge as defined in section [3.3.2](#). ChallengeFromClient corresponds to the temp variable from section [3.3.2](#).

2.2.2.9 VERSION

The VERSION structure contains Windows version information that SHOULD be ignored. This structure is used for debugging purposes only and its value does not affect NTLM message processing. It is present in the [NEGOTIATE MESSAGE](#), [CHALLENGE MESSAGE](#), and [AUTHENTICATE MESSAGE](#) messages only if NTLMSSP_NEGOTIATE_VERSION is negotiated. [<23>](#)

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31
ProductMajorVersion								ProductMinorVersion								ProductBuild															
Reserved																								NTLMRevisionCurrent							

ProductMajorVersion (1 byte): An 8-bit unsigned integer that contains the minor version number of the Windows operating system in use. This field MUST contain one of the following values [<24>](#):

Value	Meaning
WINDOWS_MAJOR_VERSION_5 0x05	The major version of the Windows operating system is 0x05.
WINDOWS_MAJOR_VERSION_6 0x06	The major version of the Windows operating system is 0x06.

ProductMinorVersion (1 byte): An 8-bit unsigned integer that contains the minor version number of the Windows operating system in use. This field MUST contain one of the following values<25>:

Value	Meaning
WINDOWS_MINOR_VERSION_0 0x00	The minor version of the Windows operating system is 0x00.
WINDOWS_MINOR_VERSION_1 0x01	The minor version of the Windows operating system is 0x01.
WINDOWS_MINOR_VERSION_2 0x02	The minor version of the Windows operating system is 0x02.

ProductBuild (2 bytes): A 16-bit unsigned integer that contains the build number of the Windows operating system in use. This field MUST be set to any 16-bit quantity that identifies the operating system build number.

Reserved (3 bytes): A 24-bit data area that SHOULD be set to zero and MUST be ignored by the receiver.

NTLMRevisionCurrent (1 byte): An 8-bit unsigned integer that contains a value indicating the current revision of the NTLMSSP in use. This field MUST contain one of the following values<26>:

Value	Meaning
NTLMSSP_REVISION_W2K3 0x0F	Version 15 of the NTLMSSP is in use.
NTLMSSP_REVISION_W2K3_RC1 0x0A	Version 10 of the NTLMSSP is in use.

3 Protocol Details

The following sections offer a detailed specification of the NTLM message computation.

- Sections [3.1.5](#) and [3.2.5](#) specify how the client and server compute messages and respond to messages.
- Section [3.3](#) specifies how the response computation is calculated, depending on whether NTLM v1 or NTLM v2 is used. This includes the ComputeResponse function, as well as the **NTOWF** and LMOWF functions, which are used by the ComputeResponse function.
- Section [3.4](#) specifies how message integrity and message confidentiality are provided, including a detailed specification of the algorithms used to calculate the signing and sealing keys.

The Cryptographic Operations Reference in section [6](#) defines the cryptographic primitives used in this section.

3.1 Client Details

3.1.1 Abstract Data Model

The following sections specify variables that are internal to the client and are maintained across the NTLM authentication sequence.

3.1.1.1 Variables Internal to the Protocol

ClientConfigFlags: The set of client configuration flags (as specified in section [2.2.2.5](#)) that specify the full set of capabilities of the client.

ExportedSessionKey: A 128-bit (16-byte) session key used to derive ClientSigningKey, ClientSealingKey, ServerSealingKey, and ServerSigningKey.

NegFlg: The set of configuration flags (as specified in section [2.2.2.5](#)) that specify the negotiated capabilities of the client and server for the current NTLM session.

User: A string that indicates the name of the user.

UserDom: A string that indicates the name of the user's domain.

The following variables are internal to the client and are maintained for the entire length of the authenticated session:

ClientSigningKey: The signing key used by the client to sign messages and by the server to verify signed client messages. It is generated after the client is authenticated by the server and is not passed over the wire.

ClientSealingKey: The sealing key used by the client to seal messages and by the server to unseal client messages. It is generated after the client is authenticated by the server and is not passed over the wire.

SeqNum: A 4-byte sequence number, as specified in section [3.4.4](#).

ServerSealingKey: The sealing key used by the server to seal messages and by the client to unseal server messages. It is generated after the client is authenticated by the server and is not passed over the wire.

ServerSigningKey: The signing key used by the server to sign messages and by the client to verify signed server messages. It is generated after the client is authenticated by the server and is not passed over the wire.

3.1.1.2 Variables Exposed to the Application

The following parameters are logically available for the application to set. These logical parameters can influence various protocol-defined flags. <27>

Note The variables defined below are logical, abstract parameters that an implementation has to maintain and expose to provide the proper level of **service**. How these variables are maintained and exposed is up to the implementation.

Integrity: Indicates that the caller wants to sign messages so that they cannot be tampered while in transit. Setting this flag results in the NTLMSSP_NEGOTIATE_SIGN flag being set in the **NegotiateFlags** field of the NTLM NEGOTIATE_MESSAGE.

Replay Detect: Indicates that the caller wants to sign messages so that they cannot be replayed. Setting this flag results in the NTLMSSP_NEGOTIATE_SIGN flag being set in the **NegotiateFlags** field of the NTLM NEGOTIATE_MESSAGE.

Sequence Detect: Indicates that the caller wants to sign messages so that they cannot be sent out of order. Setting this flag results in the NTLMSSP_NEGOTIATE_SIGN flag being set in the **NegotiateFlags** field of the NTLM NEGOTIATE_MESSAGE.

Confidentiality: Indicates that the caller wants to encrypt messages so that they cannot be read while in transit. If the Confidentiality option is selected by the client, NTLM performs a bitwise OR operation with the following NTLM Negotiate Flags into the **ClientConfigFlags**(the ClientConfigFlags indicate which features the client host supports):

```
NTLMSSP_NEGOTIATE_SEAL
NTLMSSP_NEGOTIATE_KEY_EXCH
NTLMSSP_NEGOTIATE_LM_KEY
NTLMSSP_NEGOTIATE_NTLM2
```

Datagram: Indicates that the connectionless mode of NTLM is to be selected. If the Datagram option is selected by the client, then connectionless mode is used and NTLM performs a bitwise OR operation with the following NTLM Negotiate Flag into the **ClientConfigFlags**:

```
NTLMSSP_NEGOTIATE_DATAGRAM
```

Identify: Indicates that the caller wants the server to know the identity of the caller but not be allowed to impersonate the caller to resources on that system. Setting this flag results in the NTLMSSP_NEGOTIATE_IDENTIFY flag being set. Indicates that the GSS_C_IDENTIFY_FLAG flag was set in the GSS_Init_sec_context call, as discussed in [\[RFC4757\]](#) section 7.1, and results in the GSS_C_IDENTIFY_FLAG flag set in the authenticator's **checksum** field, as specified in [\[RFC4757\]](#) section 7.1.

3.1.2 Timers

No timers are used by the client.

3.1.3 Initialization

The client side initializes the random number generator when the machine is started. The client initializes the following configuration flags and sets them in the **NegotiateFlags** field of the [NEGOTIATE_MESSAGE](#):

```
NTLMSSP_REQUEST_TARGET
NTLMSSP_NEGOTIATE_NTLM
NTLMSSP_NEGOTIATE_ALWAYS_SIGN
NTLMSSP_NEGOTIATE_NTLM2
NTLMSSP_NEGOTIATE_UNICODE
```

3.1.4 Higher-Layer Triggered Events

The application client initiates NTLM authentication through the **Security Support Provider Interface (SSPI)**, the Microsoft implementation of GSS-API [\[RFC2743\]](#).

- **GSS_Init_sec_context**

The client calls `GSS_Init_sec_context()` to establish a security context with the server. NTLM has no requirements on which flags are used and will simply honor what was requested by the application or protocol. For an example of such a protocol specification, see [\[MS-RPCE\]](#) section 3.3.1.5.2.2, Using a Security Context. The application will send the `NEGOTIATE_MESSAGE`, as specified in section [2.2.1.1](#), to the server as specified in section [3.1.5.1.1](#) Client Initiates the `NEGOTIATE_MESSAGE` Packet.

When the client receives the `CHALLENGE_MESSAGE`, as specified in section [2.2.1.2](#), from the server, the client will call `GSS_Init_sec_context()` with the `CHALLENGE_MESSAGE` as input. The application will send the `AUTHENTICATE_MESSAGE`, as specified in section [2.2.1.3](#), to the server as specified in section [3.1.5.1.2](#) Client Receives a `CHALLENGE_MESSAGE` from the Server.

- **GSS_Wrap**

Once the security context is established, the client can call `GSS_WrapEx()` (section [3.4.9](#)) to encrypt messages.

- **GSS_Unwrap**

Once the security context is established, the client can call `GSS_UnwrapEx()` (section [3.4.10](#)) to decrypt messages that were encrypted by `GSS_WrapEx`.

- **GSS_GetMIC**

Once the security context is established, the client can call `GSS_GetMICEx()` (section [3.4.11](#)) to sign messages, producing an `NTLMSSP_MESSAGE_SIGNATURE` structure whose fields are defined in section [2.2.1.4](#).

- **GSS_VerifyMIC**

Once the security context is established, the client can call `GSS_VerifyMICEx()` (section [3.4.12](#)) to verify a signature produced by `GSS_GetMICEx()`.

3.1.5 Message Processing Events and Sequencing Rules

This section specifies how the client processes and returns messages. As discussed earlier, the message transport is provided by the application that is using NTLM.

3.1.5.1 Connection-Oriented

Message processing on the client takes place in the following two cases:

- When the application initiates authentication and the client then sends a [NEGOTIATE_MESSAGE](#).
- When the client needs to process the [CHALLENGE_MESSAGE](#) that it receives from the server and send back an [AUTHENTICATE_MESSAGE](#).

These two cases are described in the following sections.

When encryption is desired, the stream cipher RC4 (as specified in [SCHNEIER]) is used. The key for RC4 is established at the start of the session for an instance of RC4 dedicated to that session. RC4 then continues to generate key stream in order over all packets of the session, without rekeying.

The pseudo-code RC4(handle, message) is defined as the bytes of the message XORed with bytes of the RC4 key stream, using the current state of the session's RC4 internal key state. When the session is torn down, the key structure is destroyed.

The pseudo-code RC4K(key,message) is defined as a one-time instance of RC4 whose key is initialized to key, after which RC4 is applied to the message. On completion of this operation, the internal key state is destroyed.

3.1.5.1.1 Client Initiates the NEGOTIATE_MESSAGE Packet

When the client application initiates the exchange through Security Support Provider Interface (SSPI), the NTLM client sends the [NEGOTIATE_MESSAGE](#) packet to the server, which is embedded in an application protocol message, and encoded according to that application protocol.

The client prepares a NEGOTIATE_MESSAGE and sets the following fields:

- The **Signature** field is set to the string, "NTLMSSP".
- The **MessageType** field is set to NtLmNegotiate.

The client sets the flags specified by the application in the **NegotiateFlags** field in addition to the initialized flags.

If the NTLMSSP_NEGOTIATE_VERSION flag is set by the client application, the **Version** field is set to the current version (as specified in section [2.2.2.9](#)), the **DomainName** field is set to a zero-length string, and the **Workstation** field is set to a zero-length string.

3.1.5.1.2 Client Receives a CHALLENGE_MESSAGE from the Server

When the client receives a [CHALLENGE_MESSAGE](#) from the server, it determines if the features selected by the server are strong enough for the client authentication policy. If not, the authentication process is terminated. Otherwise, the client responds with an [AUTHENTICATE_MESSAGE](#) message.

The client processes the CHALLENGE_MESSAGE and constructs an AUTHENTICATE_MESSAGE per the pseudocode below where all strings are encoded as UNICODE_STRING (as specified in [\[MS-DTYP\]](#) section 2.3.8):

```
-- Input:
-- ClientConfigFlags, User, and UserDom - Defined in section 3.1.1.
-- NtMachineName - The NETBIOS machine name of the server.
-- An NTLM CHALLENGE_MESSAGE whose message fields are defined in
```

```

    section 2.2.1.2.
--
-- Output:
-- ClientHandle - The handle to a key state structure corresponding
-- to the current state of the ClientSealingKey
-- ServerHandle - The handle to a key state structure corresponding
-- to the current state of the ServerSealingKey
-- An NTLM AUTHENTICATE_MESSAGE whose message fields are defined in
-- section 2.2.1.3.
--
-- The following NTLM keys generated by the client are defined in
-- section 3.1.1:
-- ExportedSessionKey, ClientSigningKey, ClientSealingKey,
-- ServerSigningKey, and ServerSealingKey.
--
-- Temporary variables that do not pass over the wire are defined
-- below:
-- KeyExchangeKey, ResponseKeyNT, ResponseKeyLM, SessionBaseKey -
-- Temporary variables used to store 128-bit keys.
-- Time - Temporary variable used to hold the 64-bit time.
--
-- Functions used:
-- NTOWFv1, LMOWFv1, NTOWFv2, LMOWFv2, ComputeResponse - Defined in
-- section 3.3
-- KXKEY, SIGNKEY, SEALKEY - Defined in sections 3.4.5, 3.4.6,
-- and 3.4.7
-- Currenttime, NIL, NONCE - Defined in section 6.

```

The **ClientChallenge** field, as specified in section [2.2.2.4](#) is set to an 8-byte nonce. The **UserName** field is set to User. The **DomainName** field is set to UserDom. The **Signature** field is set to the string, "NTLMSSP". The **MessageType** field is set to NtLmAuthenticate.

If the NTLMSSP_NEGOTIATE_VERSION is set by the client application, the **Version** field is set to the current version, as specified in section [2.2.2.9](#), and the **Workstation** field is set to NbMachineName.

Compute the ResponseKeyNT and ResponseKeyLM using LMOWF v1 and NTOWF v1 functions if NTLM v1 authentication is being used, or compute the ResponseKeyNT and ResponseKeyLM using LMOWF v2 and NTOWF v2 if NTLM v2 authentication is being used.

```

Set Time to Currenttime
Set AUTHENTICATE_MESSAGE.NtChallengeResponse,
AUTHENTICATE_MESSAGE.LmChallengeResponse, SessionBaseKey to
ComputeResponse(CHALLENGE_MESSAGE.NegotiateFlags, ResponseKeyNT,
ResponseKeyLM, CHALLENGE_MESSAGE.ServerChallenge,
AUTHENTICATE_MESSAGE.ClientChallenge, Time,
CHALLENGE_MESSAGE.TargetInfo)

Set KeyExchangeKey to KXKEY(SessionBaseKey, LmChallengeResponse)
If (NTLMSSP_NEGOTIATE_KEY_EXCH bit is set in
CHALLENGE_MESSAGE.NegotiateFlags )
    Set ExportedSessionKey to NONCE(16)
    Set AUTHENTICATE_MESSAGE.EncryptedRandomSessionKey to
    RC4K(KeyExchangeKey, ExportedSessionKey)
Else
    Set ExportedSessionKey to KeyExchangeKey

```

```

    Set AUTHENTICATE_MESSAGE.EncryptedRandomSessionKey to NIL
Endif

Set ClientSigningKey to SIGNKEY(ExportedSessionKey, "Client")
Set ServerSigningKey to SIGNKEY(ExportedSessionKey, "Server")
Set ClientSealingKey to SEALKEY(NegFlg, ExportedSessionKey, "Client")
Set ServerSealingKey to SEALKEY(NegFlg, ExportedSessionKey, "Server")

RC4Init(ClientHandle, ClientSealingKey)
RC4Init(ServerHandle, ServerSealingKey)

```

If NTLM v1 authentication is used, the **LmChallengeResponse** and **NtChallengeResponse** fields of the NTLM AUTHENTICATE_MESSAGE packet are populated by [LM_RESPONSE](#) and [NTLM_RESPONSE](#), respectively. See section [3.3](#) for details.

If NTLMv2 authentication is used, the **LmChallengeResponse** and **NtChallengeResponse** fields of the NTLM AUTHENTICATE_MESSAGE packet are populated by [LMv2_RESPONSE](#) and [NTLMv2_RESPONSE](#), respectively. See section [3.3](#) for details.

When this process is complete, the client sends the AUTHENTICATE_MESSAGE packet to the server, embedded in an application protocol message, and encoded as per that application protocol.

3.1.5.2 Connectionless

The client action for connectionless NTLM authentication is similar to that of connection-oriented authentication, as specified in section [3.1.5.1](#). However, the first packet sent in connectionless authentication is the [CHALLENGE_MESSAGE](#) from the server to the client; there is no client-initiated [NEGOTIATE_MESSAGE](#) as in the connection-oriented authentication.

The message processing for datagram NTLM authentication is as specified in the following sections.

3.1.5.2.1 Client Receives a CHALLENGE_MESSAGE

When the client receives a [CHALLENGE_MESSAGE](#), it produces a challenge response and an encrypted session key. The client sends the negotiated features (flags), the user name, the user's domain, the client part of the challenge, the challenge response, and the encrypted session key to the server. This message is sent to the server as an AUTHENTICATE_MESSAGE.

If NTLM v1 authentication is used, the **LmChallengeResponse** and **NtChallengeResponse** fields of the NTLM [AUTHENTICATE_MESSAGE](#) packet are populated by [LM_RESPONSE](#) and [NTLM_RESPONSE](#), respectively.

If NTLM v2 authentication is used, the **LmChallengeResponse** and **NtChallengeResponse** fields of the NTLM AUTHENTICATE_MESSAGE packet are populated by [LMv2_RESPONSE](#) and [NTLMv2_RESPONSE](#), respectively.

3.1.5.2.2 Server Receives an AUTHENTICATE_MESSAGE

When the server receives an [AUTHENTICATE_MESSAGE](#), it checks the signature and/or verifies the response. If the response is verified, the client has authenticated to the server.

3.1.6 Timer Events

No timer events are used.

3.1.7 Other Local Events

No other local events are used.

3.2 Server Details

3.2.1 Abstract Data Model

The following sections specify variables that are internal to the server and are maintained across the NTLM authentication sequence.

3.2.1.1 Variables Internal to the Protocol

The server maintains all of the variables except ClientConfigFlags (as specified in section [3.1.1.2](#)), as well as those defined in this section.

CfgFlg: The set of server configuration flags (as specified in section [2.2.2.5](#)) that specify the full set of capabilities of the server.

DnsDomainName: A string that indicates the DNS domain name of the server's domain.

DnsForestName: A string that indicates the DNS name of the server's **forest**.

DnsMachineName: A string that indicates the DNS machine name of the server.

NbDomainName: A string that indicates the NetBIOS name of the user's domain.

NbMachineName: A string that indicates the NetBIOS machine name of the server.

3.2.1.2 Variables Exposed to the Application

Datagram: The Datagram option indicates that the connectionless mode of NTLM is to be used. If the Datagram option is selected by the server, connectionless mode is used, and NTLM performs a bitwise OR operation with the following NTLM Negotiate Flags into the CfgFlgs:

- NTLMSSP_NEGOTIATE_DATAGRAM

3.2.2 Timers

No timers are used by the server.

3.2.3 Initialization

The sequence number is set to zero.

3.2.4 Higher-Layer Triggered Events

The application client initiates NTLM authentication through the Security Support Provider Interface (SSPI), the Microsoft implementation of GSS-API [\[RFC2743\]](#).

- GSS_Init_sec_context

The client calls `GSS_Init_sec_context()` to establish a security context with the server. NTLM has no requirements on which flags are used and will simply honor what was requested by the application or protocol. For an example of such a protocol specification, see [\[MS-RPCE\]](#) section 3.3.1.5.2.2, Using a Security Context. The application will send the `NEGOTIATE_MESSAGE`, as specified in section [2.2.1.1](#), to the server as specified in section [3.1.5.1.1](#) Client Initiates the `NEGOTIATE_MESSAGE` Packet.

When the client receives the `CHALLENGE_MESSAGE`, as specified in section [2.2.1.2](#), from the server, the client will call `GSS_Init_sec_context()` with the `CHALLENGE_MESSAGE` as input. The application will send the `AUTHENTICATE_MESSAGE`, as specified in section [2.2.1.3](#), to the server as specified in section [3.1.5.1.2](#) Client Receives a `CHALLENGE_MESSAGE` from the Server.

- `GSS_Wrap`

Once the security context is established, the client can call `GSS_WrapEx()` (section [3.4.9](#)) to encrypt messages.

- `GSS_Unwrap`

Once the security context is established, the client can call `GSS_UnwrapEx()` (section [3.4.10](#)) to decrypt messages that were encrypted by `GSS_WrapEx`.

- `GSS_GetMIC`

Once the security context is established, the client can call `GSS_GetMICEx()` (section [3.4.11](#)) to sign messages, producing an `NTLMSSP_MESSAGE_SIGNATURE` structure whose fields are defined in section [2.2.1.4](#).

- `GSS_VerifyMIC`

Once the security context is established, the client can call `GSS_VerifyMICEx()` (section [3.4.12](#)) to verify a signature produced by `GSS_GetMICEx()`.

3.2.5 Message Processing Events and Sequencing Rules

The server-side processing of messages can happen in response to two different messages from the client:

- The server receives a [NEGOTIATE_MESSAGE](#) from the client (the server responds with a [CHALLENGE_MESSAGE](#)).
- The server receives an [AUTHENTICATE_MESSAGE](#) from the client (the server verifies the client's authentication information that is embedded in the message).

3.2.5.1 Connection-Oriented

Message processing on the sever takes place in the following two cases:

- Upon receipt of the embedded [NEGOTIATE_MESSAGE](#) packet, the server extracts and decodes the `NEGOTIATE_MESSAGE`.
- Upon receipt of the embedded [AUTHENTICATE_MESSAGE](#) packet, the server extracts and decodes the `AUTHENTICATE_MESSAGE`.

These two cases are described in the following sections.

3.2.5.1.1 Server Receives a NEGOTIATE_MESSAGE from the Client

Upon receipt of the embedded [NEGOTIATE_MESSAGE](#) packet, the server extracts and decodes the NEGOTIATE_MESSAGE.

If the security features selected by the client are not strong enough for the server security policy, the authentication process is terminated. Otherwise, the server responds with a [CHALLENGE_MESSAGE](#) message. This includes the negotiated features and a 64-bit (8-byte) nonce value for the ServerChallenge value. The nonce is a pseudo-random number generated by the server and intended for one-time use. The flags returned as part of the CHALLENGE_MESSAGE in this step indicate which variant the server wants to use and whether the server's domain name or machine name are present in the **TargetName** field.

The server processes the NEGOTIATE_MESSAGE and constructs a CHALLENGE_MESSAGE per the following pseudocode where all strings are encoded as UNICODE_STRING (as specified in [\[MS-DTYP\]](#) section 2.3.8):

```
-- Input:
--   CfgFlg - Defined in section 3.2.1.
--   An NTLM NEGOTIATE_MESSAGE whose message fields are defined in
--       section 2.2.1.1.
--
-- Output:
--   An NTLM CHALLENGE_MESSAGE whose message fields are defined in
--       section 2.2.1.2.
--
-- Functions used:
--   AddAVPair(), NIL, NONCE - Defined in section 6.
```

Set all of the following flags in the CHALLENGE_MESSAGE **NegotiateFlags** field:

```
CfgFlg,
NEGOTIATE_MESSAGE.NegotiateFlags,
NTLMSSP_REQUEST_TARGET,
NTLMSSP_NEGOTIATE_NTLM,
NTLMSSP_NEGOTIATE_ALWAYS_SIGN
```

The **Signature** field is set to the string, "NTLMSSP". The **MessageType** field is set to 0x00000002, indicating a message type of NtLmChallenge. The **ServerChallenge** field is set to an 8-byte nonce.

If the NTLMSPP_NEGOTIATE_VERSION flag is set, the **Version** field is set to the current version, as specified in section [2.2.2.9](#).

```
If (NTLMSSP_NEGOTIATE_UNICODE is set in NEGOTIATE.NegotiateFlags)
    Set the NTLMSPP_NEGOTIATE_UNICODE flag in
    CHALLENGE_MESSAGE.NegotiateFlags
ElseIf (NTLMSSP_NEGOTIATE_OEM flag is set in NEGOTIATE.NegotiateFlag)
    Set the NTLMSPP_NEGOTIATE_OEM flag in
    CHALLENGE_MESSAGE.NegotiateFlags
EndIf
if (NTLMSSP_NEGOTIATE_NTLM2 flag is set in NEGOTIATE.NegotiateFlag)
    Set the NTLMSPP_NEGOTIATE_NTLM2 flag in
    CHALLENGE_MESSAGE.NegotiateFlags
ElseIf(NTLMSSP_NEGOTIATE_LM_KEY flag is set in
```

```

NEGOTIATE.NegotiateFlag)
    Set the NTLMSSP_NEGOTIATE_LM_KEY flag in
    CHALLENGE_MESSAGE.NegotiateFlags
EndIf

If (DomainJoined is non-zero)
    Set CHALLENGE_MESSAGE.TargetName to NbDomainName
    Set the NTLMSSP_NEGOTIATE_TARGET_DOMAIN flag in
    CHALLENGE_MESSAGE.NegotiateFlags
Else
    Set CHALLENGE_MESSAGE.TargetName to NbMachineName
    Set the NTLMSSP_NEGOTIATE_TARGET_SERVER flag in
    CHALLENGE_MESSAGE.NegotiateFlags
EndIf
If (NTLMSSP_NEGOTIATE_NTLM2 flag is set in
CHALLENGE_MESSAGE.NegotiateFlags)
    Set the NTLMSSP_NEGOTIATE_TARGET_INFO flag in
    CHALLENGE_MESSAGE.NegotiateFlags
EndIf
If (NbMachineName is not NIL)
    AddAvPair(TargetInfo, MsvAvNbComputerName, NbMachineName)
EndIf
If (NbDomainName is not NIL)
    AddAvPair(TargetInfo, MsvAvNbDomainName, NbDomainName)
EndIf
If (DnsMachineName is not NIL)
    AddAvPair(TargetInfo, MsvAvDnsComputerName, DnsMachineName)
EndIf
If (DnsDomainName is not NIL)
    AddAvPair(TargetInfo, MsvAvDnsDomainName, DnsDomainName)
EndIf
If (DnsForestName is not NIL)
    AddAvPair(TargetInfo, MsvAvDnsTreeName, DnsForestName)
EndIf
AddAvPair(TargetInfo, MsvAvEOL, NIL)

```

When this process is complete, the server sends the CHALLENGE_MESSAGE packet to the client, embedded in an application protocol message, and encoded according to that application protocol.

3.2.5.1.2 Server Receives an AUTHENTICATE_MESSAGE from the Client

Upon receipt of the embedded [AUTHENTICATE_MESSAGE](#) packet, the server extracts and decodes the AUTHENTICATE_MESSAGE.

The server obtains the **response key** by looking up the user name in a database. With the NT and LM responses key and the client challenge, the server computes the expected response. If the expected response matches the actual response, then the server generates session, signing, and sealing keys; otherwise, it denies the client access.

The keys are computed with the following algorithm where all strings are encoded as UNICODE_STRING (as specified in [\[MS-DTYP\]](#) section 2.3.8):

```

-- Input:
-- CHALLENGE_MESSAGE.ServerChallenge - The ServerChallenge field
-- from the server CHALLENGE_MESSAGE in section 3.2.5.1.1
-- NegFlg - Defined in section 3.1.1.

```

```

-- ServerName - The NETBIOS or the DNS name of the server.
-- An NTLM AUTHENTICATE_MESSAGE whose message fields are defined
-- in section 2.2.1.3.
--
-- Output:      Result of authentication
-- ClientHandle - The handle to a key state structure corresponding
-- to the current state of the ClientSealingKey
-- ServerHandle - The handle to a key state structure corresponding
-- to the current state of the ServerSealingKey
-- The following NTLM keys generated by the server are defined in
-- section 3.1.1:
-- ExportedSessionKey, ClientSigningKey, ClientSealingKey,
-- ServerSigningKey, and ServerSealingKey.
--
-- Temporary variables that do not pass over the wire are defined
-- below:
-- KeyExchangeKey, ResponseKeyNT, ResponseKeyLM, SessionBaseKey -
-- Temporary variables used to store 128-bit keys.
-- Time - Temporary variable used to hold the 64-bit current time in
-- the AUTHENTICATE_MESSAGE.ClientChallenge, in the format of a
-- FILETIME as defined in [MS-DTYP] section 2.3.1.
-- ExpectedNtChallengeResponse - Temporary variable to hold results
-- returned from Compute Response.
-- ExpectedLmChallengeResponse - Temporary variable to hold results
-- returned from Compute Response.
--
-- Functions used:
-- ComputeResponse - Defined in section 3.3
-- KXKEY, SIGNKEY, SEALKEY - Defined in sections 3.4.5, 3.4.6, and
-- 3.4.7
-- GetVersion(), NIL - Defined in section 6

```

Retrieve the ResponseKeyNT and ResponseKeyLM from the local user account database using the UserName and DomainName specified in the AUTHENTICATE_MESSAGE.

```

Set ExpectedNtChallengeResponse, ExpectedLmChallengeResponse,
SessionBaseKey to ComputeResponse(NegFlg, ResponseKeyNT,
ResponseKeyLM, CHALLENGE_MESSAGE.ServerChallenge,
AUTHENTICATE_MESSAGE.ClientChallenge, Time, ServerName)

```

```

Set KeyExchangeKey to KXKEY(SessionBaseKey,
AUTHENTICATE_MESSAGE.LmChallengeResponse)
If (AUTHENTICATE_MESSAGE.NtChallengeResponse is NOT EQUAL to
ExpectedNtChallengeResponse)
  If AUTHENTICATE_MESSAGE.LmChallengeResponse !=
    ExpectedLmChallengeResponse
    Return INVALID message error
  EndIf
EndIf
If (NTLMSSP_NEGOTIATE_KEY_EXCH flag is set in NegFlg )
  Set ExportedSessionKey to RC4K(KeyExchangeKey,
  AUTHENTICATE_MESSAGE.EncryptedRandomSessionKey)
Else
  Key exchange key is not available
EndIf
Set ClientSigningKey to SIGNKEY(ExportedSessionKey , "Client")
Set ServerSigningKey to SIGNKEY(ExportedSessionKey , "Server")

```

```
Set ClientSealingKey to SEALKEY(NegFlg, ExportedSessionKey , "Client")
Set ServerSealingKey to SEALKEY(NegFlg, ExportedSessionKey , "Server")
```

```
RC4Init(ClientHandle, ClientSealingKey)
RC4Init(ServerHandle, ServerSealingKey)
```

Both the client and the server now have the session, signing, and sealing keys. When the client runs an integrity check on the next message from the server, it is aware that the server knows (either directly or indirectly) the user password.

3.2.5.2 Connectionless NTLM

3.2.5.2.1 Server Sends the Client an Initial CHALLENGE_MESSAGE Packet

The server sends a set of supported features, and a random key to use as part of the challenge. This key is in the form of a 64-bit (8-byte) nonce value for the ServerChallenge value. The nonce is a pseudo-random number generated by the server and intended for one-time use. The connectionless variant always uses key exchange, so the NTLMSSP_NEGOTIATE_KEY_EXCH flag MUST be set in the required flags mask. The client determines the set of supported features, and whether those meet minimum security requirements or not. This message is sent to the client as a [CHALLENGE_MESSAGE](#).

3.2.5.2.2 Server Response Checking

The client computes the expected session key for signing and encryption, which it sends to the server in the AUTHENTICATE_MESSAGE as specified in section [3.1.5.2.1](#). Using this key from the AUTHENTICATE_MESSAGE, the server checks the signature and/or decrypts the protocol response sent in step 3, and computes a response. The response is signed and/or encrypted and sent to the client.

3.2.6 Timer Events

No timer events are used.

3.2.7 Other Local Events

No other local events are defined.

3.3 NTLM v1 and NTLM v2 Messages

This section provides further details about how the client and server compute the responses depending on whether NTLM v1 or NTLM v2 is used. It also includes details about the NTOWF and LMOWF functions whose output is subsequently used to compute the response.

3.3.1 NTLM v1 Authentication

The following pseudocode defines the details of the algorithms used to calculate the keys used in NTLM v1 authentication.

Note The NTLM authentication version is not negotiated by the protocol. It must be configured on both the client and the server prior to authentication. The NTOWF v1 and LMOWF v1 functions defined in this section are NTLM version-dependent and are used only by NTLM v1.

The NT and LM response keys are encoded using the following specific one-way functions where all strings are encoded as UNICODE_STRING (as specified in [\[MS-DTYP\]](#) section 2.3.8):

```
-- Explanation of message fields and variables:
-- ClientChallenge - The 8-byte challenge message generated by
-- the client.
-- LmChallengeResponse - The LM response to the server challenge.
-- Computed by the client.
-- NegFlg, User, UserDom - Defined in section 3.1.1.
-- NTChallengeResponse - The NT response to the server challenge.
-- Computed by the client.
-- Password - Password of the user. If the password is longer than
-- 14 characters, then the LMOWF v1 cannot be computed. For LMOWF
-- v1, if the password is shorter than 14 characters, it is padded
-- by appending zeroes.
-- ResponseKeyNT - Temporary variable to hold the results of
-- calling NTOWF.
-- ResponseKeyLM - Temporary variable to hold the results of
-- calling LMGETKEY.
-- CHALLENGE_MESSAGE.ServerChallenge - The 8-byte challenge message
-- generated by the server.
--
-- Functions Used:
-- Z(M)- Defined in section 6.
Define NTOWFv1(Passwd, User, UserDom) as MD4(UNICODE(Passwd))
EndDefine
Define LMOWFv1(Passwd, User, UserDom) as
    ConcatenationOf( DES( UpperCase( Passwd) [0..6], "KGS!@#$$%"),
        DES( UpperCase( Passwd) [7..13], "KGS!@#$$%"))
EndDefine
Set ResponseKeyNT to NTOWFv1(Passwd, User, UserDom)
Set ResponseKeyLM to LMOWFv1( Passwd, User, UserDom )

Define ComputeResponse(NegFlg, ResponseKeyNT, ResponseKeyLM,
CHALLENGE_MESSAGE.ServerChallenge, ClientChallenge, Time, ServerName)
as
If (NTLMSSP_NEGOTIATE_NTLM2 flag is set in NegFlg)
    Set NtChallengeResponse to DESL(ResponseKeyNT,
MD5(ConcatenationOf(CHALLENGE_MESSAGE.ServerChallenge,
ClientChallenge)) [0..7])
    Set LmChallengeResponse to ConcatenationOf(ClientChallenge,
Z(16))
Else
    Set NtChallengeResponse to DESL(ResponseKeyNT,
CHALLENGE_MESSAGE.ServerChallenge)
    If (NTLMSSP_NEGOTIATE_NT_ONLY flag is set in NegFlg)
        Set LmChallengeResponse to NtChallengeResponse
    Else
        Set LmChallengeResponse to DESL(ResponseKeyLM,
CHALLENGE_MESSAGE.ServerChallenge)
    EndIf
EndIf

If (NTLMSSP_NEGOTIATE_NTLM2 flag is set in NegFlg)
```

```

    Send the following as the challenge:
    MD5(ConcatenationOf(CHALLENGE_MESSAGE.ServerChallenge,
    ClientChallenge))[0..7])
EndIf
EndDefine

Set SessionBaseKey to MD4(NTOWF)

```

On the server, if the user account to be authenticated is hosted in Active Directory (AD), the challenge-response pair is sent to the domain controller (DC) to verify, as specified in [\[MS-APDS\]](#) section 3.1.5.

The DC calculates the expected value of the response using the NTOWF v1 and/or LMOWF v1, and matches it against the response provided. If the response values match, it sends back the KeyExchangeKey; otherwise, it returns a wrong-password error. The server then rejects the client authentication if the DC returns an error.

If the user account to be authenticated is hosted locally on the server, the server calculates the expected value of the response using the NTOWF v1 and/or LMOWF v1 stored locally, and matches it against the response provided. If the response values match, it calculates KeyExchangeKey; otherwise, it returns a wrong-password error and rejects the client authentication. [<28>](#)

3.3.2 NTLM v2 Authentication

The following pseudo code defines the details of the algorithms used to calculate the keys used in NTLM v2 authentication.

Note The NTLM authentication version is not negotiated by the protocol. It must be configured on both the client and the server prior to authentication. The NTOWF v2 and LMOWF v2 functions defined in this section are NTLM version-dependent and are only used by NTLM v2.

The NT and LM response keys are encoded using the following specific one-way functions where all strings are encoded as UNICODE_STRING (as specified in [\[MS-DTYP\]](#) section **2.3.9**):

```

-- Explanation of message fields and variables:
-- NegFlg, User, UserDom - Defined in section 3.1.1.
-- Password - Password of the user.
-- LmChallengeResponse - The LM response to the server challenge.
    Computed by the client.
-- NTChallengeResponse - The NT response to the server challenge.
    Computed by the client.
-- ClientChallenge - The 8-byte challenge message generated by the
    client.
-- CHALLENGE_MESSAGE.ServerChallenge - The 8-byte challenge message
    generated by the server.
-- ResponseKeyNT - Temporary variable to hold the results of
    calling NTOWF.
-- ResponseKeyLM - Temporary variable to hold the results of
    calling LMGETKEY.
-- ServerName - The TargetInfo field structure of the
    CHALLENGE_MESSAGE payload.
-- KeyExchangeKey - Temporary variable to hold the results of
    calling KXKEY.
-- HiResponseVersion - The 1-byte highest response version
    understood by the client. Currently set to 1.

```

```

-- Responserverversion - The 1-byte response version. Currently set
to 1.
-- Time - The 8-byte little-endian time in GMT.
--
-- Functions Used:
-- Z(M) - Defined in section 6.

Define NTOWFv2(Passwd, User, UserDom) as HMAC_MD5(
MD4(Unicode(Passwd)), ConcatenationOf( Uppercase(User),
UserDom ) )
EndDefine

Define LMOWFv2(Passwd, User, UserDom) as NTOWFv2(Passwd, User,
UserDom)
EndDefine

Set ResponseKeyNT to NTOWFv2(Passwd, User, UserDom)
Set ResponseKeyLM to LMOWFv2(Passwd, User, UserDom)

Define ComputeResponse(NegFlg, ResponseKeyNT, ResponseKeyLM,
CHALLENGE_MESSAGE.ServerChallenge, ClientChallenge, Time, ServerName)
as
Set temp to ConcatenationOf(Responserverversion, HiResponserverversion,
Z(6), Time, ClientChallenge, Z(4), ServerName, Z(4))
Set NTProofStr to HMAC_MD5(ResponseKeyNT,
ConcatenationOf(CHALLENGE_MESSAGE.ServerChallenge,temp))
Set NtChallengeResponse to ConcatenationOf(NTProofStr, temp)
Set LmChallengeResponse to ConcatenationOf(HMAC_MD5(ResponseKeyLM,
ConcatenationOf(CHALLENGE_MESSAGE.ServerChallenge, ClientChallenge)),
ClientChallenge )

Set SessionBaseKey to HMAC_MD5(ResponseKeyNT, NTProofStr)
EndDefine

```

On the server, if the user account to be authenticated is hosted in Active Directory (AD), the challenge-response pair is sent to the DC to verify, as specified in [\[MS-APDS\]](#).

The DC calculates the expected value of the response using the NTOWF and/or LMOWF, and matches it against the response provided. If the response values match, it sends back the KeyExchangeKey; otherwise, it returns a wrong-password error. The server rejects the client authentication if the DC returns an error.

If the user account to be authenticated is hosted locally on the server, the server calculates the expected value of the response using the NTOWF and/or LMOWF stored locally, and matches it against the response provided. If the response values match, it calculates KeyExchangeKey; otherwise, it returns a wrong-password error and rejects the client authentication. [<29>](#)

3.4 Session Security Details

If it is negotiated, session security provides message integrity (signing) and message confidentiality (sealing). When NTLM v2 authentication is not negotiated, only one key is used for sealing. As a result, operations are performed in a half-duplex mode: the client sends a message and then waits for a server response. For information on how key exchange, signing, and sealing keys are generated, see [KXKEY](#), [SIGNKEY](#), and [SEALKEY](#).

In connection-oriented mode, messages are assumed to be received in the order sent. The application or communications protocol is expected to guarantee this property. As a result, the client and server sealing keys are computed only once per session.

Note In datagram (connectionless) mode, messages can arrive out of order. Because of this, the sealing key must be reset for every message. Re-keying with the same sealing key for multiple messages would not maintain message security. Therefore, a per-message sealing key, `SealingKey'`, is computed as the MD5 hash of the original sealing key and the packet sequence number. The resulting `SealingKey'` value is used to re-initialize the key state structure prior to invoking the SIGN, SEAL, and MAC algorithms below. To compute the `SealingKey'` and initialize the key state structure identified by the `Handle` parameter, use:

```
SealingKey' = MD5(ConcatenationOf(SealingKey, SequenceNumber))
RC4Init(Handle, SealingKey')
```

3.4.1 Abstract Data Model

The following define the services provided by the NTLM SSP.

Note The variables defined below are logical, abstract parameters that an implementation has to maintain and expose to provide the proper level of Service. How these variables are maintained and exposed is up to the implementation.

- **Integrity:** Indicates that the caller wishes to construct signed messages so that they cannot be tampered with while in transit. If the client requests integrity, then the server MUST respond with integrity if supported or MUST NOT respond with integrity if not supported.
- **Sequence Detect:** Indicates that the caller wishes to construct signed messages such that out-of-order sequences can be detected. For more information, see section [3.4.2](#) Message Integrity.
- **Confidentiality:** Indicates that the caller wishes to encrypt messages such that they cannot be read while in transit. If the client requests confidentiality, then the server MUST respond with confidentiality if supported or MUST NOT respond with confidentiality if not supported.

Usage of integrity and confidentiality is the responsibility of the application:

- If confidentiality is established, then the application MUST call `GSS_Wrap()` to invoke confidentiality with the NTLM SSP. For more information, see section [3.4.3](#) Message Confidentiality.
- If integrity is established, then the application MUST call `GSS_GetMIC()` to invoke integrity with the NTLM SSP. For more information, see section [3.4.2](#) Message Integrity.

3.4.2 Message Integrity

Message integrity is available only if NTLM v2 session security is negotiated. If it is negotiated, the function to sign a message is calculated as follows.

```
-- Input:
--   SigningKey - The key used to sign the message.
--   Message - The message being sent between the client and server.
--   SeqNum - Defined in section 3.1.1.
--   Handle - The handle to a key state structure corresponding to
```

```

-- the current state of the SealingKey
--
-- Output:      Signed message
-- Functions used:
-- ConcatenationOf() - Defined in Section 6.
-- MAC() - Defined in Section 3.4.3.

Define SIGN(Handle, SigningKey, SeqNum, Message) as
ConcatenationOf(Message, MAC(Handle, SigningKey, SeqNum, Message))
EndDefine

```

The format of the message integrity data that is appended to each message for signing and sealing purposes is defined by the NTLMSPP_MESSAGE_SIGNATURE structure (section [2.2.1.4](#)).

Note If the client is sending the message, the signing key is the one that the client calculated. If the server is sending the message, the signing key is the one that the server calculated. The same is true for the sealing key. The sequence number can be explicitly provided by the application protocol or by the NTLM security service provider. If the latter is chosen, the sequence number is initialized to zero and then incremented by one for each message sent.

On receipt, the message authentication code (MAC) value is computed and compared with the received value. If they differ, the message is discarded (as specified in section [3.4.4](#)).

3.4.3 Message Confidentiality

Message confidentiality, if it is negotiated, also implies message integrity. If message confidentiality is negotiated, a sealed (and implicitly signed) message is sent instead of a signed or unsigned message. The function that seals a message using the signing key, sealing key, and message sequence number is as follows.

```

-- Input:
-- SigningKey - The key used to sign the message.
-- Message - The message being sent between the client and server.
-- NegFlg, SeqNum - Defined in section 3.1.1.
-- Handle - The handle to a key state structure corresponding to the
-- current state of the SealingKey
--
-- Output:      Sealed message
--
-- Functions used:
-- ConcatenationOf() - Defined in Section 6.
-- RC4() - Defined in Section 6 and 3.1.
-- CRC32() - Defined in Section 6.

If (NTLMSSP_NEGOTIATE_NTLM2 flag is set in NegFlg)

    Define SEAL(Handle, SigningKey, SeqNum, Message) as
        ConcatenationOf(RC4(Handle, Message), MAC(Handle, SigningKey,
            SeqNum, Message))
    EndDefine

Else

    Define SEAL(Handle, SigningKey, SeqNum, Message) as
        ConcatenationOf(RC4(Handle, Message), 0x00000001,

```

```

RC4(Handle, ConcatenationOf(NONCE(4),
CRC32(ConcatenationOf(SeqNum, Message)), SeqNum))
EndDefine

```

Message confidentiality is available in datagram mode only if the client configures NTLM v2 session security.

3.4.4 Message Signature Functions

When session security is negotiated, the message signature for NTLM v2 is a 16-byte value that contains the following components, as described by the NTLMSSP_MESSAGE_SIGNATURE structure:

- A 4-byte version-number value that is set to 1.
- The first eight bytes of the message's HMAC_MD5.
- The 4-byte sequence number (*SeqNum*).

In the case of connectionless NTLM authentication, the *SeqNum* parameter is specified by the application and the RC4 (as specified in [SCHNEIER]) stream is reinitialized before each message, as specified in section 3.4.

In the case of connection-oriented authentication, the *SeqNum* parameter starts at 0 and is incremented by one for each message sent. The receiver expects the first received message to have *SeqNum* equal to 0, and to be one greater for each subsequent message received. If a received message does not contain the expected *SeqNum*, an error is returned to the receiving application, and *SeqNum* is not incremented.

If message integrity is negotiated, the message signature is calculated as follows.

```

-- Input:
--   SigningKey - The key used to sign the message.
--   SealingKey - The key used to seal the message or checksum.
--   Message - The message being sent between the client and server.
--   SeqNum - Defined in section 3.1.1.
--   Handle - The handle to a key state structure corresponding to the
--            current state of the SealingKey
--
-- Output:
--   An NTLMSSP_MESSAGE_SIGNATURE structure whose fields are defined
--   in section 2.2.1.4.
--   SeqNum - Defined in section 3.1.1.
--
-- Functions used:
--   ConcatenationOf() - Defined in Section 6.
--   RC4() - Defined in Section 6.
--   CRC32() - Defined in Section 6.

Define MAC(Handle, SigningKey, SeqNum, Message) as
  Set NTLMSSP_MESSAGE_SIGNATURE.Version to 0x00000001
  Set NTLMSSP_MESSAGE_SIGNATURE.Checksum to HMAC_MD5(SigningKey,
ConcatenationOf(SeqNum, Message))[0..7]
  Set NTLMSSP_MESSAGE_SIGNATURE.SeqNum to SeqNum
  Set SeqNum to SeqNum + 1
EndDefine

```

If a **key exchange key** is negotiated, the message signature for the NTLM security service provider is the same as above, except the 8 bytes of the HMAC_MD5 are encrypted with RC4, as follows:

```
Define MAC(Handle, SigningKey, SeqNum, Message) as
    Set NTLMSSP_MESSAGE_SIGNATURE.Version to 0x00000001
    Set NTLMSSP_MESSAGE_SIGNATURE.Checksum to RC4(Handle,
        HMAC_MD5(SigningKey, ConcatenationOf(SeqNum, Message))[0..7])
    Set NTLMSSP_MESSAGE_SIGNATURE.SeqNum to SeqNum
    Set SeqNum to SeqNum + 1
EndDefine
```

Message integrity (signing) is available only when NTLM v2 session security is negotiated.

3.4.5 KXKEY, SIGNKEY, and SEALKEY

This topic specifies how key exchange ([KXKEY](#)), signing ([SIGNKEY](#)), and sealing ([SEALKEY](#)) keys are generated.

3.4.6 SKXKEY

The choice of a key exchange key does not depend on whether NTLM v2 is configured or not. If NTLM v2 session security is negotiated, the key exchange key is the 128-bit session base key.

If NTLM v2 session security is not negotiated, the 128-bit key exchange key value is calculated as follows:

```
-- Input:
--   SessionBaseKey - A session key calculated from the user's
--                   password.
--   LmChallengeResponse - The LM response to the server challenge.
--                       Computed by the client.
--   NegFlg - Defined in section 3.1.1.
--
-- Output:
--   KeyExchangeKey - The Key Exchange Key.
--
-- Functions used:
--   ConcatenationOf() - Defined in Section 6.
--   DES() - Defined in Section 6.

Define KXKEY(SessionBaseKey, LmChallengeResponse) as
If ( NTLMSSP_NEGOTIATE_LMKEY flag is set in NegFlg)
    Set KeyExchangeKey to ConcatenationOf(DES(SessionBaseKey[0..6],
        LmChallengeResponse[0..7]),
        DES(ConcatenationOf(SessionBaseKey[7], 0xBDBDBDBDBDBD),
        LmChallengeResponse[0..7]))
Else
    Set KeyExchangeKey to SessionBaseKey
Endif
EndDefine
```

3.4.7 SIGNKEY

If NTLM v2 session security is not negotiated, signing is not supported and the signing key is a zero-length string.

If NTLM v2 session security is negotiated, the signing key is a 128-bit value that is calculated as follows from the random session key and the null-terminated ASCII constants shown.

```
-- Input:
-- RandomSessionKey - A randomly generated session key.
-- NegFlg - Defined in section 3.1.1.
-- Mode - An enum that defines the local machine performing
-- the computation.
-- Mode always takes the value "Client" or "Server.
--
-- Output:
-- SignKey - The key used for signing messages.
--
-- Functions used:
-- ConcatenationOf(), MD5(), NIL - Defined in Section 6.

Define SIGNKEY(NegFlg, RandomSessionKey, Mode) as
If (NTLMSSP_NEGOTIATE_NTLM2 flag is set in NegFlg)
  If (Mode equals "Client")
    Set SignKey to MD5(ConcatenationOf(RandomSessionKey,
    "session key to client-to-server signing key magic
    constant"))
  Else
    Set SignKey to MD5(ConcatenationOf(RandomSessionKey,
    "session key to server-to-client signing key magic
    constant"))
  Endif
Else
  Set SignKey to NIL
Endif
EndDefine
```

3.4.8 SEALKEY

The sealing key function produces an encryption key from the random session key and the null-terminated ASCII constants shown.

- If NTLM v2 session security is negotiated, the sealing key has either 40, 56, or 128 bits of entropy stored in a 128-bit value.
- If NTLM v2 session security is not negotiated, the sealing key has either 40 or 56 bits of entropy stored in a 64-bit value.

Note The MD5 hashes completely overwrite and fill the 64-bit or 128-bit value.

```
-- Input:
-- RandomSessionKey - A randomly generated session key.
-- NegFlg - Defined in section 3.1.1.
-- Mode - An enum that defines the local machine performing
-- the computation.
-- Mode always takes the value "Client" or "Server.
```

```

--
-- Output:
--   SealKey - The key used for sealing messages.
--
-- Functions used:
--   ConcatenationOf(), MD5() - Defined in Section 6.

Define SEALKEY(NegotiateFlags, RandomSessionKey, Mode) as
If (NTLMSSP_NEGOTIATE_NTLM2 flag is set in NegFlg)
  If ( NTLMSSP_NEGOTIATE_128 is set in NegFlg)
    Set SealKey to RandomSessionKey
  ElseIf ( NTLMSSP_NEGOTIATE_56 flag is set in NegFlg)
    Set SealKey to RandomSessionKey[0..6]
  Else
    Set SealKey to RandomSessionKey[0..4]
  Endif
Endif
If (Mode equals "Client")
  Set SealKey to MD5(ConcatenationOf(SealKey, "session key to
client-to-server sealing key magic constant"))
Else
  Set SealKey to MD5(ConcatenationOf(SealKey, "session key to
server-to-client sealing key magic constant"))
Endif
ElseIf (NTLMSSP_NEGOTIATE_56 flag is set in NegFlg)
  Set SealKey to ConcatenationOf(RandomSessionKey[0..6], 0xA0)
Else
  Set SealKey to ConcatenationOf(RandomSessionKey[0..4], 0xE5,
0x38, 0xB0)
Endif
Endif
EndDefine

```

3.4.9 GSS_WrapEx() Call

This call is an extension to **GSS_Wrap** [\[RFC2743\]](#) that passes multiple buffers. The Microsoft implementation of **GSS_WrapEx()** is called **EncryptMessage()**. For more information, see [\[MSDN-EncryptMsg\]](#).

Inputs:

- context_handle CONTEXT HANDLE
- qop_req INTEGER, -- 0 specifies default QOP
- input_message ORDERED LIST of:
 - conf_req_flag BOOLEAN
 - sign BOOLEAN
 - data OCTET STRING

Outputs:

- major_status INTEGER

- minor_status INTEGER
- output_message ORDERED LIST (in same order as input_message) of:
 - conf_state BOOLEAN
 - signed BOOLEAN
 - data OCTET STRING
- signature OCTET STRING

This call is identical to **GSS_Wrap**, except that it supports multiple input buffers.

The input data can be a list of security buffers. The caller can request encryption by setting **fQOP** to 0. If the caller requests just signing the input data messages and no encryption will be performed, it sets the **fQOP** parameter as SECQOP_WRAP_NO_ENCRYPT (0x80000001).

Input data buffers for which conf_req_flag==TRUE are encrypted (section [3.4.3](#), Message Confidentiality) in output_message.

For NTLMv1, input data buffers for which sign==TRUE are included in the message signature. For NTLMv2, all input data buffers are included in the message signature, as specified in section [3.4.9.1](#).

3.4.9.1 Signature Creation for GSS_WrapEx()

Section [3.4.3](#) describes the algorithm used by GSS_WrapEx() to create the signature. The signature contains the NTLMSSP_MESSAGE_SIGNATURE structure, as specified in section [2.2.1.4](#).

The checksum is computed over the concatenated input buffers, as specified in section [3.1.5.2.2](#), using only the input data buffers where sign==TRUE for NTLMv1 and all of the input data buffers for NTLMv2, including the cleartext data buffers.

3.4.10 GSS_UnwrapEx() Call

This call is an extension to GSS_Unwrap [\[RFC2743\]](#) that passes multiple buffers. The Microsoft implementation of GSS_WrapEx() is called **DecryptMessage()**. For more information, see [\[MSDN-DecryptMsg\]](#).

Inputs:

- context_handle CONTEXT HANDLE
- input_message ORDERED LIST of:
 - conf_state BOOLEAN
 - signed BOOLEAN
 - data OCTET STRING
- signature OCTET STRING

Outputs:

- qop_req INTEGER, -- 0 specifies default QOP
- major_status INTEGER

- minor_status INTEGER
- output_message ORDERED LIST (in same order as input_message) of:
 - conf_state BOOLEAN
 - data OCTET STRING

This call is identical to **GSS_Unwrap**, except that it supports multiple input buffers. Input data buffers having conf_state==TRUE are decrypted in the output_message. For NTLMv1, all input data buffers where signed==TRUE are concatenated together and the signature is verified against the resulting concatenated buffer. For NTLMv2, the signature is verified for all of the input data buffers.

3.4.11 GSS_GetMICEx() Call

Inputs:

- context_handle CONTEXT HANDLE
- qop_req INTEGER, -- o specifies default QOP
- message ORDERED LIST of:
 - sign BOOLEAN
 - data OCTET STRING

Outputs:

- major_status INTEGER
- minor_status INTEGER
- message ORDERED LIST of:
 - sign BOOLEAN
 - data OCTET STRING
- per_msg_token OCTET STRING

This call is identical to GSS_GetMIC(), except that it supports multiple input buffers.

3.4.11.1 Signature Creation for GSS_GetMICEx()

Section [3.4.3](#) describes the algorithm used by GSS_GetMICEx() to create the signature. The per_msg_token contains the NTLMSSP_MESSAGE_SIGNATURE structure (section [2.2.1.4](#)).

The checksum is computed over the concatenated input buffers (section [3.1.5.2.2](#)) using only the input data buffers where ignore==FALSE for NTLMv1 and all of the input data buffers including the buffers where ignore==TRUE for NTLMv2.

3.4.12 GSS_VerifyMICEx() Call

Inputs:

- context_handle CONTEXT HANDLE

- message ORDERED LIST of:
 - sign BOOLEAN
 - data OCTET STRING
- per_msg_token OCTET STRING

Outputs:

- qop_state INTEGER
- major_status INTEGER
- minor_status INTEGER

This call is identical to `GSS_VerifyMIC()`, except that it supports multiple input buffers.

For NTLMv1, all input data buffers where `sign==TRUE` are concatenated together and the signature is verified against the resulting concatenated buffer. For NTLMv2, the signature is verified for all of the input data buffers including the buffers where `sign==FALSE`.

4 Protocol Examples

NTLM over a Server Message Block (SMB) transport is one of the most common uses of NTLM authentication and encryption. Kerberos is the preferred authentication method of an SMB session in Windows 2000 Server, Windows XP, Windows Server 2003, Windows Vista, and Windows Server 2008. However, when a client attempts to authenticate to an SMB server using the Kerberos protocol and fails, it attempts to authenticate with NTLM.

The following is an example protocol flow of NTLM and Simple and Protected Generic Security Service Application Program Interface Negotiation Mechanism (SPNEGO) (as specified in [\[MS-SPNG\]](#)) authentication of an SMB session.

Note The NTLM messages are embedded in the SMB messages. For details about how SMB embeds NTLM messages, see [\[MS-SMB\]](#) section 4.1.

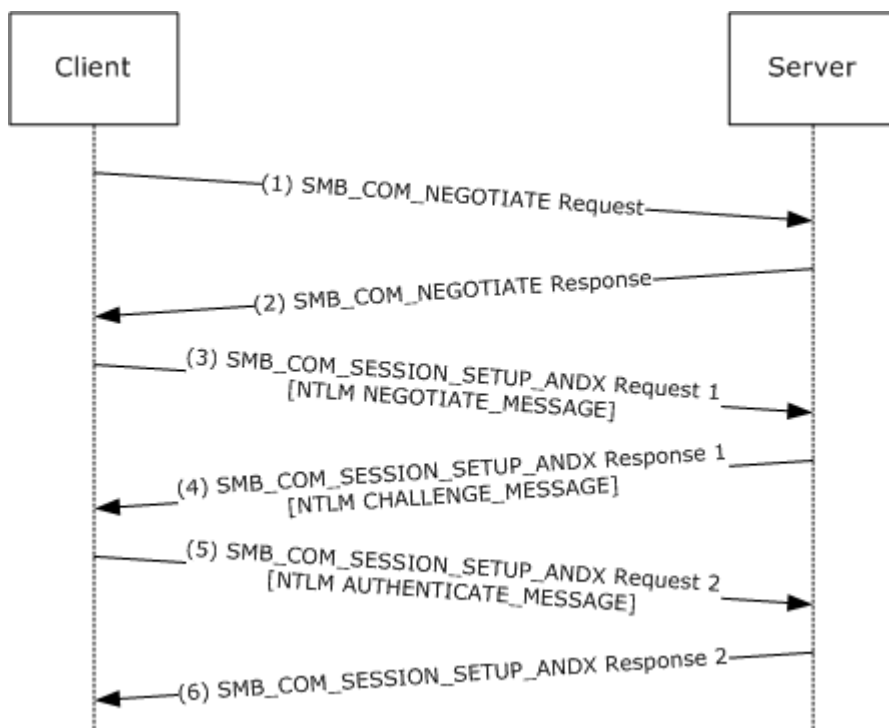


Figure 4: Message sequence to authenticate an SMB session

Steps 1 and 2: The SMB protocol negotiates protocol-specific options using the SMB_COM_NEGOTIATE request and response messages.

Step 3: The client sends an SMB_COM_SESSION_SETUP_ANDX request message. Assuming that NTLM authentication is negotiated, within this message an NTLM [NEGOTIATE_MESSAGE](#) is embedded.

Step 4: The server responds with an SMB_COM_SESSION_SETUP_ANDX response message within which an NTLM CHALLENGE_MESSAGE is embedded. The packet includes an 8-byte random number, called a "challenge", that the server generates and sends in the **ServerChallenge** field of the packet.

Step 5: The client extracts the **ServerChallenge** field from the NTLM [CHALLENGE MESSAGE](#) packet and sends an NTLM [AUTHENTICATE MESSAGE](#) packet to the server (embedded in an SMB_COM_SESSION_SETUP_ANDX request message).

If the challenge and the response prove that the client knows the user's password, the authentication succeeds and the client's security context is now established on the server.

Step 6: The server sends a success message embedded in an SMB_COM_SESSION_SETUP_ANDX response message.

5 Security

The following sections specify security considerations for implementers of the NT LAN Manager (NTLM) Authentication Protocol.

5.1 Security Considerations for Implementers

Implementers should be aware that NTLM does not support any recent cryptographic methods, such as AES or SHA-256. It uses **cyclic redundancy check (CRC)** or message digest algorithms (as specified in [\[RFC1321\]](#)) for integrity, and it uses RC4 (as specified in [SCHNEIER]) for encryption. Deriving a key from a password is as specified in [\[RFC1320\]](#) and [\[FIPS46-2\]](#). Therefore, applications are generally advised not to use NTLM. <30>

5.2 Index of Security Parameters

Security Parameter	Section
MD4/MD5 usage in NTLM v1	3.3.1
MD4/MD5 usage in NTLM v2	3.3.2
MD5/RC4 usage during session security	3.4

6 Appendix A: Cryptographic Operations Reference

In the algorithms provided in this documentation, pseudocode is provided to illustrate the process used to compute keys and perform other cryptographic operations prior to protocol exchange. The following table defines the general purpose functions and operations used in this pseudocode.

Functions	Description	Section
AddAVPair(T, Id, Value)	<p>An auxiliary function that is used to manage AV pairs in NTLM messages. It is defined as follows:</p> <pre> AddAvPair(T, Id, Value) { STRING T USHORT Id STRING Value T = ConcatenationOf(T, Id) T = ConcatenationOf(T, Length(Value)) T = ConcatenationOf(T, Value) } </pre>	3.2.5.1.1
ComputeResponse(...)	A function that computes the NT response, LM responses, and key exchange key from the response keys and challenge.	3.1.5.1.2 , 3.2.5.1.2 , 3.3.1 , 3.3.2
ConcatenationOf(string1, string2, ... stringN)	Indicates the left-to-right concatenation of the string parameters, from the first string to the <i>N</i> th. Any numbers are converted to strings and all numeric conversions to strings retain all digits, even non-significant ones. The result is a string. For example, ConcatenationOf(0x00122, "XYZ", "Client") results in the string "00122XYZClient."	3.3.1 , 3.3.2 , 3.4.2 , 3.4.3 , 3.4.4 , 3.4.6 , 3.4.7 , 3.4.8
CRC32(M)	Indicates a 32-bit cyclical redundancy check (CRC) calculated over M.	3.4.3 , 3.4.4
DES(K, D)	Indicates the encryption of an 8-byte data item D with the 7-byte key K using the Data Encryption Standard (DES) algorithm in ECB mode. The result is 8 bytes in length, as specified in [FIPS46-2] .	3.3.1 , 3.4.6
DESL(K, D)	Indicates the encryption of an 8-byte data item D with the 16-byte key K using the Data Encryption Standard Long (DESL) algorithm. The result is 24 bytes in length. DESL(K, D) is computed as follows:	3.3.1

Functions	Description	Section
	<pre>ConcatenationOf(DES(K[0..6], D), \ DES(K[7..13], D), DES(\ ConcatenationOf(K[14..15], Z(5)), D));</pre> <p>Note K[] implies a key represented as a character array.</p>	
GetVersion()	An auxiliary function that returns an OS version-specific value as specified in section 2.2.2.8 .	3.1.5.1.1 , 3.1.5.1.2 , 3.2.5.1.1 , 3.2.5.1.2
LMGETKEY(U, D)	Retrieve the user's LM response key from the server database (directory or local database).	3.2.5.1.2
NTGETKEY(U, D)	Retrieve the user's NT response key from the server database.	3.2.5.1.2
HMAC(K, M)	Indicates the encryption of data item M with the key K using the HMAC algorithm, as specified in [RFC2104] .	
HMAC_MD5(K, M)	Indicates the computation of a 16-byte HMAC-keyed MD5 message digest of the byte string M using the key K.	3.3.2 , 3.4.4
KXKEY(K, LM)	Produces a key exchange key from the LM response as defined in the sections KXKEY, SIGNKEY, and SEALKEY.	3.1.5.1.2 , 3.2.5.1.2 , 3.4.6
LMOWF	Computes a one-way function of the user's password to use as the response key. NTLM v1 and NTLM v2 define separate LMOWF functions in the NTLM v1 authentication and NTLM v2 authentication sections, respectively.	3.1.5.1.2 , 3.3.1 , 3.3.2
MD4(M)	Indicates the computation of an MD4 message digest of the null-terminated byte string M, as specified in [RFC1320] .	3.3.1 , 3.3.2
MD5(M)	Indicates the computation of an MD5 message digest of the null-terminated byte string M, as specified in [RFC1321] .	3.3.1 , 3.3.2 , 3.4.4 , 3.4.7 , 3.4.8
NIL	A zero-length string.	3.1.5.1.1 , 3.1.5.1.2 , 3.2.5.1.1 , 3.1.5.2.2 , 3.4.7
NONCE(N)	Indicates the computation of an N-byte cryptographic-strength random number. Note The NTLM Authentication Protocol does not define the statistical properties of the random number generator. It is left to the discretion of the implementation to define the strength requirements of the NONCE(N) operation.	3.1.5.1.2 , 3.2.5.1.1 , 3.4.3
NTOWF	Computes a one-way function of the user's password to use	3.1.5.1.2 ,

Functions	Description	Section
	as the response key. NTLM v1 and NTLM v2 define separate NTOWF functions in the NTLM v1 authentication and NTLM v2 authentication sections, respectively.	3.3.1 , 3.3.2
RC4(H, D)	Indicates the encryption of data item D with the current session or message key state, using the RC4 algorithm. For more information see [SCHNEIER]. H is the handle to a key state structure initialized by RC4INIT.	3.4.3 , 3.4.4
RC4K(K, D)	Indicates the encryption of data item D with the key K using the RC4 algorithm. For more information see [SCHNEIER]. Note The key sizes for RC4 encryption in NTLM are defined in sections KXKEY, SIGNKEY, and SEALKEY, where they are created. The length of D is variable and this must be taken into consideration for any implementation of RC4.	3.1.5.1.2 , 3.1.5.2.2 , 3.4.4
RC4Init(H, K)	Initialization of the RC4 key and handle to a key state structure for the session.	3.1.5.1.2 , 3.2.5.1.2
SEALKEY(F, K, string1)	Produces an encryption key from the session key as defined in sections KXKEY, SIGNKEY, and SEALKEY.	3.1.5.1.2 , 3.1.5.2.2 , 3.4.8
SIGNKEY(K, string1)	Produces a signing key from the session key as defined in sections KXKEY, SIGNKEY, and SEALKEY.	3.1.5.1.2 , 3.1.5.2.2 , 3.4.7
Currenttime	Indicates the retrieval of the current time as a 64-bit value, represented as the number of 100-nanosecond ticks elapsed since midnight of January 1st, 1601.	3.1.5.1.2
UNICODE(string)	Indicates the 2-byte little-endian byte order encoding of the Unicode representation of string.	3.3.1 , 3.3.2
UpperCase(string)	Indicates the uppercase representation of string.	3.3.1 , 3.3.2
Z(N)	Indicates the creation of a byte array of length N. Each byte in the array is initialized to the value zero.	3.3.1 , 3.3.2

7 Appendix B: Windows Behavior

The information in this specification is applicable to the following versions of Windows:

- Windows NT
- Windows 2000
- Windows XP
- Windows Server 2003
- Windows Vista

Exceptions, if any, are noted below. Unless otherwise specified, any statement of optional behavior in this specification prescribed using the terms SHOULD or SHOULD NOT implies Windows behavior in accordance with the SHOULD or SHOULD NOT prescription. Unless otherwise specified, the term MAY implies that Windows does not follow the prescription.

[<1> Section 1.3.1:](#) It is possible, with the Windows implementation of connectionless NTLM, for messages protected by NTLM session security to precede the completion of the established NTLM session, but such message orderings do not occur in practice.

[<2> Section 1.4:](#) When authenticating a domain account with NTLM, Windows uses Netlogon (as specified in [\[MS-APDS\]](#)) to have the domain controller (DC) take the challenge and the client's response, and validate the user authentication against the DC's user database.

[<3> Section 1.6:](#) Windows applications that use Negotiate (as specified in [\[MS-SPNG\]](#)) may authenticate via NTLM if Kerberos is not available. Authenticating via NTLM would occur if either the client or server are down-level (running Windows NT 4.0 or earlier) systems, if the server is not joined to a domain, if the application is using an RPC interface that uses NTLM directly, or if the administrator has not configured Kerberos properly. An implementer who wants to support these scenarios in which Kerberos does not work would need to implement NTLM.

[<4> Section 2.2.1.1:](#) Windows implements the SHOULD behaviors specified for this message.

[<5> Section 2.2.1.1:](#) The **Version** field is sent or consumed only by Windows XP, Windows Server 2003, Windows Vista, and Windows Server 2008. Earlier versions of Windows assume that the **Payload** field started immediately after **WorkstationNameBufferOffset**. However, because all references into the **Payload** field are by offset from the start of the message (not from the start of the **Payload** field), an earlier version of Windows can correctly interpret a packet constructed by a later version of Windows.

[<6> Section 2.2.1.1:](#) The **codepage** mapping the OEM character set to UNICODE is configurable via HKEY_LOCAL_MACHINE\System\CurrentControlSet\control\Nls\Codepage\OEMCP, which is a **DWORD** that contains the assigned number of the codepage.

[<7> Section 2.2.1.2:](#) Windows implements the SHOULD behaviors specified for this message.

[<8> Section 2.2.1.2:](#) The **Version** field is sent or consumed only by Windows XP, Windows Server 2003, Windows Vista, and Windows Server 2008. Earlier versions of Windows assume that the **Payload** field started immediately after **WorkstationNameBufferOffset**. However, because all references into the **Payload** field are by offset from the start of the message (not from the start of the **Payload** field), an earlier version of Windows can correctly interpret a packet constructed by a later version of Windows.

<9> [Section 2.2.1.3:](#) Windows implements the SHOULD behaviors specified for this message.

<10> [Section 2.2.1.3:](#) Although the protocol allows authentication to succeed if the client provides either **LmChallengeResponse** or **NtChallengeResponse**, Windows implementations provide both.

<11> [Section 2.2.1.3:](#) The **Version** field is sent or consumed only by Windows XP, Windows Server 2003, Windows Vista, and Windows Server 2008. Earlier versions of Windows assume that the **Payload** field started immediately after **WorkstationNameBufferOffset**. However, because all references into the **Payload** field are by offset from the start of the message (not from the start of the **Payload** field), an earlier version of Windows can correctly interpret a packet constructed by a later version of Windows.

<12> [Section 2.2.2.1:](#) This type is present and used in Windows Vista only.

<13> [Section 2.2.2.2:](#) Restrictions were first added in Windows Vista. No prior version of Windows supports these restrictions.

<14> [Section 2.2.2.2:](#) Windows Vista uses a hierarchical order of values to indicate the trustworthiness of the client application. Lower values indicate that the subject has lower integrity.

<15> [Section 2.2.2.2:](#) The **MachineID** is a random number generated once per boot of a Windows system.

<16> [Section 2.2.2.5:](#) This flag is supported in Windows XP, Windows Server 2003, Windows Vista, and Windows Server 2008, and is used for debug purposes only.

<17> [Section 2.2.2.5:](#) NTLMSSP_NEGOTIATE_NTLM2 is always set in the NEGOTIATE_MESSAGE to the server and the CHALLENGE_MESSAGE to the client in Windows NT 4.0 SP4 and later, Windows 2000, Windows XP, Windows Server 2003, Windows Vista, and Windows Server 2008.

<18> [Section 2.2.2.5:](#) This flag is supported in Windows XP, Windows Server 2003, Windows Vista, and Windows Server 2008.

<19> [Section 2.2.2.5:](#) This flag is supported in Windows XP, Windows Server 2003, Windows Vista and Windows Server 2008.

<20> [Section 2.2.2.5:](#) NTLMSSP_NEGOTIATE_NTLM2 is set in the NEGOTIATE_MESSAGE to the server and the CHALLENGE_MESSAGE to the client in Windows NT 4.0 SP4 and later, Windows 2000, Windows XP, Windows Server 2003, Windows Vista, and Windows Server 2008.

<21> [Section 2.2.2.5:](#) Windows supports the SHOULD behavior described in this section.

<22> [Section 2.2.2.7:](#) Windows implements the SHOULD behaviors described in this section.

<23> [Section 2.2.2.9:](#) NTLMSSP_NEGOTIATE_VERSION can be negotiated only in Windows XP SP2, Windows Server 2003, Windows Vista and Windows Server 2008.

<24> [Section 2.2.2.9:](#) For Windows Server 2003 and Windows XP SP2, the value of this field is WINDOWS_MAJOR_VERSION_5. For Windows Server 2008 and Windows Vista, the value of this field is WINDOWS_MAJOR_VERSION_6.

<25> [Section 2.2.2.9:](#) For Windows Server 2008 and Windows Vista, the value of this field is WINDOWS_MINOR_VERSION_0. For Windows XP SP2, the value of this field is WINDOWS_MINOR_VERSION_1. For Windows Server 2003, the value of this field is WINDOWS_MINOR_VERSION_2.

<26> [Section 2.2.2.9:](#) For Windows Server 2008, Windows Vista, Windows Server 2003 and Windows XP SP2, the value of this field is NTLMSSP_REVISION_W2K3.

<27> [Section 3.1.1.2:](#) Windows exposes these logical parameters to applications through the SSPI interface on Windows.

<28> [Section 3.3.1:](#) If the client sends a domain that is unknown to the server, the server tries to perform the authentication against the local database.

<29> [Section 3.3.2:](#) If the client sends a domain that is unknown to the server, the server tries to perform the authentication against the local database.

<30> [Section 5.1:](#) NTLM domain considerations are as follows:

Microsoft domain controllers (DCs) determine the minimum security requirements for NTLM authentication between a Windows client and the local Windows domain. Based on the minimum security settings in place, the DC can either allow or refuse the use of LM, NTLM, or NTLM v2 authentication, and servers can force the use of NTLM v2 session security on all messages between the client and server. In a Windows domain, the DC controls domain level security settings through the use of Windows Group Policy, which replicates security policies to clients and servers throughout the local domain.

Domain-level security policies dictated by Windows Group Policy must be supported on the local system for authentication to take place. Clients and servers exchange NTLM capability flags during NTLM authentication that specify what levels of security they are able to support. If either the client or server's level of security support is less than the security policies of the domain, the authentication attempt is refused by the computer with the higher level of minimum security requirements. This is important for interdomain authentication where differing security policies may be enforced on either domain, and the client or server may not be able to support the security policies of the other's domain.

NTLM security levels are as follows:

The security policies exchanged by the server and client can be set independently of the DC minimum security requirements dictated by Windows Group Policy. Higher local security policies can be exchanged by a client and server in a domain with low minimum security requirements in connection-oriented authentication during the capability flags exchange. However, during connectionless (datagram-oriented) authentication, it is not possible to exchange higher local security policies because they are strictly enforced by Windows Group Policy. Local security policies that are set independently of the DC are subordinate to domain-level security policies for clients authenticating to a server on the local domain; therefore, it is not possible to use local-system policies that are less secure than domain-level policies.

Stand-alone servers that do not have a DC to authenticate clients set their own minimum security requirements.

NTLM security levels determine the minimum security settings allowed on a client, server, or DC to authenticate in an NTLM domain. The security levels can be modified in Windows Server 2008, Windows Vista, Windows Server 2003, Windows XP, Windows 2000, and Windows NT 4.0 SP4 by setting this registry key to one of the following security level values.

```
HKEY_LOCAL_MACHINE\System\CurrentControlSet\Control\Lsa\  
LMCompatibilityLevel
```

Security-level descriptions:

0: Server sends LM and NTLM response and never uses NTLM v2 session security. Clients use LM and NTLM authentication, and never use NTLM v2 session security. Domain Controllers (DCs) accept LM, NTLM, and NTLM v2 authentication.

1: Servers use NTLM v2 session security if it is negotiated. Clients use LM and NTLM authentication and use NTLM v2 session security if the server supports it. DCs accept LM, NTLM, and NTLM v2 authentication.

2: Server sends NTLM response only. Clients use only NTLM authentication and use NTLM v2 session security if the server supports it. DCs accept LM, NTLM, and NTLM v2 authentication.

3: Server sends NTLM v2 response only. Clients use NTLM v2 authentication and use NTLM v2 session security if the server supports it. DCs accept LM, NTLM, and NTLM v2 authentication.

4:DCs refuse LM responses. Clients use NTLM authentication and use NTLM v2 session security if the server supports it. DCs refuse LM authentication but accept NTLM and NTLM v2 authentication.

5:DCs refuse LM and NTLM responses, and accept only NTLM v2. Clients use NTLM v2 authentication and use NTLM v2 session security if the server supports it. DCs refuse NTLM and LM authentication, and accept only NTLM v2 authentication.

The default minimum security setting for Windows Server 2008, Windows Vista, Windows Server 2003, Windows XP, Windows 2000, and Windows NT 4.0 SP4 is level 0. With the default security setting in place, the client [AUTHENTICATE MESSAGE](#) packet is sent with both the NTLM and LM session key responses to the server [CHALLENGE MESSAGE](#) packet, and NTLM v2 session security is not used by the client and server to sign and seal messages. The client response contains the [LMv2_RESPONSE](#) and [NTLMv2_RESPONSE](#) only if the use of NTLM v2 authentication is supported; otherwise, the client response contains the [LM_RESPONSE](#) and [NTLM_RESPONSE](#).

8 Index

A

Abstract data model

[client](#)
[server](#)

[Applicability](#)

AUTHENTICATE_MESSAGE ([section 3.1.5.2.2](#), [section 3.2.5.1.2](#))

[AUTHENTICATE MESSAGE packet](#)

Authentication

[NTLMv1](#)
[NTLMv2](#)

[AV_PAIR packet](#)

C

Call flow

[connectionless](#)
[connection-oriented](#)
[overview](#)

[Capability negotiation](#)

CHALLENGE_MESSAGE ([section 3.1.5.1.2](#), [section 3.1.5.2.1](#), [section 3.2.5.2.1](#))

[CHALLENGE MESSAGE packet](#)

Client

[abstract data model](#)
[connectionless message processing](#)
[connection-oriented message processing](#)
[exposed variables](#)
[higher-layer triggered events](#)
[initialization](#)
[internal variables](#)
[local events](#)
[message processing](#)
[overview](#)
[sequencing rules](#)
[timer events](#)
[timers](#)

[Confidentiality](#)

Connectionless

[call flow](#)
[message processing - client](#)
[message processing - server](#)

Connection-oriented

[call flow](#)
[message processing - client](#)
[message processing - server](#)

[Cryptographic operations reference](#)

D

Data model - abstract

[client](#)
[server](#)

E

[Examples](#)

Exposed variables

[client](#)
[server](#)

F

[Fields - vendor-extensible](#)

G

[Glossary](#)

H

Higher-layer triggered events

[client](#)
[server](#)

I

[Implementer - security considerations](#)

[Index of security parameters](#)

[Informative references](#)

Initialization

[client](#)
[server](#)

Internal variables

[client](#)
[server](#)

[Introduction](#)

K

KXKEY ([section 3.4.5](#), [section 3.4.6](#))

L

[LM_RESPONSE packet](#)

[LMv2_RESPONSE packet](#)

Local events

[client](#)
[server](#)

M

[Message packet](#)

Message processing

[client](#)
[server](#)

Messages

[NTLM](#)
[overview](#)
[structures](#)
[syntax](#)
[transport](#)

N

[NEGOTIATE packet](#)

[NEGOTIATE_MESSAGE](#) ([section 3.1.5.1.1](#), [section 3.2.5.1.1](#))

[NEGOTIATE_MESSAGE](#) packet

[Normative references](#)

[NTLM authentication call flow](#)

[NTLM connectionless call flow](#)

[NTLM connection-oriented call flow](#)

[NTLM messages](#)

[NTLM_RESPONSE](#) packet

[NTLMSSP_MESSAGE_SIGNATURE](#) packet

NTLMv1

[authentication](#)

[overview](#)

NTLMv2

[authentication](#)

[overview](#)

[NTLMv2_CLIENT_CHALLENGE](#) packet

[NTLMv2_RESPONSE](#) packet

O

[Overview \(synopsis\)](#)

P

[Parameters - security index](#)

[Preconditions](#)

[Prerequisites](#)

R

[References](#)

[informative](#)

[normative](#)

[overview](#)

[Relationship to other protocols](#)

[Response checking](#)

[Restriction Encoding](#) packet

S

SEALKEY ([section 3.4.5](#), [section 3.4.8](#))

Security

[implementer considerations](#)

[overview](#)

[parameter index](#)

[session](#)

Sequencing rules

[client](#)

[server](#)

Server

[abstract data model](#)

[connectionless message processing](#)

[connection-oriented message processing](#)

[exposed variables](#)

[higher-layer triggered events](#)

[initialization](#)

[internal variables](#)

[local events](#)

[message processing](#)

[overview](#)

[response checking](#)

[sequencing rules](#)

[timer events](#)

[timers](#)

Session security

[confidentiality](#)

[integrity](#)

KXKEY ([section 3.4.5](#), [section 3.4.6](#))

[overview](#)

SEALKEY ([section 3.4.5](#), [section 3.4.8](#))

[signature functions](#)

SIGNKEY ([section 3.4.5](#), [section 3.4.7](#))

[Signature functions](#)

SIGNKEY ([section 3.4.5](#), [section 3.4.7](#))

[Standards assignments](#)

[Structures](#)

[Syntax](#)

T

Timer events

[client](#)

[server](#)

Timers

[client](#)

[server](#)

[Transport](#)

Triggered events - higher-layer

[client](#)

[server](#)

V

Variables

[exposed - client](#)

[exposed - server](#)

[internal - client](#)

[internal - server](#)

[Vendor-extensible fields](#)

[VERSION](#) packet

[Versioning](#)

W

[Windows behavior](#)