The NTLM Authentication Protocol and Security Support Provider

Abstract

This article seeks to describe the NTLM authentication protocol and related security support provider functionality at an intermediate to advanced level of detail, suitable as a reference for implementors. It is hoped that this document will evolve into a comprehensive description of NTLM; at this time there are omissions, both in the author's knowledge and in his documentation, and almost certainly inaccuracies. However, this document should at least be able to provide a solid foundation for further research. The information presented herein was used as the basis for the implementation of NTLM authentication in the open-source jCIFS library, available at http://jcifs.samba.org. This documentation is based on independent research by the author and analysis of functionality implemented in the Samba software suite.

Contents

What is NTLM?

NTLM is a suite of authentication and session security protocols used in various Microsoft network protocol implementations and supported by the NTLM Security Support Provider ("NTLMSSP"). Originally used for authentication and negotiation of secure DCE/RPC, NTLM is also used throughout Microsoft's systems as an integrated single sign-on mechanism. It is probably best recognized as part of the "Integrated Windows Authentication" stack for HTTP authentication; however, it is also used in Microsoft implementations of SMTP, POP3, IMAP (all part of Exchange), CIFS/SMB, Telnet, SIP, and possibly others.

The NTLM Security Support Provider provides authentication, integrity, and confidentiality services within the Window Security Support Provider Interface (SSPI) framework. SSPI specifies a core set of security functionality that is implemented by supporting providers; the NTLMSSP is such a provider. The SSPI specifies, and the NTLMSSP implements, the following core operations:

1. Authentication -- NTLM provides a challenge-response authentication mechanism, in which clients are able to prove their identities without sending a password to the server.

2. Signing -- The NTLMSSP provides a means of applying a digital "signature" to a message. This ensures that the signed message has not been modified (either accidentally or intentionally) and that that signing party has knowledge of a shared secret. NTLM implements a symmetric signature scheme (Message Authentication Code, or MAC); that is, a valid signature can only be generated and verified by parties that possess the common shared key.

3. Sealing -- The NTLMSSP implements a symmetric-key encryption mechanism, which provides message confidentiality. In the case of NTLM, sealing also implies signing (a signed message is not necessarily sealed, but all sealed messages are signed).

NTLM has been largely supplanted by Kerberos as the authentication protocol of choice for domain-based scenarios. However, Kerberos is a trusted-third-party scheme, and cannot be used in situations where no trusted third party exists; for example, member servers (servers that are not part of a domain), local accounts, and authentication to resources in an untrusted domain. In such scenarios, NTLM continues to be the primary authentication mechanism (and likely will be for a long time).

NTLM Terminology

Before we start digging in any further, we will need to define a few terms used in the various protocols.

NTLM authentication is a challenge-response scheme, consisting of three messages, commonly referred to as Type 1 (negotiation), Type 2 (challenge) and Type 3 (authentication). It basically works like this:

1. The client sends a Type 1 message to the server. This primarily contains a list of features supported by the client and requested of the server.
2. The server responds with a Type 2 message. This contains a list of features supported and agreed upon by the server. Most importantly, however, it contains a challenge generated by the server.
3. The client replies to the challenge with a Type 3 message. This contains several pieces of information about the client, including the domain and username of the client user. It also contains one or more responses to the Type 2 challenge.

The responses in the Type 3 message are the most critical piece, as they prove to the server that the client user has knowledge of the account password.

The process of authentication establishes a shared context between the two involved parties; this includes a shared session key, used for subsequent signing and sealing operations.

In this document, to avoid confusion (as much as possible, anyway) the following convention will be observed:

- When discussing authentication, the protocol version will use "v-numbering"; for example, "NTLM***v1*** Authentication".
- When discussing session security (signing & sealing), the "v" will be omitted; for example, "NTLM***1*** Session Security".

This should keep things fairly clear, except for the possibly awkward case of "NTLM2 Session Response" authentication (a variant of NTLMv1 authentication that is used in conjunction with NTLM2 session security). Hopefully by the time we get there this will all make much more sense.

For our purposes, a "`short`" is a little-endian, 16-bit unsigned value. For example, the decimal value "`1234`" represented as a `short` would be physically laid out as "`0xd204`" in hexadecimal.

A "`long`" is a little-endian, 32-bit unsigned value. The decimal value "`1234`" represented as a `long` in hexidecimal would be "`0xd2040000`".

A Unicode string is a string in which each character is represented as a 16-bit little-endian value (16-bit UCS-2 Transformation Format, little-endian byte order, with no Byte Order Mark and no null-terminator). The string "hello" in Unicode would be represented hexidecimally as "`0x680065006c006c006f00`".

An OEM string is a string in which each character is represented as an 8-bit value from the local machine's native character set (DOS codepage). There is no null-terminator. In NTLM messages, OEM strings are typically presented in uppercase. The string "HELLO" in OEM would be represented hexidecimally as "`0x48454c4c4f`".

A "security buffer" is a structure used to point to a buffer of binary data. It consists of:

1. A `short` containing the length of the buffer content in bytes (may be zero).
2. A `short` containing the allocated space for the buffer in bytes (greater than or equal to the length; typically the same as the length).
3. A `long` containing the offset to the start of the buffer in bytes (from the beginning of the NTLM message).

So the security buffer "`0xd204d204e1100000`" would be read as:

Length: `0xd204` (1234 bytes)
Allocated Space: `0xd204` (1234 bytes)
Offset: `0xe1100000` (4321 bytes)

If you started at the first byte in the message, and skipped ahead 4321 bytes, you would be at the start of the data buffer. You would read 1234 bytes (which is the length of the buffer). Since the allocated space for the buffer is also 1234 bytes, you would then be at the end of the buffer.

The NTLM Message Header Layout

Now we're ready to look at the physical layout of NTLM authentication message headers.

All messages start with the NTLMSSP signature, which is (aptly enough) the null-terminated ASCII string "NTLMSSP" (hexadecimal "`0x4e544c4d53535000`").

Next is a `long` containing the message type (1, 2, or 3). A Type 1 message, for example, has type "`0x01000000`" in hex.

This is followed by message-specific information, typically consisting of security buffers and the message flags.

The NTLM Flags

The message flags are contained in a bitfield within the header. This is a `long`, in which each bit represents a specific flag. Most of these will make more sense later, but we'll go ahead and present them here to establish a frame of reference for the rest of the discussion. Flags marked as "unidentified" or "unknown" in the table below are outside the realm of the author's knowledge (which is not by any means absolute).

Flag	Name	Description
`0x00000001`	Negotiate Unicode	Indicates that Unicode strings are supported for use in security buffer data.
`0x00000002`	Negotiate OEM	Indicates that OEM strings are supported for use in security buffer data.
`0x00000004`	Request Target	Requests that the server's authentication realm be included in the Type 2 message.
`0x00000008`	*unknown*	*This flag's usage has not been identified.*
`0x00000010`	Negotiate Sign	Specifies that authenticated communication between the client and server should carry a digital signature (message integrity).
`0x00000020`	Negotiate Seal	Specifies that authenticated communication between the client and server should be encrypted (message confidentiality).
`0x00000040`	Negotiate Datagram Style	Indicates that datagram authentication is being used.
`0x00000080`	Negotiate Lan Manager Key	Indicates that the Lan Manager Session Key should be used for signing and sealing authenticated communications.
`0x00000100`	Negotiate Netware	*This flag's usage has not been identified.*
`0x00000200`	Negotiate NTLM	Indicates that NTLM authentication is being used.
`0x00000400`	*unknown*	*This flag's usage has not been identified.*
`0x00000800`	Negotiate Anonymous	Sent by the client in the Type 3 message to indicate that an anonymous context has been established. This also affects the response fields (as detailed in the "Anonymous Response" section).
`0x00001000`	Negotiate Domain Supplied	Sent by the client in the Type 1 message to indicate that the name of the domain in which the client workstation has membership is included in the message. This is used by the server to determine whether the client is eligible for local authentication.
`0x00002000`	Negotiate Workstation Supplied	Sent by the client in the Type 1 message to indicate that the client workstation's name is included in the message. This is used by the server to determine whether the client is eligible for local authentication.
`0x00004000`	Negotiate Local Call	Sent by the server to indicate that the server and client are on the same machine. Implies that the client may use the established local credentials for authentication instead of calculating a response to the challenge.
`0x00008000`	Negotiate Always Sign	Indicates that authenticated communication between the client and server should be signed with a "dummy" signature.

`0x00010000`	Target Type Domain	Sent by the server in the Type 2 message to indicate that the target authentication realm is a domain.
`0x00020000`	Target Type Server	Sent by the server in the Type 2 message to indicate that the target authentication realm is a server.
`0x00040000`	Target Type Share	*Sent by the server in the Type 2 message to indicate that the target authentication realm is a share. Presumably, this is for share-level authentication. Usage is unclear.*
`0x00080000`	Negotiate NTLM2 Key	Indicates that the NTLM2 signing and sealing scheme should be used for protecting authenticated communications. Note that this refers to a particular session security scheme, and is not related to the use of NTLMv2 authentication. This flag can, however, have an effect on the response calculations (as detailed in the "NTLM2 Session Response" section).
`0x00100000`	Request Init Response	*This flag's usage has not been identified.*
`0x00200000`	Request Accept Response	*This flag's usage has not been identified.*
`0x00400000`	Request Non-NT Session Key	*This flag's usage has not been identified.*
`0x00800000`	Negotiate Target Info	Sent by the server in the Type 2 message to indicate that it is including a Target Information block in the message. The Target Information block is used in the calculation of the NTLMv2 response.
`0x01000000`	*unknown*	*This flag's usage has not been identified.*
`0x02000000`	*unknown*	*This flag's usage has not been identified.*
`0x04000000`	*unknown*	*This flag's usage has not been identified.*
`0x08000000`	*unknown*	*This flag's usage has not been identified.*
`0x10000000`	*unknown*	*This flag's usage has not been identified.*
`0x20000000`	Negotiate 128	Indicates that 128-bit encryption is supported.
`0x40000000`	Negotiate Key Exchange	Indicates that the client will provide an encrypted master key in the "Session Key" field of the Type 3 message.
`0x80000000`	Negotiate 56	Indicates that 56-bit encryption is supported.

As an example, consider a message specifying:

Negotiate Unicode (`0x00000001`)
Request Target (`0x00000004`)
Negotiate NTLM (`0x00000200`)
Negotiate Always Sign (`0x00008000`)

Combining the above gives "`0x00008205`". This would be physically laid out as "`0x05820000`" (since it is represented in little-endian byte order).

The Type 1 Message

Let's jump in and take a look at the Type 1 message:

	Description	Content
0	NTLMSSP Signature	Null-terminated ASCII "NTLMSSP" (`0x4e544c4d53535000`)
8	NTLM Message Type	`long` (`0x01000000`)
12	Flags	`long`
(16)	Supplied Domain *(Optional)*	security buffer
(24)	Supplied Workstation *(Optional)*	security buffer
(32)	OS Version Structure *(Optional)*	8 bytes
(32) *(40)*	*start of data block (if required)*	

The Type 1 message is sent from the client to the server to initiate NTLM authentication. Its primary purpose is to establish the "ground rules" for authentication by indicating supported options via the flags. Optionally, it can also provide the server with the client's workstation name and the domain in which the client workstation has membership; this information is used by the server to determine whether the client is eligible for local authentication.

Typically, the Type 1 message contains flags from the following set:

Negotiate Unicode (`0x00000001`)	The client sets this flag to indicate that it supports Unicode strings.
Negotiate OEM (`0x00000002`)	This is set to indicate that the client supports OEM strings.
Request Target (`0x00000004`)	This requests that the server send the authentication target with the Type 2 reply.
Negotiate NTLM (`0x00000200`)	Indicates that NTLM authentication is supported.
Negotiate Domain Supplied (`0x00001000`)	When set, the client will send with the message the name of the domain in which the workstation has membership.
Negotiate Workstation Supplied (`0x00002000`)	Indicates that the client is sending its workstation name with the message.
Negotiate Always Sign (`0x00008000`)	Indicates that communication between the client and server after authentication should carry a "dummy" signature.
Negotiate NTLM2 Key (`0x00080000`)	Indicates that this client supports the NTLM2 signing and sealing scheme; if negotiated, this can also affect the response calculations.
Negotiate 128 (`0x20000000`)	Indicates that this client supports strong (128-bit) encryption.
Negotiate 56 (`0x80000000`)	Indicates that this client supports medium (56-bit) encryption.

The supplied domain is a security buffer containing the domain in which the client workstation has membership. This is always in OEM format, even if Unicode is supported by the client.

The supplied workstation is a security buffer containing the client workstation's name. This, too, is in OEM rather than Unicode.

The OS Version structure was introduced in recent Windows updates; it identifies the host's operating system build level, and is formatted as follows:

	Description	Content
0	Major Version Number	1 byte
1	Minor Version Number	1 byte
2	Build Number	`short`
4	Unknown	`0x0000000f`

The operating system build can be found by running "winver.exe"; it should give a string similar to:

```
Version 5.1 (Build 2600.xpsp_sp2_gdr.050301-1519 : Service Pack 2)
```

This yields an OS Version structure of "`0x0501280a0000000f`":

`0x05`	(major version 5)
`0x01`	(minor version 1; Windows XP)
`0x280a`	(build number 2600 in hexadecimal little-endian)
`0x0000000f`	(unknown/reserved)

Note that the OS Version structure and the supplied domain/workstation are optional fields. There are three versions of the Type 1 message that have been observed in the wild:

1. Version 1 -- The Supplied Domain and Workstation security buffers and OS Version structure are omitted completely. In this case the message ends after the flags field, and is a fixed-length 16-byte structure. This form is typically seen in older Win9x-based systems, and is roughly documented in the Open Group's ActiveX reference documentation (Section 11.2.2).

2. Version 2 -- The Supplied Domain and Workstation buffers are present, but the OS Version structure is not. The data block begins immediately after the security buffer headers, at offset 32. This form is seen in most out-of-box shipping versions of Windows.

3. Version 3 -- Both the Supplied Domain/Workstation buffers are present, as well as the OS Version structure. The data block begins after the OS Version structure, at offset 40. This form was introduced in a relatively recent Service Pack, and is seen on currently-patched versions of Windows 2000, Windows XP, and Windows 2003.

The "most-minimal" well-formed Type 1 message, therefore, would be:

```
4e544c4d535350000100000002020000
```

This is a "Version 1" Type 1 message containing only the NTLMSSP signature, the NTLM message type, and the minimal set of flags (Negotiate NTLM and Negotiate OEM).

Type 1 Message Example

Consider the following hexadecimal Type 1 Message:

```
4e544c4d5353500001000000073200000600060033000000b000b0028000000
05009308000000f574f524b53544154494f4e444f4d41494e
```

We break this up as follows:

0	0x4e544c4d53535000	NTLMSSP Signature
8	0x01000000	Type 1 Indicator
12	0x07320000	Flags: Negotiate Unicode (0x00000001) Negotiate OEM (0x00000002) Request Target (0x00000004) Negotiate NTLM (0x00000200) Negotiate Domain Supplied (0x00001000) Negotiate Workstation Supplied (0x00002000)
16	0x0600060033000000	Supplied Domain Security Buffer: Length: 6 bytes (0x0600) Allocated Space: 6 bytes (0x0600) Offset: 51 bytes (0x33000000)
24	0x0b000b0028000000	Supplied Workstation Security Buffer: Length: 11 bytes (0x0b00) Allocated Space: 11 bytes (0x0b00) Offset: 40 bytes (0x28000000)
32	0x050093080000000f	OS Version Structure: Major Version: 5 (0x05) Minor Version: 0 (0x00) Build Number: 2195 (0x9308) Unknown/Reserved (0x0000000f)
40	0x574f524b53544154494f4e	Supplied Workstation Data ("WORKSTATION")
51	0x444f4d41494e	Supplied Domain Data ("DOMAIN")

Analyzing this information, we can see:

- This is an NTLM Type 1 message (from the NTLMSSP Signature and Type 1 Indicator).
- This client can support either Unicode or OEM strings (the Negotiate Unicode and Negotiate OEM flags are both set).
- This client supports NTLM authentication (Negotiate NTLM).
- The client is requesting that the server send information regarding the authentication target (Request Target is set).
- The client is running Windows 2000 (5.0), build 2195 (the production build number for Windows 2000 systems).
- This client is sending its domain, which is "DOMAIN" (the Negotiate Domain Supplied flag is set, and the domain name is present in the Supplied Domain Security Buffer).
- The client is sending its workstation name, which is "WORKSTATION" (the Negotiate Workstation Supplied flag is set, and the workstation name is present in the Supplied Workstation Security Buffer).

Note that the supplied workstation and domain are in OEM format. Additionally, the order in which the security buffer data blocks are laid out is unimportant; in the example, the workstation data is placed

before the domain data.

After creating the Type 1 message, the client sends it to the server. The server analyzes the message, much as we have just done, and creates a reply. This brings us to our next topic, the Type 2 message.

The Type 2 Message

	Description	Content
0	NTLMSSP Signature	Null-terminated ASCII "NTLMSSP" (`0x4e544c4d53535000`)
8	NTLM Message Type	`long` (`0x02000000`)
12	Target Name	security buffer
20	Flags	`long`
24	Challenge	8 bytes
(32)	Context *(optional)*	8 bytes (two consecutive `long`s)
(40)	Target Information *(optional)*	security buffer
(48)	OS Version Structure *(Optional)*	8 bytes
32 *(48)* *(56)*	*start of data block*	

The Type 2 message is sent by the server to the client in response to the client's Type 1 message. It serves to complete the negotiation of options with the client, and also provides a challenge to the client. It may optionally contain information about the authentication target.

Typical Type 2 message flags include:

Negotiate Unicode (`0x00000001`)	The server sets this flag to indicate that it will be using Unicode strings. This should only be set if the client indicates (in the Type 1 message) that it supports Unicode. Either this flag or Negotiate OEM should be set, but not both.
Negotiate OEM (`0x00000002`)	This flag is set to indicate that the server will be using OEM strings. This should only be set if the client indicates (in the Type 1 message) that it will support OEM strings. Either this flag or Negotiate Unicode should be set, but not both.
Request Target (`0x00000004`)	This flag is often set in the Type 2 message; while it has a well-defined meaning within the Type 1 message, its semantics here are unclear.
Negotiate NTLM (`0x00000200`)	Indicates that NTLM authentication is supported.
Negotiate Local Call (`0x00004000`)	The server sets this flag to inform the client that the server and client are on the same machine. The server provides a local security context handle with the message.
Negotiate Always Sign (`0x00008000`)	Indicates that communication between the client and server after authentication should carry a "dummy" signature.

Target Type Domain (`0x00010000`)	The server sets this flag to indicate that the authentication target is being sent with the message and represents a domain.
Target Type Server (`0x00020000`)	The server sets this flag to indicate that the authentication target is being sent with the message and represents a server.
Target Type Share (`0x00040000`)	The server apparently sets this flag to indicate that the authentication target is being sent with the message and represents a network share. This has not been confirmed.
Negotiate NTLM2 Key (`0x00080000`)	Indicates that this server supports the NTLM2 signing and sealing scheme; if negotiated, this can also affect the client's response calculations.
Negotiate Target Info (`0x00800000`)	The server sets this flag to indicate that a Target Information block is being sent with the message.
Negotiate 128 (`0x20000000`)	Indicates that this server supports strong (128-bit) encryption.
Negotiate 56 (`0x80000000`)	Indicates that this server supports medium (56-bit) encryption.

The target name is a security buffer containing the name of the authentication target. This is typically sent in response to a client requesting the target (via the Request Target flag in the Type 1 message). This can contain a domain, server, or (apparently) a network share. The target type is indicated via the Target Type Domain, Target Type Server, and Target Type Share flags. The target name can be either Unicode or OEM, as indicated by the presence of the appropriate flag in the Type 2 message.

The challenge is an 8-byte block of random data. The client will use this to formulate a response.

The context field is typically populated when Negotiate Local Call is set. It contains an SSPI context handle, which allows the client to "short-circuit" authentication and effectively circumvent responding to the challenge. Physically, the context is two `long` values. This is covered in greater detail later, in the "Local Authentication" section.

The target information is a security buffer containing a Target Information block, which is used in calculating the NTLMv2 response (discussed later). This is composed of a sequence of subblocks, each consisting of:

Field	Content	Description
Type	`short`	Indicates the type of data in this subblock: 1 (`0x0100`): Server name 2 (`0x0200`): Domain name 3 (`0x0300`): Fully-qualified DNS host name (i.e., `server.domain.com`) 4 (`0x0400`): DNS domain name (i.e., `domain.com`)
Length	`short`	Length in bytes of this subblock's content field
Content	Unicode string	Content as indicated by the type field. Always sent in Unicode, even when OEM is indicated by the message flags.

The sequence is terminated by a terminator subblock; this is a subblock of type "0", of zero length. Subblocks of type "5" have also been encountered, apparently containing the "parent" DNS domain for servers in subdomains; it may be that there are other as-yet-unidentified subblock types as well.

The OS Version structure was described previously.

As with the Type 1 message, there are a few versions of the Type 2 that have been observed:

1. Version 1 -- The Context, Target Information, and OS Version structure are all omitted. The data block (containing only the contents of the Target Name security buffer) begins at offset 32. This form is seen in older Win9x-based systems, and is roughly documented in the Open Group's ActiveX reference documentation (Section 11.2.3).

2. Version 2 -- The Context and Target Information fields are present, but the OS Version structure is not. The data block begins after the Target Information header, at offset 48. This form is seen in most out-of-box shipping versions of Windows.

3. Version 3 -- The Context, Target Information, and OS Version structure are all present. The data block begins after the OS Version structure, at offset 56. Again, the buffers may be empty (yielding a zero-length data block). This form was introduced in a relatively recent Service Pack, and is seen on currently-patched versions of Windows 2000, Windows XP, and Windows 2003.

A minimal Type 2 message would look something like this:

```
4e544c4d53535000020000000000000000000000202000000123456789abcdef
```

This message contains the NTLMSSP signature, the NTLM message type, an empty target name, minimal flags (Negotiate NTLM and Negotiate OEM), and the challenge.

Type 2 Message Example

Let's look at the following hexadecimal Type 2 Message:

```
4e544c4d53535000020000000c000c003000000001028100
0123456789abcdef0000000000000000620062003c000000
44004f004d00410049004e0002000c0044004f004d004100
49004e0001000c00530045005200560045005200040014000
64006f006d00610069006e002e0063006f006d0003002200
730065007200760065007200 2e0064006f006d0061006900
6e002e0063006f006d0000000000
```

Breaking this into its constituent fields gives:

0	`0x4e544c4d53535000`	NTLMSSP Signature
8	`0x02000000`	Type 2 Indicator
12	`0x0c000c0030000000`	Target Name Security Buffer: Length: 12 bytes (`0x0c00`) Allocated Space: 12 bytes (`0x0c00`) Offset: 48 bytes (`0x30000000`)
20	`0x01028100`	Flags: Negotiate Unicode (`0x00000001`) Negotiate NTLM (`0x00000200`) Target Type Domain (`0x00010000`) Negotiate Target Info (`0x00800000`)
24	`0x0123456789abcdef`	Challenge

32	`0x0000000000000000`	Context
40	`0x620062003c000000`	Target Information Security Buffer: Length: 98 bytes (`0x6200`) Allocated Space: 98 bytes (`0x6200`) Offset: 60 bytes (`0x3c000000`)
48	`0x44004f004d004100` `49004e00`	Target Name Data ("`DOMAIN`")

60	`0x02000c0044004f00`
	`4d00410049004e00`
	`01000c0053004500`
	`5200560045005200`
	`0400140064006f00`
	`6d00610069006e00`
	`2e0063006f006d00`
	`0300220073006500`
	`7200760065007200`
	`2e0064006f006d00`
	`610069006e002e00`
	`63006f006d000000`
	`0000`
		Target Information Data:

`0x02000c0044004f00` `4d00410049004e00`	Domain name subblock: Type: 2 (Domain name, `0x0200`) Length: 12 bytes (`0x0c00`) Data: "`DOMAIN`"
`0x01000c0053004500` `5200560045005200`	Server name subblock: Type: 1 (Server name, `0x0100`) Length: 12 bytes (`0x0c00`) Data: "`SERVER`"

`0x0400140064006f00`
`6d00610069006e00`
`2e0063006f006d00`
	DNS domain name subblock:
	
	Type: 4 (DNS domain name, `0x0400`)
	Length: 20 bytes (`0x1400`)
	Data: "`domain.com`"

`0x0300220073006500`
`7200760065007200`
`2e0064006f006d00`
`610069006e002e00`
`63006f006d00`
	DNS server name subblock:
	
	Type: 3 (DNS server name, `0x0300`)
	Length: 34 bytes (`0x2200`)
	Data: "`server.domain.com`"

`0x00000000`	Terminator subblock: Type: 0 (terminator, `0x0000`) Length: 0 bytes (`0x0000`)

An analysis of this message shows:

- This is an NTLM Type 2 message (from the NTLMSSP Signature and Type 2 Indicator).
- The server has indicated that strings will be encoded using Unicode (the Negotiate Unicode flag is set).
- The server supports NTLM authentication (Negotiate NTLM).
- The Target Name provided by the server is populated and represents a domain (the Target Type Domain flag is set and the domain name is present in the Target Name Security Buffer).
- The server is providing a Target Information structure (Negotiate Target Info is set). This structure is present in the Target Information Security Buffer (domain name "`DOMAIN`", server name "`SERVER`", DNS domain name "`domain.com`", and DNS server name "`server.domain.com`").
- The challenge generated by the server is "`0x0123456789abcdef`".
- An empty context has been sent.

Note that the target name is in Unicode format (as specified by the Negotiate Unicode flag).

After the server creates the Type 2 message, it is sent to the client. The response to the server's challenge is provided in the client's Type 3 message.

The Type 3 Message

	Description	Content
0	NTLMSSP Signature	Null-terminated ASCII "NTLMSSP" (`0x4e544c4d53535000`)
8	NTLM Message Type	`long` (`0x03000000`)
12	LM/LMv2 Response	security buffer
20	NTLM/NTLMv2 Response	security buffer
28	Target Name	security buffer
36	User Name	security buffer
44	Workstation Name	security buffer
(52)	Session Key *(optional)*	security buffer
(60)	Flags *(optional)*	`long`
(64)	OS Version Structure *(Optional)*	8 bytes
52 *(64)* *(72)*	*start of data block*	

The Type 3 message is the final step in authentication. This message contains the client's responses to the Type 2 challenge, which demonstrate that the client has knowledge of the account password without sending the password directly. The Type 3 message also indicates the authentication target (domain or server name) and username of the authenticating account, as well as the client workstation name.

Note that the flags in the Type 3 message are optional; older clients include neither the session key nor the flags in the message. It has been determined experimentally that the Type 3 flags (when included) do not carry any additional semantics in connection-oriented authentication; they do not appear to have any discernable effect on either authentication or the establishment of session security. Clients sending flags typically mirror the established Type 2 settings fairly closely. It is possible that the flags are sent as a "reminder" of established options, to allow the server to avoid caching the negotiated settings. The Type 3 flags are relevant during datagram-style authentication, however.

The LM/LMv2 and NTLM/NTLMv2 responses are security buffers containing replies created from the user's password in response to the Type 2 challenge; the process for generating these responses is outlined in the next section.

The target name is a security buffer containing the authentication realm in which the authenticating account has membership (a domain name for domain accounts, or server name for local machine accounts). This is either Unicode or OEM, depending on the negotiated encoding.

The user name is a security buffer containing the authenticating account name. This is either Unicode or OEM, depending on the negotiated encoding.

The workstation name is a security buffer containing the client workstation's name. This is either Unicode or OEM, depending on the negotiated encoding.

The session key value is used by the session security mechanism during key exchange; this is discussed in more detail in the Session Security section.

When "Negotiate Local Call" has been established in the Type 2 message, the security buffers in the Type 3 message are typically all empty (zero length). The client "adopts" the SSPI context sent in the Type 2 message, effectively circumventing the need to calculate an appropriate response.

The OS Version structure is the same format described previously.

Again, there are a few variations of the Type 3 message:

1. Version 1 -- The Session Key, flags, and OS Version structure are omitted. The data block in this case starts after the Workstation Name security buffer header, at offset 52. This form is seen in older Win9x-based systems.

2. Version 2 -- The Session Key and flags are included, but the OS Version structure is not. In this case, the data block begins after the flags field, at offset 64. This form is seen in most out-of-box shipping versions of Windows, and is roughly documented in the Open Group's ActiveX reference documentation (Section 11.2.4).

3. Version 3 -- The Session Key, flags, and OS Version structure are all present. The data block begins after the OS Version structure, at offset 72. This form was introduced in a relatively recent Service Pack, and is seen on currently-patched versions of Windows 2000, Windows XP, and Windows 2003.

Name Variations

In addition to the variations in message layout, user and target names can be presented in a few different formats within the Type 3 message. In the typical scenario, the User Name field is populated with the Windows account name, and the Target Name is populated with the NT domain name. However, the username and/or domain can also be presented in the Kerberos-style "user@domain.com" format in various combinations. It has been observed that several variations are supported, with some possible implications/caveats:

Format	Type 3 Field Content	Notes
`DOMAIN\user`	User Name = "`user`" Target Name = "`DOMAIN`"	This is the "normal" format; the User Name field contains the Windows user name, and the Target Name contains the NT-style NetBIOS domain or server name.
`domain.com\user`	User Name = "`user`" Target Name = "`domain.com`"	Here, the Target Name field in the Type 3 message is populated with the DNS domain/realm name (or fully-qualified DNS host name in the case of local machine accounts).
`user@DOMAIN`	User Name = "`user@DOMAIN`" Target Name is empty	In this case, the Target Name field is empty (zero-length), and the User Name field uses the Kerberos-style "user@realm" format; however, the NetBIOS domain name is used instead of the DNS domain. It has been observed that this format is not supported for local machine accounts; additionally, this form does not appear to be supported under NTLMv2/LMv2 authentication.

`user@domain.com`	User Name = "`user@domain.com`" Target Name is empty	Here, the Target Name field in the Type 3 message is empty; the User Name field contains the Kerberos-style "user@realm" format, with the DNS domain. This form does not appear to be supported for local machine accounts.

Responding to the Challenge

The client creates one or more responses to the Type 2 challenge, and sends these to the server in the Type 3 message. There are six types of responses:

- LM (LAN Manager) Response - Sent by most older clients, this is the "original" response type.
- NTLM Response - This is sent by NT-based clients, including Windows 2000 and XP.
- NTLMv2 Response - A newer response type, introduced in Windows NT Service Pack 4. This replaces the NTLM response on systems that have NTLM version 2 enabled.
- LMv2 Response - The replacement for the LM response on NTLM version 2 systems.
- NTLM2 Session Response - Used when NTLM2 session security is negotiated without NTLMv2 authentication, this scheme alters the semantics of both the LM and NTLM responses.
- Anonymous Response - This is used when an anonymous context is being established; actual credentials are not presented, and no true authentication takes place. "Stub" fields are presented in the Type 3 message.

For more information on these schemes, it is highly recommended that you read Christopher Hertel's *Implementing CIFS*, especially the section on authentication.

The responses serve as an indirect proof of knowledge of the password. The password is used by the client to derive the LM and/or NTLM hash (discussed in the next section); these values are in turn used to calculate an appropriate response to the challenge. The domain controller (or server for local machine accounts) stores the LM and NTLM hashes for the password; when the response is received from the client, these stored values are used to calculate the appropriate response values which are compared to those sent by the client. A match yields a successful authentication of the user.

Note that unlike Unix password hashes, the LM and NTLM hash are password-equivalents in the context of the response calculations; they must be protected, as they can be used to authenticate users across the network even without knowledge of the actual password itself.

The LM Response

The LM response is sent by most clients. This scheme is older than the NTLM response, and less secure. While newer clients support the NTLM response, they typically send both responses for compatibility with legacy servers; hence, the security flaws present in the LM response are still exhibited in many clients supporting the NTLM response.

The LM response is calculated as follows (see Appendix D for a sample implementation in Java):

1. The user's password (as an OEM string) is converted to uppercase.
2. This password is null-padded to 14 bytes.
3. This "fixed" password is split into two 7-byte halves.
4. These values are used to create two DES keys (one from each 7-byte half).
5. Each of these keys is used to DES-encrypt the constant ASCII string "`KGS!@#$%`" (resulting in two 8-byte ciphertext values).
6. These two ciphertext values are concatenated to form a 16-byte value - the LM hash.
7. The 16-byte LM hash is null-padded to 21 bytes.
8. This value is split into three 7-byte thirds.
9. These values are used to create three DES keys (one from each 7-byte third).

10. Each of these keys is used to DES-encrypt the challenge from the Type 2 message (resulting in three 8-byte ciphertext values).
11. These three ciphertext values are concatenated to form a 24-byte value. This is the LM response.

In the event that the user's password is longer than 15 characters, the host or domain controller will not store the LM hash for the user; the LM response cannot be used to authenticate the user in this case. A response is still generated and placed in the LM Response field, using a 16-byte null value (`0x00000000000000000000000000000000`) as the LM hash in the calculation. This value is ignored by the target.

The response calculation process is best illustrated with a detailed example. Consider a user with the password "`SecREt01`", responding to the Type 2 challenge "`0x0123456789abcdef`".

1. The password (as an OEM string) is converted to uppercase, giving "`SECRET01`" (or "`0x5345435245543031`" in hexadecimal).
2. This password is null-padded to 14 bytes, giving "`0x53454352455430310000000000000`".
3. This value is split into two 7-byte halves, "`0x53454352455430`" and "`0x31000000000000`".
4. These two values are used to create two DES keys. A DES key is 8 bytes long; each byte contains seven bits of key material and one odd-parity bit (the parity bit may or may not be checked, depending on the underlying DES implementation). Our first 7-byte value, "`0x53454352455430`", would be represented in binary as:

   `01010011 01000101 01000011 01010010 01000101 01010100 00110000`

   A non-parity-adjusted DES key for this value would be:

   `0101001`0 `1010001`0 `0101000`0 `0110101`0 `0010010`0 `0010101`0 `0101000`0 `0110000`0

   (the parity bits are shown in red above). This is "`0x52a2506a242a5060`" in hexadecimal. Applying odd-parity to ensure that the total number of set bits in each octet is odd gives:

   `0101001`0 `1010001`0 `0101000`1 `0110101`1 `0010010`1 `0010101`0 `0101000`1 `0110000`1

   This is the first DES key ("`0x52a2516b252a5161`" in hex). We then apply the same process to our second 7-byte value, "`0x31000000000000`", represented in binary as:

   `00110001 00000000 00000000 00000000 00000000 00000000 00000000`

   Creating a non-parity-adjusted DES key gives:

   `0011000`0 `1000000`0 `0000000`0 `0000000`0 `0000000`0 `0000000`0 `0000000`0 `0000000`0

   ("`0x3080000000000000`" in hexadecimal). Adjusting the parity bits gives:

   `0011000`1 `1000000`0 `0000000`1 `0000000`1 `0000000`1 `0000000`1 `0000000`1 `0000000`1

   This is our second DES key, "`0x3180010101010101`" in hexadecimal. Note that if our particular DES implementation does not enforce parity (many do not), the parity-adjustment steps can be skipped; the non-parity-adjusted values would then be used as the DES keys. In any case, the parity bits will not affect the encryption process.

5. Each of our keys is used to DES-encrypt the constant ASCII string "`KGS!@#$%`" ("`0x4b47532140232425`" in hex). This gives us "`0xff3750bcc2b22412`" (using the first key) and "`0xc2265b23734e0dac`" (using the second).
6. These ciphertext values are concatenated to form our 16-byte LM hash - "`0xff3750bcc2b22412c2265b23734e0dac`".
7. This is null-padded to 21 bytes, giving "`0xff3750bcc2b22412c2265b23734e0dac0000000000`".
8. This value is split into three 7-byte thirds, "`0xff3750bcc2b224`", "`0x12c2265b23734e`" and

"0x0dac0000000000".

9. These three values are used to create three DES keys. Using the process outlined previously, our first value:

   11111111 00110111 01010000 10111100 11000010 10110010 00100100

   Gives us the parity-adjusted DES key:

   11111110 10011011 11010101 00010110 11001101 00010101 11001000 01001001

   ("0xfe9bd516cd15c849" in hexadecimal). The second value:

   00010010 11000010 00100110 01011011 00100011 01110011 01001110

   Results in the key:

   00010011 01100001 10001001 11001011 10110011 00011010 11001101 10011101

   ("0x136189cbb31acd9d"). Finally, the third value:

   00001101 10101100 00000000 00000000 00000000 00000000 00000000

   Gives us:

   00001101 11010110 00000001 00000001 00000001 00000001 00000001 00000001

   This is the third DES key ("0x0dd6010101010101").

10. Each of the three keys is used to DES-encrypt the challenge from the Type 2 message (in our example, "0x0123456789abcdef"). This gives the results "0xc337cd5cbd44fc97" (using the first key), "0x82a667af6d427c6d" (using the second) and "0xe67c20c2d3e77c56" (using the third).
11. These three ciphertext values are concatenated to form the 24-byte LM response:

    0xc337cd5cbd44fc9782a667af6d427c6de67c20c2d3e77c56

There are several weaknesses in this algorithm which make it susceptible to attack. While these are covered in detail in the Hertel text, the most prominent problems are:

- Passwords are converted to upper case before calculating the response. This significantly reduces the set of possible passwords that must be tested in a brute-force attack.
- If the password is seven or fewer characters, the second value from step 3 above will be 7 null bytes. This effectively compromises half of the LM hash (as it will always be the ciphertext of "KGS!@#$%" encrypted with the DES key "0x0101010101010101" - the constant "0xaad3b435b51404ee"). This in turn compromises the three DES keys used to produce the response; the entire third key and all but one byte of the second will be known constant values.

The NTLM Response

The NTLM response is sent by newer clients. This scheme addresses some of the flaws in the LM response; however, it is still considered fairly weak. Additionally, the NTLM response is nearly always sent in conjunction with the LM response. The weaknesses in that algorithm can be exploited to obtain the case-insensitive password, and trial-and-error used to find the case-sensitive password employed by the NTLM response.

The NTLM response is calculated as follows (see Appendix D for a sample Java implementation):

1. The MD4 message-digest algorithm (described in RFC 1320) is applied to the Unicode mixed-case password. This results in a 16-byte value - the NTLM hash.

2. The 16-byte NTLM hash is null-padded to 21 bytes.
3. This value is split into three 7-byte thirds.
4. These values are used to create three DES keys (one from each 7-byte third).
5. Each of these keys is used to DES-encrypt the challenge from the Type 2 message (resulting in three 8-byte ciphertext values).
6. These three ciphertext values are concatenated to form a 24-byte value. This is the NTLM response.

Note that only the calculation of the hash value differs from the LM scheme; the response calculation is the same. To illustrate this process, we will apply it to our previous example (a user with the password "`SecREt01`", responding to the Type 2 challenge "`0x0123456789abcdef`").

1. The Unicode mixed-case password is "`0x5300650063005200450074003000 3100`" in hexadecimal; the MD4 hash of this value is calculated, giving "`0xcd06ca7c7e10c99b1d33b7485a2ed808`". This is the NTLM hash.
2. This is null-padded to 21 bytes, giving "`0xcd06ca7c7e10c99b1d33b7485a2ed8080000000000`".
3. This value is split into three 7-byte thirds, "`0xcd06ca7c7e10c9`", "`0x9b1d33b7485a2e`" and "`0xd8080000000000`".
4. These three values are used to create three DES keys. Our first value:

   `11001101 00000110 11001010 01111100 01111110 00010000 11001001`

   Results in the parity-adjusted key:

   `11001101 10000011 10110011 01001111 11000111 11110001 01000011 10010010`

   ("`0xcd83b34fc7f14392`" in hexadecimal). The second value:

   `10011011 00011101 00110011 10110111 01001000 01011010 00101110`

   Gives the key:

   `10011011 10001111 01001100 01110110 01110101 01000011 01101000 01011101`

   ("`0x9b8f4c767543685d`"). Our third value:

   `11011000 00001000 00000000 00000000 00000000 00000000 00000000`

   Yields our third key:

   `11011001 00000100 00000001 00000001 00000001 00000001 00000001 00000001`

   ("`0xd904010101010101`" in hexadecimal).

5. Each of the three keys is used to DES-encrypt the challenge from the Type 2 message ("`0x0123456789abcdef`"). This yields the results "`0x25a98c1c31e81847`" (using our first key), "`0x466b29b2df4680f3`" (using the second) and "`0x9958fb8c213a9cc6`" (using the third key).
6. These three ciphertext values are concatenated to form the 24-byte NTLM response:

   `0x25a98c1c31e81847466b29b2df4680f39958fb8c213a9cc6`

The NTLMv2 Response

NTLM version 2 ("NTLMv2") was concocted to address the security issues present in NTLM. When NTLMv2 is enabled, the NTLM response is replaced with the NTLMv2 response, and the LM response is replaced with the LMv2 response (which we will discuss next).

The NTLMv2 response is calculated as follows (see Appendix D for a sample implementation in Java):

1. The NTLM password hash is obtained (as discussed previously, this is the MD4 digest of the Unicode mixed-case password).
2. The Unicode uppercase username is concatenated with the Unicode authentication target (the domain or server name specified in the Target Name field of the Type 3 message). Note that this calculation always uses the Unicode representation, even if OEM encoding has been negotiated; also note that the username is converted to uppercase, while the authentication target is case-sensitive and must match the case presented in the Target Name field.
   The HMAC-MD5 message authentication code algorithm (described in RFC 2104) is applied to this value using the 16-byte NTLM hash as the key. This results in a 16-byte value - the NTLMv2 hash.
3. A block of data known as the "blob" is constructed. The Hertel text discusses the format of this structure in greater detail; briefly:

	Description	Content
0	Blob Signature	`0x01010000`
4	Reserved	`long` (`0x00000000`)
8	Timestamp	Little-endian, 64-bit signed value representing the number of tenths of a microsecond since January 1, 1601.
16	Client Nonce	8 bytes
24	Unknown	4 bytes
28	Target Information	Target Information block (from the Type 2 message).
(variable)	Unknown	4 bytes

4. The challenge from the Type 2 message is concatenated with the blob. The HMAC-MD5 message authentication code algorithm is applied to this value using the 16-byte NTLMv2 hash (calculated in step 2) as the key. This results in a 16-byte output value.
5. This value is concatenated with the blob to form the NTLMv2 response.

Let's look at an example. Since we need a bit more information to calculate the NTLMv2 response, we will use the following values from the examples presented previously:

Target:	`DOMAIN`
Username:	`user`
Password:	`SecREt01`
Challenge:	`0x0123456789abcdef`

Target Information:	`0x02000c0044004f00`
	`4d00410049004e00`
	`01000c0053004500`
	`5200560045005200`
	`0400140064006f00`
	`6d00610069006e00`
	`2e0063006f006d00`
	`0300220073006500`
	`7200760065007200`
	`2e0064006f006d00`
	`610069006e002e00`
	`63006f006d000000`
	`0000`

1. The Unicode mixed-case password is "`0x5300650063005200450074003000 3100`" in hexadecimal; the

MD4 hash of this value is calculated, giving "`0xcd06ca7c7e10c99b1d33b7485a2ed808`". This is the NTLM hash.

2. The Unicode uppercase username is concatenated with the Unicode authentication target, giving "`USERDOMAIN`" (or "`0x550053004500520044004f004d00410049004e00`" in hexadecimal). HMAC-MD5 is applied to this value using the 16-byte NTLM hash from the previous step as the key, which yields "`0x04b8e0ba74289cc540826bab1dee63ae`". This is the NTLMv2 hash.
3. Next, the blob is constructed. The timestamp is the most tedious part of this; looking at the clock on my desk, it's about 6:00 AM EDT on June 17th, 2003. In Unix time, that would be 1055844000 seconds after the Epoch. Adding 11644473600 will give us seconds after January 1, 1601 (12700317600). Multiplying by 10^7 (10000000) will give us tenths of a microsecond (127003176000000000). As a little-endian 64-bit value, this is "`0x0090d336b734c301`" (in hexadecimal).

   We also need to generate an 8-byte random "client nonce"; we will use the not-so-random "`0xffffff0011223344`". Constructing the rest of the blob is easy; we just concatenate:

`0x01010000`	(the blob signature)
`0x00000000`	(reserved value)
`0x0090d336b734c301`	(our timestamp)
`0xffffff0011223344`	(a random client nonce)
`0x00000000`	(unknown, but zero will work)

`0x02000c0044004f00`
`4d00410049004e00`
`01000c0053004500`
`5200560045005200`
`0400140064006f00`
`6d00610069006e00`
`2e0063006f006d00`
`0300220073006500`
`7200760065007200`
`2e0064006f006d00`
`610069006e002e00`
`63006f006d000000`
`0000`
	(our target information block)

`0x00000000`	(unknown, but zero will work)

4. We then concatenate the Type 2 challenge with our blob:

   ```
   0x0123456789abcdef0101000000000000
     0090d336b734c301ffffff0011223344
     0000000002000c0044004f004d004100
     49004e0001000c00530045005200560
     4500520004001400640060060061006
     69006e002e0063006f006d0003002200
     7300650072007600650072002e006400
     6f006d00610069006e002e0063006f00
     6d000000000000000000
   ```

   Applying HMAC-MD5 to this value using the NTLMv2 hash from step 2 as the key gives us the 16-byte value "`0xcbabbca713eb795d04c97abc01ee4983`".

5. This value is concatenated with the blob to obtain the NTLMv2 response:

   ```
   0xcbabbca713eb795d04c97abc01ee4983
     0101000000000000090d336b734c301
   ```

```
ffffff0011223344000000002000c00
44004f004d00410049004e0001000c00
5300450052005600450052000400140 0
64006f006d00610069006e002e006300
6f006d0003002200730065007200760 0
650072002e0064006f006d0061006900
6e002e0063006f006d00000000000000 0
0000
```

The LMv2 Response

The LMv2 response is used to provide pass-through authentication compatibility with older servers. It is quite possible that the server with which the client is communicating will not actually perform the authentication; rather, it will pass the responses through to a domain controller for verification. Older servers pass only the LM response, and expect it to be exactly 24 bytes. The LMv2 response was designed to allow such servers to operate properly; it is effectively a "miniature" NTLMv2 response, obtained as follows (see Appendix D for a sample Java implementation):

1. The NTLM password hash is calculated (the MD4 digest of the Unicode mixed-case password).
2. The Unicode uppercase username is concatenated with the Unicode authentication target (domain or server name) presented in the Target Name field of the Type 3 message. The HMAC-MD5 message authentication code algorithm is applied to this value using the 16-byte NTLM hash as the key. This results in a 16-byte value - the NTLMv2 hash.
3. A random 8-byte client nonce is created (this is the same client nonce used in the NTLMv2 blob).
4. The challenge from the Type 2 message is concatenated with the client nonce. The HMAC-MD5 message authentication code algorithm is applied to this value using the 16-byte NTLMv2 hash (calculated in step 2) as the key. This results in a 16-byte output value.
5. This value is concatenated with the 8-byte client nonce to form the 24-byte LMv2 response.

We will illustrate this process with a brief example using our tried-and-true sample values:

Target:	`DOMAIN`
Username:	`user`
Password:	`SecREt01`
Challenge:	`0x0123456789abcdef`

1. The Unicode mixed-case password is "`0x5300650063005200450074003000 3100`" in hexadecimal; the MD4 hash of this value is calculated, giving "`0xcd06ca7c7e10c99b1d33b7485a2ed808`". This is the NTLM hash.
2. The Unicode uppercase username is concatenated with the Unicode authentication target, giving "`USERDOMAIN`" (or "`0x55005300450052004400 4f004d00410049004e00`" in hexadecimal). HMAC-MD5 is applied to this value using the 16-byte NTLM hash from the previous step as the key, which yields "`0x04b8e0ba74289cc540826bab1dee63ae`". This is the NTLMv2 hash.
3. A random 8-byte client nonce is created. From our NTLMv2 example, we will use "`0xffffff0011223344`".
4. We then concatenate the Type 2 challenge with our client nonce:

   `0x0123456789abcdefffffff0011223344`

   Applying HMAC-MD5 to this value using the NTLMv2 hash from step 2 as the key gives us the 16-byte value "`0xd6e6152ea25d03b7c6ba6629c2d6aaf0`".

5. This value is concatenated with the client nonce to obtain the 24-byte LMv2 response:

```
0xd6e6152ea25d03b7c6ba6629c2d6aaf0ffffff0011223344
```

The NTLM2 Session Response

The NTLM2 session response can be employed in conjunction with NTLM2 session security (it is made available with the "Negotiate NTLM2 Key" flag). This is used to provide enhanced protection against precomputed dictionary attacks (particularly Rainbow Table-based attacks) in environments which do not support full NTLMv2 authentication.

The NTLM2 session response replaces both the LM and NTLM response fields as follows (see Appendix D for a sample implementation in Java):

1. A random 8-byte client nonce is created.
2. The client nonce is null-padded to 24 bytes. This value is placed in the LM response field of the Type 3 message.
3. The challenge from the Type 2 message is concatenated with the 8-byte client nonce to form a session nonce.
4. The MD5 message-digest algorithm (described in RFC 1321) is applied to the session nonce, resulting in a 16-byte value.
5. This value is truncated to 8 bytes to form the NTLM2 session hash.
6. The NTLM password hash is obtained (as discussed, this is the MD4 digest of the Unicode mixed-case password).
7. The 16-byte NTLM hash is null-padded to 21 bytes.
8. This value is split into three 7-byte thirds.
9. These values are used to create three DES keys (one from each 7-byte third).
10. Each of these keys is used to DES-encrypt the NTLM2 session hash (resulting in three 8-byte ciphertext values).
11. These three ciphertext values are concatenated to form a 24-byte value. This is the NTLM2 session response, which is placed in the NTLM response field of the Type 3 message.

To demonstrate this with our previous example values (a user with the password "`SecREt01`", responding to the Type 2 challenge "`0x0123456789abcdef`"):

1. A random 8-byte client nonce is created; we will use "`0xffffff0011223344`", as in the previous examples.
2. The challenge is null-padded to 24 bytes:

   ```
   0xffffff0011223344000000000000000000000000000000
   ```

   This value is placed in the LM response field of the Type 3 message.

3. The challenge from the Type 2 message is concatenated with the client nonce, forming a session nonce ("`0x0123456789abcdefffffff0011223344`").
4. Applying the MD5 digest to this nonce yields the 16-byte value "`0xbeac9a1bc5a9867c15192b3105d5beb1`".
5. This is truncated to 8 bytes to obtain the NTLM2 session hash ("`0xbeac9a1bc5a9867c`").
6. The Unicode mixed-case password is "`0x5300650063005200450074003000310 0`"; applying the MD4 digest to this value gives us the NTLM hash ("`0xcd06ca7c7e10c99b1d33b7485a2ed808`").
7. This is null-padded to 21 bytes, giving "`0xcd06ca7c7e10c99b1d33b7485a2ed8080000000000`".
8. This value is split into three 7-byte thirds, "`0xcd06ca7c7e10c9`", "`0x9b1d33b7485a2e`" and "`0xd8080000000000`".
9. These values are used to create three DES keys (as calculated in our previous NTLM response example, "`0xcd83b34fc7f14392`", "`0x9b8f4c767543685d`", and "`0xd904010101010101`").
10. Each of these three keys is used to DES-encrypt the NTLM2 session hash ("`0xbeac9a1bc5a9867c`"). This yields the results "`0x10d550832d12b2cc`" (using our first key), "`0xb79d5ad1f4eed3df`" (using the second), and "`0x82aca4c3681dd455`" (using the third key).

11. These three ciphertext values are concatenated to form the 24-byte NTLM2 session response:

    `0x10d550832d12b2ccb79d5ad1f4eed3df82aca4c3681dd455`

    which is placed in the NTLM response field of the Type 3 message.

The Anonymous Response

The Anonymous Response is seen when the client is establishing an anonymous context, rather than a true user-based context. This is typically seen when a "placeholder" is needed for operations that do not require an authenticated user. Anonymous connections are not the same as the Windows "Guest" user (the latter is an actual user account, while anonymous connections are associated with no account at all).

In an anonymous Type 3 message, the client indicates the "Negotiate Anonymous" flag; the NTLM response field is empty (zero-length); and the LM response field contains a single null byte ("`0x00`").

Type 3 Message Example

Now that we're familiar with the Type 3 responses, we are ready to examine a Type 3 Message:

```
4e544c4d535350000300000180018006a00000018001800
820000000c000c00400000000800080004c00000016001600
540000000000000009a0000001020000044004f004d004100
49004e00750073006500720057004f0052004b0053005400
4100540049004f004e00c337cd5cbd44fc9782a667af6d42
7c6de67c20c2d3e77c5625a98c1c31e81847466b29b2df46
80f39958fb8c213a9cc6
```

This message is decomposed as:

0	`0x4e544c4d53535000`	NTLMSSP Signature
8	`0x03000000`	Type 3 Indicator
12	`0x180018006a000000`	LM Response Security Buffer: Length: 24 bytes (`0x1800`) Allocated Space: 24 bytes (`0x1800`) Offset: 106 bytes (`0x6a000000`)
20	`0x1800180082000000`	NTLM Response Security Buffer: Length: 24 bytes (`0x1800`) Allocated Space: 24 bytes (`0x1800`) Offset: 130 bytes (`0x82000000`)
28	`0x0c000c0040000000`	Target Name Security Buffer: Length: 12 bytes (`0x0c00`) Allocated Space: 12 bytes (`0x0c00`) Offset: 64 bytes (`0x40000000`)
36	`0x080008004c000000`	User Name Security Buffer: Length: 8 bytes (`0x0800`) Allocated Space: 8 bytes (`0x0800`) Offset: 76 bytes (`0x4c000000`)

44	0x1600160054000000	Workstation Name Security Buffer: Length: 22 bytes (0x1600) Allocated Space: 22 bytes (0x1600) Offset: 84 bytes (0x54000000)
52	0x000000009a000000	Session Key Security Buffer: Length: 0 bytes (0x0000) Allocated Space: 0 bytes (0x0000) Offset: 154 bytes (0x9a000000)
60	0x01020000	Flags: Negotiate Unicode (0x00000001) Negotiate NTLM (0x00000200)
64	0x44004f004d004100 49004e00	Target Name Data ("DOMAIN")
76	0x7500730065007200	User Name Data ("user")
84	0x57004f0052004b00 5300540041005400 49004f004e00	Workstation Name Data ("WORKSTATION")
106	0xc337cd5cbd44fc97 82a667af6d427c6d e67c20c2d3e77c56	LM Response Data
130	0x25a98c1c31e81847 466b29b2df4680f3 9958fb8c213a9cc6	NTLM Response Data

Analysis of this reveals:

- This is an NTLM Type 3 message (from the NTLMSSP Signature and Type 3 Indicator).
- The client has indicated that strings are encoded using Unicode (the Negotiate Unicode flag is set).
- The client supports NTLM authentication (Negotiate NTLM).
- The client's domain is "DOMAIN".
- The client's username is "user".
- The client's workstation is "WORKSTATION".
- The client's LM response is "0xc337cd5cbd44fc9782a667af6d427c6de67c20c2d3e77c56".
- The client's NTLM response is "0x25a98c1c31e81847466b29b2df4680f39958fb8c213a9cc6".
- An empty session key has been sent.

Upon receipt of the Type 3 message, the server calculates the LM and NTLM responses and compares them to the values provided by the client; if they match, the user is successfully authenticated.

NTLM Version 2

NTLM version 2 consists of three new response algorithms (NTLMv2, LMv2, and the NTLM2 session response, discussed previously) and a new signing and sealing scheme (NTLM2 session security). NTLM2 session security is negotiated via the "Negotiate NTLM2 Key" flag; NTLMv2 authentication, however, is enabled through a modification to the registry. Further, the registry setting on the client and

domain controller must be compatible in order for authentication to be successful (although it is possible for NTLMv2 authentication to pass through an older server to an NTLMv2 domain controller). The result of the configuration and planning required to deploy NTLMv2 is that many hosts just use the default setting (NTLMv1), and NTLMv2 authentication is underutilized.

Instructions for enabling NTLM version 2 are detailed in Microsoft Knowledge Base Article 239869; briefly, a modification is made to the registry value:

```
HKEY_LOCAL_MACHINE\System\CurrentControlSet\Control\LSA\LMCompatibilityLevel
```

(`LMCompatibility` on Win9x-based systems). This is a `REG_DWORD` entry, and can be set to one of the following values:

Level	Sent by Client	Accepted by Server
0	LM NTLM	LM NTLM LMv2 NTLMv2
1	LM NTLM	LM NTLM LMv2 NTLMv2
2	NTLM	LM NTLM LMv2 NTLMv2
3	LMv2 NTLMv2	LM NTLM LMv2 NTLMv2
4	LMv2 NTLMv2	NTLM LMv2 NTLMv2
5	LMv2 NTLMv2	LMv2 NTLMv2

In all levels, NTLM2 session security is supported and negotiated when available (most available documentation indicates that NTLM2 session security is only enabled on levels 1 and above, but it is seen in practice with Level 0 as well). By default, only the LM response is supported on Windows 95 and Windows 98 platforms; installing the Directory Services client makes NTLMv2 available on these hosts as well (and enables the `LMCompatibility` setting, although only levels 0 and 3 are available).

In Level 2, clients send the NTLM response twice (in both the LM and NTLM response fields). At Level 3 and higher, the LMv2 and NTLMv2 responses replace the LM and NTLM responses, respectively.

When NTLM2 session security has been negotiated (indicated by the "Negotiate NTLM2 Key" flag), the NTLM2 session response can be used in Levels 0, 1, and 2 as a replacement for the weaker LM and NTLM responses. This offers heightened protection over NTLMv1 against server-based precomputed dictionary attacks; the client's response to a given challenge is made variable by adding a random client nonce to the calculation.

The NTLM2 session response is interesting in that it can be negotiated between a client and server that

support the newer schemes, even in the presence of an older domain controller that does not. In a typical scenario, the server in an authentication transaction does not actually possess the user's password hash; that is instead held at the domain controller. When a machine is joined to an NT-style domain, it establishes an encrypted, mutually-authenticated channel to the domain controller (colloquially deemed the "NetLogon pipe"). When a client authenticates to the server using the "vanilla" NTLMv1 handshake, the following transactions occur in the background:

1. The client sends the Type 1 message, containing flags and other information as discussed previously.
2. The server generates a challenge for the client and sends the Type 2 message containing the negotiated flag set.
3. The client responds to the challenge, providing the LM/NTLM responses.
4. The server sends the challenge and client responses over the NetLogon pipe to the domain controller.
5. The domain controller uses the stored hashes and the challenge given by the server to reproduce the authentication calculations; if they match the responses, the authentication is successful.
6. The domain controller calculates and sends the session key to the server, which can be used for subsequent signing and sealing operations between the server and the client.

In the case of the NTLM2 Session Response, it is possible that a client and server have been upgraded to allow the newer protocol, but the domain controller has not. To allow for this contingency, the handshake described above is modified slightly as follows:

1. The client sends the Type 1 message, in this case indicating the "Negotiate NTLM2 Key" flag.
2. The server generates a challenge for the client and sends the Type 2 message containing the negotiated flag set (also including the "Negotiate NTLM2 Key" flag).
3. The client responds to the challenge, providing the client nonce in the LM field, and the NTLM2 Session Response in the NTLM field. Note that the latter is exactly the same calculation as the NTLM response, except instead of encrypting the server challenge the client has encrypted the MD5 hash of the server challenge concatenated with the client nonce.
4. Instead of sending the server challenge directly over the NetLogon pipe to the domain controller, the server sends the MD5 hash of the server challenge concatenated with the client nonce (lifted from the LM response field). Additionally it sends the client responses (as usual).
5. The domain controller encrypts the challenge field sent by the server using the stored hash as the key and notes that it matches the NTLM response field; hence, the client is successfully authenticated.
6. The domain controller calculates and sends the normal NTLM User Session Key to the server; the server uses this in a secondary calculation to obtain the NTLM2 Session Response User Session Key (discussed in a subsequent section)

Essentially, this allows upgraded clients and servers to use the NTLM2 Session Response in networks where the domain controller has not yet been upgraded to NTLMv2 (or where the network administrator has not yet configured the `LMCompatibilityLevel` registry setting to use NTLMv2).

Related to the `LMCompatibilityLevel` setting are the `NtlmMinClientSec` and `NtlmMinServerSec` settings; these specify minimum requirements for NTLM contexts established by the NTLMSSP. Both are `REG_WORD` entries, and are bitfields specifying a combination of the following NTLM flags:

- Negotiate Sign (`0x00000010`) - Indicates the context must be established with support for message integrity (signing).
- Negotiate Seal (`0x00000020`) - Indicates the context must be established with support for message confidentiality (sealing).
- Negotiate NTLM2 Key (`0x00080000`) - Indicates the context must be established with NTLM2 session security.
- Negotiate 128 (`0x20000000`) - Indicates the context must support at least 128-bit signing/sealing

keys.
- Negotiate 56 (`0x80000000`) - Indicates the context must support at least 56-bit signing/sealing keys.

While most of these are more applicable to NTLM2 signing and sealing, the "Negotiate NTLM2 Key" is significant to authentication in that it can prevent sessions from being established with hosts that are unable to negotiate NTLM2 session security. This serves to ensure that the LM and NTLM responses are not sent (requiring that authentication will at least use the NTLM2 Session Response in all cases).

NTLMSSP and SSPI

At this point, we will start to look at how NTLM fits into the "big picture".

Windows provides a security framework known as SSPI - the Security Support Provider interface. This is the Microsoft equivalent of the GSS-API (Generic Security Service Application Program Interface, RFC 2743), and allows for a very high-level, mechanism-independent means of applying authentication, integrity, and confidentiality primitives. SSPI supports several underlying providers; one of these is the NTLMSSP (NTLM Security Support Provider), which provides the NTLM authentication mechanism we have been discussing thus far. SSPI supplies a flexible API for handling opaque, provider-specific authentication tokens; the NTLM Type 1, Type 2, and Type 3 messages are such tokens, specific to and processed by the NTLMSSP. The API provided by SSPI abstracts away almost all the details of NTLM. The application developer doesn't even have to be aware that NTLM is being used, and another authentication mechanism (such as Kerberos) can be swapped in with little or no changes at the application level.

We aren't going to delve too deeply into the SSPI framework, but this is a good point to look at the SSPI authentication handshake as applied to NTLM:

1. The client obtains a representation of the credential set for the user via the SSPI `AcquireCredentialsHandle` function.
2. The client calls the SSPI `InitializeSecurityContext` function to obtain an authentication request token (in our case, a Type 1 message). The client sends this token to the server. The return value from the function indicates that authentication will require multiple steps.
3. The server receives the token from the client, and uses it as input to the `AcceptSecurityContext` SSPI function. This creates a local security context on the server to represent the client, and yields an authentication response token (the Type 2 message), which is sent to the client. The return value from the function indicates that further information is needed from the client.
4. The client receives the response token from the server and calls `InitializeSecurityContext` again, passing the server's token as input. This provides us with another authentication request token (the Type 3 message). The return value indicates that the security context was successfully initialized; the token is sent to the server.
5. The server receives the token from the client and calls `AcceptSecurityContext` again, using the Type 3 message as input. The return value indicates the context was successfully accepted; no token is produced, and authentication is complete.

Local Authentication

We have alluded to the local authentication sequence at various points in our discussion; having a basic understanding of SSPI, we can look at this scenario in more detail.

Local authentication is negotiated through a series of decisions made by the client and server, based on the information in the NTLM messages. It works as follows:

1. The client calls the `AcquireCredentialsHandle` function, specifying the default credentials by passing in null to the "pAuthData" parameter. This obtains a handle to the credentials of the logged

in user for single sign-on.
2. The client calls the SSPI `InitializeSecurityContext` function to create the Type 1 message. When the default credential handle is supplied, the Type 1 message contains the workstation and domain name of the client. This is indicated by the presence of the "Negotiate Domain Supplied" and "Negotiate Workstation Supplied" flags, and the inclusion of populated Supplied Domain and Supplied Workstation security buffers in the message.
3. The server receives the Type 1 message from the client, and calls `AcceptSecurityContext`. This creates a local security context on the server to represent the client. The server examines the domain and workstation information sent by the client to determine if the client and server are the same machine. If so, the server initiates local authentication by setting the "Negotiate Local Call" flag in the resultant Type 2 message. The first `long` in the Context field of the Type 2 message is populated with the "upper" portion of the newly obtained SSPI context handle (specifically, the "dwUpper" field of the SSPI `CtxtHandle` structure). The second `long` in the Context field appears to be empty in all cases. (although logically one would assume it should contain the "lower" portion of the context handle).
4. The client receives the Type 2 message from the server and passes it to `InitializeSecurityContext`. Having noted the presence of the "Negotiate Local Call" flag, the client examines the server context handle to determine if it represents a valid local security context. If the context cannot be validated, authentication proceeds as usual - the appropriate responses are calculated, and included with the domain, workstation, and username in the Type 3 message. If the security context handle from the Type 2 message *can* be validated, however, no responses are prepared whatsoever. Instead, the default credentials are internally associated with the server context. The resulting Type 3 message is completely empty, containing zero-length security buffers for the responses as well as the username, domain, and workstation.
5. The server receives the Type 3 message and uses it as input to the `AcceptSecurityContext` function. The server verifies that the security context has been associated with a user; if so, authentication has successfully completed. If the context has not been bound to a user, authentication fails.

Datagram Authentication

Datagram-style authentication is used to negotiate NTLM over a connectionless transport. While much of the semantics around the messages remain unchanged, there are a few significant differences:

- SSPI does not create a Type 1 message during the first call to `InitializeSecurityContext`.
- Authentication options are offered by the server, rather than requested by the client.
- Rather than being superfluous (as in connection-oriented authentication), the flags in the Type 3 message carry their usual meanings.

During "normal" (connection-oriented) authentication, all options are negotiated in the first transaction between the client and the server, during the exchange of the Type 1 and Type 2 messages. The negotiated settings are "remembered" by the server and applied to the client's Type 3 message. Although most clients send the agreed-upon flags with the Type 3 message, they are not used in connection authentication.

In datagram authentication, however, the game changes a bit; to alleviate the server's need to track the negotiated options (which becomes more difficult without a persistent connection), the Type 1 message is removed completely. The server generates a Type 2 message containing all supported flags (as well as the challenge, of course). The client then decides which options it will support, and replies with a Type 3 message containing the responses to the challenge and the set of selected flags. The SSPI handshake sequence for datagram authentication is as follows:

1. The client calls `AcquireCredentialsHandle` to obtain a representation of the credential set for the user.
2. The client calls `InitializeSecurityContext`, passing the `ISC_REQ_DATAGRAM` flag as a context requirement via the fContextReq parameter. This starts the construction of the client's security context, but does *not* produce a request token (Type 1 message).

3. The server calls the `AcceptSecurityContext` function, specifying the `ASC_REQ_DATAGRAM` context requirement flag and passing in a null input token. This creates the local security context and yields an authentication response token (the Type 2 message). This Type 2 message will contain the "Negotiate Datagram Style" flag, as well as all flags supported by the server. This is sent to the client as usual.
4. The client receives the Type 2 message and passes it to `InitializeSecurityContext`. The client selects appropriate options from those presented by the server (including "Negotiate Datagram Style", which must be set), creates the responses to the challenge, and populates the Type 3 message. The message is then relayed to the server.
5. The server passes the Type 3 message into the `AcceptSecurityContext` function. The message is processed according to the flags selected by the client, and the context is successfully accepted.

When used with SSPI, there is apparently no means of producing a datagram-style Type 1 message. It is interesting to note, however, that we can "induce" datagram semantics at a lower level by subtly manipulating the NTLMSSP tokens to produce our own datagram Type 1 token.

This can be achieved by setting the "Negotiate Datagram Style" flag on the Type 1 message produced by the first `InitializeSecurityContext` call in a connection-oriented SSPI handshake before passing the token to the server. When the modified Type 1 message is passed into the `AcceptSecurityContext` function, the server will adopt datagram semantics (even though `ASC_REQ_DATAGRAM` was not specified). This will produce a Type 2 message with the "Negotiate Datagram Style" flag set, but otherwise identical to the connection-oriented message that would normally have been generated; that is, the Type 1 flags sent by the client are considered during the construction of the Type 2 message, rather than simply offering all supported options.

The client can then call `InitializeSecurityContext` with this Type 2 token. Note that the client is still in connection-oriented mode; the Type 3 message produced will ignore the "Negotiate Datagram Style" flag applied to the Type 2 message. The server, however, is enforcing datagram semantics, and will now require the Type 3 flags to be set appropriately. Adding the "Negotiate Datagram Style" flag to the Type 3 message manually before sending it to the server allows the server to successfully call `AcceptSecurityContext` with the modified token.

This results in successful authentication; the "doctored" Type 1 message effectively switches the server into datagram-style authentication, in which the Type 3 flags are observed and enforced. There is no known practical use for this, but it does demonstrate some of the interesting and unexpected behavior that can be observed by strategically manipulating the NTLM messages.

Session Security - Signing & Sealing Concepts

In addition to the SSPI authentication services, message integrity and confidentiality functionality is provided. This is also implemented by the NTLM Security Support Provider. "Signing" is performed by the SSPI `MakeSignature` function, which applies a Message Authentication Code (MAC) to a message. This can be verified by the recipient, and provides a strong assurance that the message was not modified in transit. The signature is generated using a secret key, known to the sender and receiver; the MAC can only be verified by a party possessing the key (which in turn provides assurance that the signature was created by the sender). "Sealing" is performed by the SSPI `EncryptMessage` function. This applies encryption to a message to prevent it from being viewed by a third party in transit; the NTLMSSP uses a variety of symmetric encryption mechanisms (the same key is used to decrypt as to encrypt).

The keys used in signing and sealing are established as a by-product of the NTLM authentication process; in addition to verifying a client's identity, the authentication handshake establishes a context between the client and server which includes the key(s) needed to sign and seal messages between the parties. We will discuss the derivation of these keys, and the mechanisms used for signing and sealing by the NTLMSSP.

There are numerous key schemes employed during signing and sealing; we will start with an overview of

the different types of keys and core session security concepts.

The User Session Key

This is the basic key type employed in session security. There are many variants:

- The LM User Session Key
- The NTLM User Session Key
- The LMv2 User Session Key
- The NTLMv2 User Session Key
- The NTLM2 Session Response User Session Key

The method of derivation used depends on the responses sent in the Type 3 message. These variants and their calculations are outlined below.

The LM User Session Key

Used when only the LM response is provided (i.e., with Win9x clients). The LM User Session Key is derived as follows:

1. The 16-byte LM hash (calculated previously) is truncated to 8 bytes.
2. This is null-padded to 16 bytes. This value is the LM User Session Key.

As with the LM hash itself, this key only changes in response to a change of password by the user. Note also that only the first 7 password characters have input to the key (see the process for computing the LM Response; the LM User Session Key is the first half of the LM hash). Additionally, the keyspace is actually much smaller, as the LM hash itself is based on the uppercase password. All of these factors combined render the LM User Session Key very weak against attack.

The NTLM User Session Key

This variant is used when the client sends the NTLM response. The calculation of the key is fairly straightforward:

1. The NTLM hash is obtained (the MD4 digest of the Unicode mixed-case password, calculated previously).
2. The MD4 message-digest algorithm is applied to the NTLM hash, resulting in a 16-byte value. This is the NTLM User Session Key.

The NTLM User Session Key is much improved over the LM User Session Key. The password space is larger (it is case-sensitive, rather than converting the password to uppercase); additionally, all password characters have input in the key generation. However, it is still only changed when the user changes his or her password; this makes offline attacks much easier.

The LMv2 User Session Key

Used when the LMv2 response is sent (but not the NTLMv2 response). Deriving this key is a bit more complicated, but not terribly complex:

1. The NTLMv2 hash is obtained (as calculated previously).
2. The LMv2 client nonce is obtained (used in the LMv2 response).
3. The challenge from the Type 2 message is concatenated with the client nonce. The HMAC-MD5 message authentication code algorithm is applied to this value using the NTLMv2 hash as the key, resulting in a 16-byte output value.

4. The HMAC-MD5 algorithm is applied to this value, again using the NTLMv2 hash as the key. The resulting 16-byte value is the LMv2 User Session Key.

The LMv2 User Session Key offers several improvements over the NTLMv1-based keys. It is derived from the NTLMv2 hash (itself derived from the NTLM hash), which is specific to the username and domain/server; additionally, both the server challenge and client nonce provide input to the key calculation. The key calculation can also be stated simply as the HMAC-MD5 digest of the first 16 bytes of the LMv2 response (using the NTLMv2 hash as the key).

The NTLMv2 User Session Key

Used when the NTLMv2 response is sent. Calculation of this key is very similar to the LMv2 User Session Key:

1. The NTLMv2 hash is obtained (as calculated previously).
2. The NTLMv2 "blob" is obtained (as used in the NTLMv2 response).
3. The challenge from the Type 2 message is concatenated with the blob. The HMAC-MD5 message authentication code algorithm is applied to this value using the NTLMv2 hash as the key, resulting in a 16-byte output value.
4. The HMAC-MD5 algorithm is applied to this value, again using the NTLMv2 hash as the key. The resulting 16-byte value is the NTLMv2 User Session Key.

The NTLMv2 User Session Key is quite similar cryptographically to the LMv2 User Session Key. It can be stated as the HMAC-MD5 digest of the first 16 bytes of the NTLMv2 response (using the NTLMv2 hash as the key).

The NTLM2 Session Response User Session Key

Used when NTLMv1 authentication is employed with NTLM2 session security. This key is derived from the NTLM2 session response information as follows:

1. The NTLM User Session Key is obtained as outlined previously.
2. The session nonce is obtained (discussed previously, this is the concatenation of the Type 2 challenge and the nonce from the NTLM2 session response).
3. The HMAC-MD5 algorithm is applied to the session nonce, using the NTLM User Session Key as the key. The resulting 16-byte value is the NTLM2 Session Response User Session Key.

The NTLM2 Session Response User Session Key is notable in that it is calculated between the client and server, rather than at the domain controller. The domain controller derives the NTLM User Session Key and supplies it to the server, as before; if NTLM2 session security has been negotiated with the client, the server then takes the HMAC-MD5 digest of the session nonce using the NTLM User Session Key as the MAC key.

The Null User Session Key

The Null User Session Key is employed when Anonymous authentication is performed. This one is simple; it's just 16 null bytes ("`0x00000000000000000000000000000000`").

The Lan Manager Session Key

The Lan Manager Session Key is an alternative to the User Session Keys, used to derive keys in NTLM1 signing and sealing when the "Negotiate Lan Manager Key" NTLM flag is set. Calculation of the Lan Manager Session Key is as follows:

1. The 16-byte LM hash (calculated previously) is truncated to 8 bytes.
2. This is padded to 14 bytes with the value "`0xbdbdbdbdbdbd`".
3. This value is split into two 7-byte halves.
4. These values are used to create two DES keys (one from each 7-byte half).
5. Each of these keys is used to DES-encrypt the first 8 bytes of the LM response (resulting in two 8-byte ciphertext values).
6. These two ciphertext values are concatenated to form a 16-byte value - the Lan Manager Session Key.

Note that the Lan Manager Session Key is based on the LM response (rather than simply the LM hash), which means that it will change in response to a different server challenge. This is an advantage over the LM and NTLM User Session Keys, which are based solely on the password hash; the Lan Manager Session Key changes for each authentication operation, while the LM/NTLM User Session Keys remain the same until the user changes his or her password. For this reason, the Lan Manager Session Key is a much stronger scheme than the LM User Session Key (both have a similar key strength, but the Lan Manager Session Key prevents against replay attacks). The NTLM User Session Key has a full 128-bit keyspace, but like the LM User Session Key does not vary on each authentication.

Key Exchange

When the "Negotiate Key Exchange" flag is negotiated, the client and server will agree upon a "secondary" key, used instead of the session key for signing and sealing. This done as follows:

1. The client selects a random 16-byte key (the secondary key).
2. The session key (either the User Session Key or Lan Manager Session Key, depending on the state of the "Negotiate Lan Manager Key" flag) is used to RC4-encrypt the secondary key. This results in a 16-byte ciphertext value.
3. This value is sent to the server in the "Session Key" field of the Type 3 message.
4. The server receives the Type 3 message and decrypts the value sent by the client (using RC4 with the User Session Key or Lan Manager Session Key).
5. The resulting value is the recovered secondary key, and is used in place of the session key for signing and sealing.

Additionally, the key exchange process subtly changes the signing protocol in NTLM2 session security (discussed in a subsequent section).

Key Weakening

The key used for signing and sealing is "weakened" in accordance with cryptographic export restrictions. The key strength is determined by the "Negotiate 128" and "Negotiate 56" flags. The strength of the final key used is the maximum strength supported by both the client and server; if neither flag is set, the default key length of 40 bits is used. NTLM1 signing and sealing supports 40-bit and 56-bit keys; NTLM2 session security supports 40-bit, 56-bit, and unweakened 128-bit keys.

NTLM1 Session Security

NTLM1 is the "original" NTLMSSP signing and sealing scheme, used when the "Negotiate NTLM2 Key" flag is not negotiated. Key derivation in this scheme is driven by the following NTLM flags:

Negotiate Lan Manager Key	When set, the Lan Manager Session Key is used as the basis for the signing and sealing keys (rather than the User Session Key). If not established, the User Session Key will be used for key derivation.

Negotiate 56	Indicates support for 56-bit keys. If not negotiated, 40-bit keys will be used. This is only applicable in combination with "Negotiate Lan Manager Key"; User Session Keys are not weakened under NTLM1 (as they are already weak).
Negotiate Key Exchange	Indicates that key exchange will be performed to negotiate a secondary key for signing and sealing.

NTLM1 Key Derivation

Deriving NTLM1 keys is essentially a three-step process:

1. Master key negotiation
2. Key exchange
3. Key weakening

Master Key Negotiation

The first step is negotiation of the 128-bit "master key" from which the final signing and sealing key will be derived. This is driven by the "Negotiate Lan Manager Key" NTLM flag; if set, the Lan Manager Session Key will be used as the master key. Otherwise, the appropriate User Session Key is employed.

As an example, consider our example user with the password "SecREt01". If the "Negotiate Lan Manager" key is not set, and an NTLM response was provided in the Type 3 message, the NTLM User Session Key will be selected as the master key. This is calculated by taking the MD4 digest of the NTLM hash (which is itself the MD4 hash of the Unicode password):

```
0x3f373ea8e4af954f14faa506f8eebdc4
```

Key Exchange

If the "Negotiate Key Exchange" flag is set, the client will populate the "Session Key" field in the Type 3 message with a new master key, RC4-encrypted with the previously selected master key. The server will decrypt this value to receive the new master key.

For example, assume that the client selects the random master key "`0xf0f0aabb00112233445566778899aabb`". The client will encrypt this value using RC4 with the previously negotiated master key ("`0x3f373ea8e4af954f14faa506f8eebdc4`") to obtain the value:

```
0x1d3355eb71c82850a9a2d65c2952e6f3
```

This is sent to the server in the "Session Key" field of the Type 3 message. The server RC4-decrypts this value using the old master key to recover the new master key selected by the client ("`0xf0f0aabb00112233445566778899aabb`").

Key Weakening

Finally, the key is weakened to comply with export restrictions. NTLM1 supports 40-bit and 56-bit keys. If the "Negotiate 56" NTLM flag is set, the 128-bit master key will be weakened to 56-bits; otherwise, it will be weakened to 40-bits. Note that key weakening is only employed under NTLM1 when the Lan Manager Session Key is used ("Negotiate Lan Manager Key" is set). The LM and NTLM User Session Keys are based on the password hashes, rather than the responses; a given password will always result in the same User Session Key under NTLM1. Weakening was apparently not deemed necessary, since the User Session Key can be easily recovered given a user's password hash.

The process for key weakening under NTLM1 is as follows:

- To produce a 56-bit key, the master key is truncated to 7 bytes (56 bits) and the byte value "`0xa0`" is appended.
- To produce a 40-bit key, the master key is truncated to 5 bytes (40 bits) and the three-byte value "`0xe538b0`" is appended.

Using the master key "`0x0102030405060708090a0b0c0d0e0f00`" as an example, the 40-bit key used for signing and sealing would be "`0x0102030405e538b0`". If 56-bit keys are negotiated, the final key would be "`0x01020304050607a0`".

Signing

Once the key has been negotiated it can be used to produce digital signatures, providing message integrity. Support for signing is indicated by the presence of the "Negotiate Sign" NTLM flag.

NTLM1 signing (as done by the SSPI `MakeSignature` function) is performed as follows:

1. An RC4 cipher is initialized using the previously negotiated key. This is done once (before the first signing operation), and the keystream is never reset.
2. The CRC32 checksum of the message is calculated; this is represented as a long (32-bit little-endian value).
3. A sequence number is obtained; this starts at zero and is incremented after each message is signed. The number is represented as a long.
4. Four zero bytes are concatenated with the CRC32 value and sequence number to obtain a 12-byte value ("`0x00000000`" + CRC32(message) + sequenceNumber).
5. This value is encrypted using the previously initialized RC4 cipher.
6. The first four bytes of the ciphertext result are overwritten with a pseudorandom counter value (the actual value used is insignificant).
7. A version number ("`0x01000000`") is concatenated with the result from the previous step to form the signature.

As an example, assume that we are signing the message "`jCIFS`" (hexadecimal "`0x6a43494653`") using the 40-bit key from the previous example:

1. The CRC32 checksum is calculated (in little-endian hexadecimal, "`0xa0310bb7`").
2. A sequence number is obtained. Since this is the first message we have signed, the sequence number is zero ("`0x00000000`").
3. Four zero bytes are concatenated with the CRC32 value and sequence number to obtain a 12-byte value ("`0x00000000a0310bb700000000`").
4. This value is RC4-encrypted using our key ("`0x0102030405e538b0`"); this yields the ciphertext "`0xecbf1ced397420fe0e5a0f89`".
5. The first four bytes are overwritten with a counter value; using "`0x78010900`" gives "`0x78010900397420fe0e5a0f89`".
6. The version stamp is concatenated with the result to form the final signature:

   `0x0100000078010900397420fe0e5a0f89`

The next message signed would receive the sequence number 1; also, note again that the RC4 keystream initialized with the first signing is not reset for subsequent signatures.

Sealing

In addition to message integrity, message confidentiality is provided via sealing. The "Negotiate Seal" NTLM flag indicates support for sealing. Under SSPI with the NTLM provider, sealing is always

performed in combination with signing (sealing a message simultaneously generates a signature); the same RC4 keystream is used for both signing and sealing.

NTLM1 sealing (as done by the SSPI `EncryptMessage` function) is performed as follows:

1. An RC4 cipher is initialized using the previously negotiated key. This is done once (before the first sealing operation), and the keystream is never reset.
2. The message is encrypted using the RC4 cipher; this produces the sealed ciphertext.
3. A signature for the message is produced, as described previously, and placed in the security trailer buffer.

As an example, consider the sealing of the message "`jCIFS`" ("`0x6a43494653`") using the 40-bit key "`0x0102030405e538b0`":

1. An RC4 cipher is initialized with our key ("`0x0102030405e538b0`").
2. Our message is passed through the RC4 cipher, yielding the ciphertext "`0x86fc55abca`". This is the sealed message.
3. The CRC32 checksum of our message is calculated (in little-endian hexadecimal, "`0xa0310bb7`").
4. A sequence number is obtained. Since this is the first signing, the sequence number is zero ("`0x00000000`").
5. Four zero bytes are concatenated with the CRC32 value and sequence number to obtain a 12-byte value ("`0x00000000a0310bb700000000`").
6. This value is RC4-encrypted using the keystream from our cipher; this yields the ciphertext "`0x452b490efa3e828bcc8affc3`".
7. The first four bytes are overwritten with a counter value; using "`0x78010900`" gives "`0x78010900fa3e828bcc8affc3`".
8. The version stamp is concatenated with the result to form the final signature, which is placed in the security trailer buffer:

   ```
   0x0100000078010900fa3e828bcc8affc3
   ```

   A hexadecimal dump of the entire sealed structure would be:

   ```
   0x86fc55abca0100000078010900fa3e828bcc8affc3
   ```

NTLM2 Session Security

NTLM2 is a newer signing and sealing scheme, used when the "Negotiate NTLM2 Key" flag has been established. Key derivation in this scheme is driven by the following NTLM flags:

Negotiate NTLM2 Key	Indicates support for NTLM2 session security.
Negotiate 56	Indicates support for 56-bit keys. If neither this flag nor "Negotiate 128" are specified, 40-bit keys will be used.
Negotiate 128	Indicates support for 128-bit keys. If neither this flag nor "Negotiate 56" are specified, 40-bit keys will be used.
Negotiate Key Exchange	Indicates that key exchange will be performed to negotiate a secondary base key for signing and sealing.

NTLM2 Key Derivation

Key derivation in NTLM2 is a four-step process:

1. Master key negotiation

2. Key exchange
3. Key weakening
4. Subkey generation

Master Key Negotiation

The User Session Key is always used as the base master key in NTLM2 signing and sealing. When NTLMv2 authentication is used, either the LMv2 or NTLMv2 User Session Key will be employed as the master key. When NTLMv1 authentication is used with NTLM2 session security, the NTLM2 Session Response User Session Key is used as the master key. Note that the User Session Keys used in NTLM2 are significantly stronger than either their NTLM1 counterparts or the Lan Manager Session Key, as they incorporate both the server challenge and client nonce.

Key Exchange

Key exchange is performed as previously discussed for NTLM1. The client selects a secondary master key, RC4-encrypts it with the base master key, and sends the ciphertext value to the server in the Type 3 "Session Key" field. This is indicated by the presence of the "Negotiate Key Exchange" flag.

Key Weakening

Key weakening in NTLM2 is performed simply by truncating the master key (or secondary master key, if key exchange is performed) to the appropriate length; for example, the master key "`0xf0f0aabb00112233445566778899aabb`" would be weakened to 40 bits as "`0xf0f0aabb00`", and 56 bits as "`0xf0f0aabb001122`". Note that 128-bit keys are supported under NTLM2; in this case, the master key is used directly in the generation of subkeys (with no weakening performed).

The master key is only weakened under NTLM2 when generating the sealing subkeys; the full 128-bit master key is always used in the generation of signing keys.

Subkey Generation

Under NTLM2, up to four subkeys are established; the master key is never actually used to sign or seal messages. The subkeys are generated as follows:

1. The 128-bit (unweakened) master key is concatenated with the null-terminated ASCII constant string:

   ```
   session key to client-to-server signing key magic constant
   ```

   In hexadecimal, this constant is:

   ```
   0x73657373696f6e206b657920746f2063
     6c69656e742d746f2d73657276657220
     7369676e696e67206b6579206d616769
     6320636f6e7374616e7400
   ```

   The line breaks in the above are for display purposes only. The MD5 message-digest algorithm is applied to this, resulting in a 16-byte value. This is the Client Signing Key, used by the client to create signatures for messages.

2. The unweakened master key is concatenated with the null-terminated ASCII constant string:

   ```
   session key to server-to-client signing key magic constant
   ```

   In hexadecimal, this constant is:

```
0x73657373696f6e206b657920746f2073
  65727665722d746f2d636c69656e7420
  7369676e696e67206b6579206d616769
  6320636f6e7374616e7400
```

The MD5 digest of this is taken, resulting in the 16-byte Server Signing Key. This is used by the server to create signatures for messages.

3. The weakened (depending on whether 40-bit, 56-bit, or 128-bit encryption is negotiated) master key is concatenated with the null-terminated ASCII constant string:

   ```
   session key to client-to-server sealing key magic constant
   ```

   In hexadecimal, this constant is:

   ```
   0x73657373696f6e206b657920746f2063
     6c69656e742d746f2d73657276657220
     7365616c696e67206b6579206d616769
     6320636f6e7374616e7400
   ```

   The MD5 digest is taken to obtain the 16-byte Client Sealing Key. This is used by the client to encrypt messages.

4. The weakened master key is concatenated with the null-terminated ASCII constant string:

   ```
   session key to server-to-client sealing key magic constant
   ```

   In hexadecimal, this constant is:

   ```
   0x73657373696f6e206b657920746f2073
     65727665722d746f2d636c69656e7420
     7365616c696e67206b6579206d616769
     6320636f6e7374616e7400
   ```

   The MD5 digest algorithm is applied, yielding the 16-byte Server Sealing Key. This key is used by the server to encrypt messages.

Signing

Signing support is again indicated by the "Negotiate Sign" NTLM flag. Signing by the client is done using the Client Signing Key; the server signs messages using the Server Signing Key. Signing keys are generated from the unweakened master key (as discussed previously).

NTLM2 signing (as done by the SSPI `MakeSignature` function) is performed as follows:

1. A sequence number is obtained; this starts at zero and is incremented after each message is signed. The number is represented as a long (32-bit little-endian value).
2. The sequence number is concatenated with the message; the HMAC-MD5 message authentication code algorithm is applied to this value, using the appropriate Signing Key. This yields a 16-byte value.
3. If Key Exchange has been negotiated, an RC4 cipher is initialized using the appropriate *Sealing* key. This is done once (during the first operation), and the keystream is never reset. The first eight bytes from the HMAC result are encrypted using this RC4 cipher. If Key Exchange has *not* been negotiated, this sealing operation is omitted.
4. A version number ("`0x01000000`") is concatenated with the result from the previous step and the sequence number to form the signature.

As an example, assume that we are signing the message "`jCIFS`" (hexadecimal "`0x6a43494653`") on the client, using the master key "`0x0102030405060708090a0b0c0d0e0f00`". This is concatenated with the client-

to-server signing constant, and MD5 is applied to yield the Client Signing Key ("0xf7f97a82ec390f9c903dac4f6aceb132") and Client Sealing Key ("0x2785f595293f3e2813439d73a223810d"); these are used as follows to sign the message:

1. A sequence number is obtained. Since this is the first message we have signed, the sequence number is zero ("0x00000000").
2. The sequence number is concatenated with the message:

   0x000000006a43494653

   HMAC-MD5 is applied using the Client Signing Key ("0xf7f97a82ec390f9c903dac4f6aceb132"). This results in the 16-byte value "0x0a003602317a759a720dc9c7a2a95257".

3. An RC4 cipher is initialized with our sealing key ("0x2785f595293f3e2813439d73a223810d"). The first eight bytes of the previous result are passed through the cipher, yielding the ciphertext "0xe37f97f2544f4d7e".
4. The version stamp is concatenated with the result from the previous step and the sequence number to form the final signature:

   0x01000000e37f97f2544f4d7e00000000

Sealing

The "Negotiate Seal" NTLM flag again indicates support for message confidentiality in NTLM2. NTLM2 sealing (as done by the SSPI `EncryptMessage` function) is performed as follows:

1. An RC4 cipher is initialized using the appropriate sealing key (depending on whether the client or server is performing sealing). This is done once (before the first sealing operation), and the keystream is never reset.
2. The message is encrypted using the RC4 cipher; this produces the sealed ciphertext.
3. A signature for the message is produced, as described previously, and placed in the security trailer buffer. Note that the RC4 cipher used in the signing operation has already been initialized (in the previous steps); it is not reset for the signing operation.

As an example, assume that we are sealing the message "jCIFS" (hexadecimal "0x6a43494653") on the client, using the master key "0x0102030405060708090a0b0c0d0e0f00". As in our previous example, we generate the Client Signing Key ("0xf7f97a82ec390f9c903dac4f6aceb132") using the unweakened master key. We also need to generate the Client Sealing Key; we will assume that 40-bit weakening has been negotiated. We concatenate the weakened master key ("0x0102030405") with the client-to-server sealing constant, and MD5 is applied to yield the Client Sealing Key ("0x6f0d99535033951cbe499cd1914fe9ee"). The following procedure is used to seal the message:

1. An RC4 cipher is initialized with our Client Sealing Key ("0x6f0d99535033951cbe499cd1914fe9ee").
2. Our message is passed through the RC4 cipher, yielding the ciphertext "0xcf0eb0a939". This is the sealed message.
3. A sequence number is obtained. Since this is the first signing, the sequence number is zero ("0x00000000").
4. The sequence number is concatenated with the message:

   0x000000006a43494653

   HMAC-MD5 is applied using the Client Signing Key ("0xf7f97a82ec390f9c903dac4f6aceb132"). This results in the 16-byte value "0x0a003602317a759a720dc9c7a2a95257".

5. The first eight bytes of this value are passed through the sealing cipher, resulting in the ciphertext "0x884b14809e53bfe7".

6. The version stamp is concatenated with the result and the sequence number to form the final signature, which is placed in the security trailer buffer:

   ```
   0x01000000884b14809e53bfe700000000
   ```

   A hexadecimal dump of the entire sealed structure would be:

   ```
   0xcf0eb0a93901000000884b14809e53bfe700000000
   ```

Miscellaneous Session Security Topics

There are a couple of other session security topics which don't really fit anywhere else:

- Datagram Signing & Sealing
- "Dummy" Signing

Datagram Signing & Sealing

This is used when a datagram context has been established (indicated by the datagram authentication handshake and the presence of the "Negotiate Datagram Style" flag). The semantics regarding datagram session security are a bit different; signing can begin on the client immediately after the first call to the SSPI `InitializeSecurityContext` function (i.e., before any communication to the server). This implies a prearranged signing and sealing scheme (as signatures can be created before any options are negotiated with the server). Datagram session security is based on 40-bit Lan Manager Session Key NTLM1 with key exchange (although there may be some means of predetermining a stronger scheme through the registry).

The sequence number is not incremented in datagram mode; it is fixed at zero, and each signature reflects this. Also, the RC4 keystream is reset for each signing or sealing operation. This is significant, in that messages are potentially susceptible to known plaintext attacks.

"Dummy" Signing

This is used if the SSPI context was initialized without specifying support for message integrity. If the "Negotiate Always Sign" NTLM flag is established, calls to `MakeSignature` will succeed, returning the constant "signature":

```
0x0100000000000000000000000000000000
```

Calls to `EncryptMessage` succeed normally (including the "real" signature in the security trailer buffer). If "Negotiate Always Sign" is not negotiated, both signing and sealing will fail.

Appendix A: Links and References

Note that due to the highly dynamic and transient nature of the Web, these may or may not be available.

[The jCIFS Project Home Page]
http://jcifs.samba.org/
jCIFS is an open-source Java implementation of CIFS/SMB. The information presented in this article was used as the basis for the jCIFS NTLM authentication implementation. jCIFS provides support for both the client and server sides of the NTLM HTTP authentication scheme, as well as non-protocol-specific NTLM utility classes.

[The Samba Home Page]

http://www.samba.org/
Samba is an open-source CIFS/SMB server and client. Implements NTLM authentication and session security, and a reference for much of this document.

Implementing CIFS: The Common Internet FileSystem
http://ubiqx.org/cifs/
A highly informative online book by Christopher R. Hertel. Especially relevant to this discussion is the section on authentication.

The Open Group ActiveX Core Technology Reference (Chapter 11, "NTLM")
http://www.opengroup.org/comsource/techref2/NCH1222X.HTM
Closest thing to an "official" reference on NTLM. Unfortunately, also rather old and not terribly accurate.

The Security Support Provider Interface
http://www.microsoft.com/windows2000/techinfo/howitworks/security/sspi2000.asp
A whitepaper discussing application development using the SSPI.

NTLM Authentication Scheme for HTTP
http://www.innovation.ch/java/ntlm.html
Informative discussion on the NTLM HTTP authentication mechanism.

Squid NTLM Authentication Project
http://squid.sourceforge.net/ntlm/
Project to provide NTLM HTTP authentication to the Squid proxy server.

Jakarta Commons HttpClient
http://jakarta.apache.org/commons/httpclient/
An open-source Java HTTP client which provides support for the NTLM HTTP authentication scheme.

The GNU Crypto Project
http://www.gnu.org/software/gnu-crypto/
An open-source Java Cryptography Extension provider supplying an implementation of the MD4 message-digest algorithm.

RFC 1320 - The MD4 Message-Digest Algorithm
http://www.ietf.org/rfc/rfc1320.txt
Specification and reference implementation for the MD4 digest (used to calculate the NTLM password hash).

RFC 1321 - The MD5 Message-Digest Algorithm
http://www.ietf.org/rfc/rfc1321.txt
Specification and reference implementation for the MD5 digest (used to calculate the NTLM2 session response).

RFC 2104 - HMAC: Keyed-Hashing for Message Authentication
http://www.ietf.org/rfc/rfc2104.txt
Specification and reference implementation for the HMAC-MD5 algorithm (used in the calculation of the NTLMv2/LMv2 responses).

How to Enable NTLM 2 Authentication
http://support.microsoft.com/default.aspx?scid=KB;en-us;239869
Describes how to enable negotiation of NTLMv2 authentication and enforce NTLM security flags.

Microsoft SSPI Function Documentation

http://windowssdk.msdn.microsoft.com/en-us/library/ms717571.aspx#sspi_functions
Provides an overview of the Security Support Provider Interface (SSPI) and related functions.

Appendix B: Application Protocol Usage of NTLM

This section examines the use of NTLM authentication within some of Microsoft's network protocol implementations.

NTLM HTTP Authentication

Microsoft has established the proprietary "NTLM" authentication scheme for HTTP to provide integrated authentication to IIS web servers. This authentication mechanism allows clients to access resources using their Windows credentials, and is typically used within corporate environments to provide single sign-on functionality to intranet sites. Historically, NTLM authentication was only supported by Internet Explorer; recently, however, support has been added to various other user agents.

The NTLM HTTP authentication mechanism works as follows:

1. The client requests a protected resource from the server:

   ```
   GET /index.html HTTP/1.1
   ```

2. The server responds with a 401 status, indicating that the client must authenticate. "NTLM" is presented as a supported authentication mechanism via the "`WWW-Authenticate`" header. Typically, the server closes the connection at this time:

   ```
   HTTP/1.1 401 Unauthorized
   WWW-Authenticate: NTLM
   Connection: close
   ```

   Note that Internet Explorer will only select NTLM if it is the first mechanism offered; this is at odds with RFC 2616, which states that the client must select the strongest supported authentication scheme.

3. The client resubmits the request with an "`Authorization`" header containing a Type 1 message parameter. The Type 1 message is Base-64 encoded for transmission. From this point forward, the connection is kept open; closing the connection requires reauthentication of subsequent requests. This implies that the server and client must support persistent connections, via either the HTTP 1.0-style "Keep-Alive" header or HTTP 1.1 (in which persistent connections are employed by default). The relevant request headers appear as follows (the line break in the "`Authorization`" header below is for display purposes only, and is not present in the actual message):

   ```
   GET /index.html HTTP/1.1
   Authorization: NTLM TlRMTVNTUAABAAAABzIAAAYABgArAAAACwALACAAAABXT1
   JLU1RBVElPTkRPTUFJTg==
   ```

4. The server replies with a 401 status containing a Type 2 message in the "`WWW-Authenticate`" header (again, Base-64 encoded). This is shown below (the line breaks in the "`WWW-Authenticate`" header are for editorial clarity only, and are not present in the actual header).

   ```
   HTTP/1.1 401 Unauthorized
   WWW-Authenticate: NTLM TlRMTVNTUAACAAAADAAMADAAAAABAoEAASNFZ4mrze8
   AAAAAAAAAAAGIAYgA8AAAARABPAE0AQQBJAE4AAgAMAEQATwBNAEEASQBOAAEADABTA
   EUAUgBWAEUAUgAEABQAZABvAG0AYQBpAG4ALgBjAG8AbQADACIAcwBlAHIAdgBlAHI
   ALgBkAG8AbQBhAGkAbgAuAGMAbwBtAAAAAAAA=
   ```

5. The client responds to the Type 2 message by resubmitting the request with an "`Authorization`" header containing a Base-64 encoded Type 3 message (again, the line breaks in the "`Authorization`" header below are for display purposes only):

```
GET /index.html HTTP/1.1
Authorization: NTLM TlRMTVNTUAADAAAAGAAYAGoAAAAYABgAggAAAAwADABAAA
AACAAIAEwAAAAWABYAVAAAAAAAAABCaAAAAAQIAAEQATwBNAEEASQBOAHUAcwBlAHIA
VwBPAFIASwBTAFQAQQBUAEkATwBOAMM3zVy9RPyXgqZnr21CfG3mfCDC0+d8ViWpjB
wx6BhHRmspst9GgPOZWPuMITqcxg==
```

6. Finally, the server validates the responses in the client's Type 3 message and allows access to the resource.

```
HTTP/1.1 200 OK
```

This scheme differs from most "normal" HTTP authentication mechanisms, in that subsequent requests over the authenticated connection are not themselves authenticated; NTLM is connection-oriented, rather than request-oriented. So a second request for "`/index.html`" would not carry any authentication information, and the server would request none. If the server detects that the connection to the client has been dropped, a request for "`/index.html`" would result in the server reinitiating the NTLM handshake.

A notable exception to the above is the client's behavior when submitting a POST request (typically employed when the client is sending form data to the server). If the client determines that the server is not the local host, the *client* will initiate reauthentication for POST requests over the active connection. The client will first submit an empty POST request with a Type 1 message in the "`Authorization`" header; the server responds with the Type 2 message (in the "`WWW-Authenticate`" header as shown above). The client then resubmits the POST with the Type 3 message, sending the form data with the request.

The NTLM HTTP mechanism can also be used for HTTP proxy authentication. The process is similar, except:

- The server uses the 407 response code (indicating proxy authentication required) rather than 401.
- The client's Type 1 and 3 messages are sent in the "`Proxy-Authorization`" request header, rather than the "`Authorization`" header.
- The server's Type 2 challenge is sent in the "`Proxy-Authenticate`" response header (instead of "`WWW-Authenticate`").

With Windows 2000, Microsoft introduced the "Negotiate" HTTP authentication mechanism. While primarily aimed at providing a means of authenticating the user against Active Directory via Kerberos, it is backward-compatible with the NTLM scheme. When the Negotiate mechanism is used in "legacy" mode, the headers passed between the client and server are identical, except "Negotiate" (rather than "NTLM") is indicated as the mechanism name.

NTLM POP3 Authentication

Microsoft's Exchange server provides an NTLM authentication mechanism for the POP3 protocol. This is a proprietary extension used with the POP3 `AUTH` command as documented in RFC 1734. On the client side, this mechanism is supported by Outlook and Outlook Express, and is called "Secure Password Authentication".

The POP3 NTLM authentication handshake occurs during the POP3 "authorization" state, and works as follows:

1. The client may request a list of supported authentication mechanisms by sending the `AUTH` command with no arguments:

```
AUTH
```

2. The server responds with a success message, followed by the list of supported mechanisms; this list should include "`NTLM`", and is terminated by a line containing a single period ("`.`").

```
+OK The operation completed successfully.
```

```
NTLM
.
```

3. The client initiates NTLM authentication by sending an `AUTH` command specifying NTLM as the authentication mechanism:

```
AUTH NTLM
```

4. The server responds with a success message as shown below. Note that there is a space between the "+" and the "`OK`"; RFC 1734 states that the server should reply with a challenge, but NTLM requires the Type 1 message from the client. So the server sends a "non-challenge", which is basically the message "`OK`".

```
+ OK
```

5. The client then sends the Type 1 message, Base-64 encoded for transmission:

```
TlRMTVNTUAABAAAABzIAAAYABgArAAAACwALACAAAABXT1JLU1RBVElPTkRPTUFJTg==
```

6. The server replies with the Type 2 challenge message (again, Base-64 encoded). This is send in the challenge format specified by RFC 1734 ("+", followed by a space, followed by the challenge message). This is shown below; the line breaks are for editorial clarity and are not present in the server's reply:

```
+ TlRMTVNTUAACAAAADAAMADAAAAABAoEAASNFZ4mrze8AAAAAAAAAAGIAYgA8AAAA
RABPAE0AQQBJAE4AAgAMAEQATwBNAEEASQBOAAEADABTAEUAUgBWAEUAUgAEABQAZA
BvAG0AYQBpAG4ALgBjAG8AbQADACIAcwBlAHIAdgBlAHIALgBkAG8AbQBhAGkAbgAu
AGMAbwBtAAAAAAA=
```

7. The client calculates and sends the Base-64 encoded Type 3 response (the line breaks below are for display purposes only):

```
TlRMTVNTUAADAAAAGAAYAGoAAAAYABgAggAAAAwADABAAAAACAAIAEwAAAAWABYAVA
AAAAAAAACaAAAAAQIAAEQATwBNAEEASQBOAAHUAcwBlAHIAVwBPAFIASwBTAFQAQQBU
AEkATwBOAMM3zVy9RPyXgqZnr21CfG3mfCDC0+d8ViWpjBwx6BhHRmspst9GgPOZWP
uMITqcxg==
```

8. The server validates the response and indicates the result of authentication:

```
+OK User successfully logged on
```

After successful authentication has occurred, the POP3 session enters the "transaction" state, allowing messages to be retrieved by the client.

NTLM IMAP Authentication

Exchange provides an IMAP authentication mechanism similar in form to the POP3 mechanism previously discussed. IMAP authentication is documented in RFC 1730; the NTLM mechanism is a proprietary extension provided by Exchange and supported by the Outlook client family.

The handshake sequence is similar to the POP3 mechanism:

1. The server may indicate support for the NTLM authentication mechanism in the capability response. Upon connecting to the IMAP server, the client would request the list of server capabilities:

```
0000 CAPABILITY
```

2. The server responds with the list of supported capabilities; the NTLM authentication extension is indicated by the presence of the string "`AUTH=NTLM`" in the server's reply:

```
* CAPABILITY IMAP4 IMAP4rev1 IDLE LITERAL+ AUTH=NTLM
```

```
0000 OK CAPABILITY completed.
```

3. The client initiates NTLM authentication by sending an `AUTHENTICATE` command specifying NTLM as the authentication mechanism:

```
0001 AUTHENTICATE NTLM
```

4. The server responds with an empty challenge, consisting simply of a "+":

```
+
```

5. The client then sends the Type 1 message, Base-64 encoded for transmission:

```
TlRMTVNTUAABAAAABzIAAAYABgArAAAACwALACAAAABXT1JLU1RBVElPTkRPTUFJTg==
```

6. The server replies with the Type 2 challenge message (again, Base-64 encoded). This is send in the challenge format specified by RFC 1730 ("+", followed by a space, followed by the challenge message). This is shown below; the line breaks are for editorial clarity and are not present in the server's reply:

```
+ TlRMTVNTUAACAAAADAAMADAAAAABAoEAASNFZ4mrze8AAAAAAAAAAGIAYgA8AAAA
RABPAE0AQQBJAE4AAgAMAEQATwBNAEEASQBOAAEADABTAEUAUgBWAEUAUgAEABQAZA
BvAG0AYQBpAG4ALgBjAG8AbQADACIAcwBlAHIAdgBlAHIALgBkAG8AbQBhAGkAbgAu
AGMAbwBtAAAAAAA=
```

7. The client calculates and sends the Base-64 encoded Type 3 response (the line breaks below are for display purposes only):

```
TlRMTVNTUAADAAAAGAAYAGoAAAAYABgAggAAAAwADABAAAAACAAIAEwAAAAWABYAVA
AAAAAAAACaAAAAAQIAAEQATwBNAEEASQBOAHUAcwBlAHIAVwBPAFIASwBTAFQAQQBU
AEkATwBOAMM3zVy9RPyXgqZnr21CfG3mfCDC0+d8ViWpjBwx6BhHRmspst9GgPOZWP
uMITqcxg==
```

8. The server validates the response and indicates the result of authentication:

```
0001 OK AUTHENTICATE NTLM completed.
```

After authentication has completed, the IMAP session enters the authenticated state.

NTLM SMTP Authentication

In addition to the NTLM authentication mechanisms provided for POP3 and IMAP, Exchange provides similar functionality for the SMTP protocol. This allows NTLM authentication of users sending outgoing mail messages. This is a proprietary extension used with the SMTP `AUTH` command (documented in RFC 2554).

The SMTP NTLM authentication handshake operates as follows:

1. The server may indicate support for NTLM as an authentication mechanism in the `EHLO` reply. Upon connecting to the SMTP server, the client would send the initial `EHLO` message:

```
EHLO client.example.com
```

2. The server responds with the list of supported extensions; the NTLM authentication extension is indicated by its presence in the list of `AUTH` mechanisms as shown below. Note that the `AUTH` list is sent twice (once with an "=" and once without). The "`AUTH=`" form was apparently specified in a draft of the RFC; sending both forms ensures that clients implemented against this draft are supported.

```
250-server.example.com Hello [10.10.2.20]
250-HELP
250-AUTH LOGIN NTLM
250-AUTH=LOGIN NTLM
```

```
250 SIZE 10240000
```

3. The client initiates NTLM authentication by sending an `AUTH` command specifying NTLM as the authentication mechanism and providing the Base-64 encoded Type 1 message as a parameter:

   ```
   AUTH NTLM TlRMTVNTUAABAAAABzIAAAYABgArAAAACwALACAAAABXT1JLU1RBVElPTkRPTUFJTg==
   ```

   According to RFC 2554, the client may opt not to send the initial response parameter (instead merely sending "`AUTH NTLM`" and waiting for an empty server challenge before replying with the Type 1 message). However, this did not appear to work properly when tested against Exchange.

4. The server replies with a 334 response containing the Type 2 challenge message (again, Base-64 encoded). This is shown below; the line breaks are for editorial clarity and are not present in the server's reply:

   ```
   334 TlRMTVNTUAACAAAADAAMADAAAAABAoEAASNFZ4mrze8AAAAAAAAAAGIAYgA8AAAA
   RABPAE0AQQBJAE4AAgAMAEQATwBNAEEASQBOAAEADABTAEUAUgBWAEUAUgAEABQAZA
   BvAG0AYQBpAG4ALgBjAG8AbQADACIAcwBlAHIAdgBlAHIALgBkAG8AbQBhAGkAbgAu
   AGMAbwBtAAAAAAA=
   ```

5. The client calculates and sends the Base-64 encoded Type 3 response (the line breaks below are for display purposes only):

   ```
   TlRMTVNTUAADAAAAGAAYAGoAAAAYABgAggAAAAwADABAAAAACAAIAEwAAAAWABYAVA
   AAAAAAAACaAAAAAQIAAEQATwBNAEEASQBOAHUAcwBlAHIAVwBPAFIASwBTAFQAQQBU
   AEkATwBOAMM3zVy9RPyXgqZnr21CfG3mfCDC0+d8ViWpjBwx6BhHRmspst9GgPOZWP
   uMITqcxg==
   ```

6. The server validates the response and indicates the result of authentication:

   ```
   235 NTLM authentication successful.
   ```

After authenticating, the client is able to send messages normally.

Appendix C: Sample NTLMSSP Operation Decompositions

This section contains detailed analysis and decomposition of various authentication and signing/sealing operations. This is by no means a comprehensive catalog of possible scenarios, but does include some of the more interesting variants. These were produced by calling the `InitializeSecurityContext` and `AcceptSecurityContext` SSPI functions between two peers, and applying subtle manipulations to the resulting NTLM messages. After the context was established, the `ExportSecurityContext` function was called to dump the contents of the context to a file, for use in subsequent offline analysis. Additionally, the `MakeSignature` and `EncryptMessage` functions were used to perform signing and sealing, respectively.

NTLMv1 Authentication; NTLM1 Signing and Sealing Using the NTLM User Session Key

Demonstration of NTLMv1 authentication with NTLM User Session Key NTLM1 signing and sealing (without key exchange).

```
LMCompatibilityLevel set to 0 (LM/NTLM).

AcquireCredentialsHandle called with domain "TESTNT", user "test", password
"test1234".

------------------------------------------------------------------------------
InitializeSecurityContext called with ISC_REQ_INTEGRITY and
ISC_REQ_CONFIDENTIALITY.  Following flags were masked off of resulting Type 1
message:
```

```
    Negotiate Lan Manager Key      (0x00000080)
    Negotiate NTLM2 Key            (0x00080000)
    Negotiate 128                  (0x20000000)
    Negotiate Key Exchange         (0x40000000)
    Negotiate 56                   (0x80000000)

4e544c4d53535000010000003782000000000000000000000000000000000000

4e544c4d53535000    "NTLMSSP"
01000000            Type 1 message
37820000            Flags
    Negotiate Unicode              (0x00000001)
    Negotiate OEM                  (0x00000002)
    Request Target                 (0x00000004)
    Negotiate Sign                 (0x00000010)
    Negotiate Seal                 (0x00000020)
    Negotiate NTLM                 (0x00000200)
    Negotiate Always Sign          (0x00008000)
0000000000000000    Supplied Domain header (empty, supplied credentials)
0000000000000000    Supplied Workstation header (empty, supplied credentials)

----------------------------------------------------------------------------
AcceptSecurityContext called with ASC_REQ_INTEGRITY and ASC_REQ_CONFIDENTIALITY.
Produces Type 2 message:

4e544c4d53535000020000000c000c0030000000358281 00b019d38bad875c9d
0000000000000000460046003c0000005400450053005400 4e00540002000c00
5400450053005400 4e00540001000c004d0045004d004200450052003001e00
6d0065006d006200650072002e00740065007300740 02e0063006f006d000000
0000

4e544c4d53535000    "NTLMSSP"
02000000            Type 2 message
0c000c0030000000    Target Name header (length 12, offset 48)
35828100            Flags
    Negotiate Unicode              (0x00000001)
    Request Target                 (0x00000004)
    Negotiate Sign                 (0x00000010)
    Negotiate Seal                 (0x00000020)
    Negotiate NTLM                 (0x00000200)
    Negotiate Always Sign          (0x00008000)
    Target Type Domain             (0x00010000)
    Negotiate Target Info          (0x00800000)
b019d38bad875c9d    Challenge
0000000000000000    Context
460046003c000000    Target Information header (length 70, length 60)
5400450053005400 4e005400    Target Name ("TESTNT")
Target Information block:
    02000c00    NetBIOS Domain Name (length 12)
    5400450053005400 4e005400    "TESTNT"
    01000c00    NetBIOS Server Name (length 12)
    4d0045004d00420045005200    "MEMBER"
    03001e00    DNS Server Name (length 30)
    6d0065006d006200650072002e00740065007300740 02e0063006f006d00
        "member.test.com"
    00000000    Target Information Terminator

----------------------------------------------------------------------------
InitializeSecurityContext called.  Produces a Type 3 message:

4e544c4d53535000030000001800180060000000180018007800000000c000c00
4000000080008004c0000000c000c0054000000000000009000000035828000
5400450053005400 4e00540074006500730074004d0045004d00420045005200
1879f60127f8a877022132ec221bcbf3ca016a9f76095606e6285df3287c5d19
4f84df1a94817c7282d09754b6f9e02a
```

```
4e544c4d53535000    "NTLMSSP"
03000000            Type 3 message
1800180060000000    LM/LMv2 Response header (length 24, offset 96)
1800180078000000    NTLM/NTLMv2 Response header (length 24, offset 120)
0c000c0040000000    Domain Name header (length 12, offset 64)
080008004c000000    User Name header (length 8, offset 76)
0c000c0054000000    Workstation Name header (length 12, offset 84)
0000000090000000    Session Key header (empty)
35828000            Flags
    Negotiate Unicode             (0x00000001)
    Request Target                (0x00000004)
    Negotiate Sign                (0x00000010)
    Negotiate Seal                (0x00000020)
    Negotiate NTLM                (0x00000200)
    Negotiate Always Sign         (0x00008000)
    Negotiate Target Info         (0x00800000)
540045005300540045004e005400                            Domain Name ("TESTNT")
7400650073007400                                        User Name ("test")
4d0045004d00420045005200                                Workstation Name ("MEMBER")
1879f60127f8a877022132ec221bcbf3ca016a9f76095606    LM/LMv2 Response
e6285df3287c5d194f84df1a94817c7282d09754b6f9e02a    NTLM/NTLMv2 Response

------------------------------------------------------------------------------
AcceptSecurityContext called to complete the server-side context.

Server Context Export:

40a42e7840a42e7800000000000000000000000000000000
08ee0b00    server context handle dwUpper
35828101    flags
0000000000000000e8290900e8290900682a0900682a0900000000000000000000
d00200000000000000000000000000000000000000000000000200000000000000
ffffffffffffff7f20010000
ae33a32dca8c9821844f740d5b3f4d6c    outbound signing key
ae33a32dca8c9821844f740d5b3f4d6c    inbound verifying key
ae33a32dca8c9821844f740d5b3f4d6c    outbound encrypting key
ae33a32dca8c9821844f740d5b3f4d6c    inbound decrypting key
00000000000000000000000000000000
ae3787b72ff94884680d3e5658c064df9d9e5d8f0655f22a002ebc7e9f944e29
08cf90e0057f622d31a92b7d0ca6f586d5d36e24af70cdbe52ea49d067aa4ffa
e85a5cb1a41e3241a288f8de8a4c8909593cc698b657750af014b26f13778eee
855310d716f61ce5c795a0a3bbac358bb02611c954a54b2791d6e297f1fd8c6d
18fffc190b9c69b80122f7c8fb036146c2b581443960fe239b967b17e33de15b
73e445401f1db3a8c5ad51f48d665e38796b8263b9ca363f763aef71d1ec12d8
a76cb46a333b0fddab7850ba3493dcd4077299d22515eb5f300221e9e79a80d9
1a1b4d2865e6ccc3470eda7c4ace04bfbddbedc1cb207483f343c4a1422c7a92
00000000000000000000000000000000
ae3787b72ff94884680d3e5658c064df9d9e5d8f0655f22a002ebc7e9f944e29
08cf90e0057f622d31a92b7d0ca6f586d5d36e24af70cdbe52ea49d067aa4ffa
e85a5cb1a41e3241a288f8de8a4c8909593cc698b657750af014b26f13778eee
855310d716f61ce5c795a0a3bbac358bb02611c954a54b2791d6e297f1fd8c6d
18fffc190b9c69b80122f7c8fb036146c2b581443960fe239b967b17e33de15b
73e445401f1db3a8c5ad51f48d665e38796b8263b9ca363f763aef71d1ec12d8
a76cb46a333b0fddab7850ba3493dcd4077299d22515eb5f300221e9e79a80d9
1a1b4d2865e6ccc3470eda7c4ace04bfbddbedc1cb207483f343c4a1422c7a92
0000000000000000
540045005300540045004e0054005c007400650073007400    "TESTNT\test"

ntlmHash = md4(password) = 0x3b1b47e42e0463276e3ded6cef349f93
NTLMUserSessionKey = md4(ntlmHash) = 0xae33a32dca8c9821844f740d5b3f4d6c

Key is *not* weakened (NTLM1 only weakens Lan Manager Session Keys).

Called MakeSignature on the server-side context for message
"0x0102030405060708".
```

```
Yields signature:

0100000090010700087de41e039ae5c5

CRC32(0x0102030405060708) = 0xc588ca3f
Sequence number is 0.
0x00000000 + crc32 + seqnum = 0x00000000c588ca3f00000000

RC4(key, 0x00000000c588ca3f00000000) = 0x20961389087de41e039ae5c5

version num + first 4 bytes overwritten with counter value (0x90010700 here):

0100000090010700087de41e039ae5c5 = signature

--------------------------------------------------------------------------------
Called EncryptMessage twice on the server-side context for message
"0x0102030405060708".

Yields encrypted messages:

3ec555aea59eb55001000000a0030700f64393466a9317f7
1caf3c9a114ca2f401000000803070095c1958123ecafce

same RC4 cipher is used from previous signing operation (i.e., not reset):

RC4(0x0102030405060708) = sealed message = 0x3ec555aea59eb550

trailer buffer gets signature; again uses same RC4 cipher
(sequence number is now 1 because of previous signing):

RC4(0x00000000c588ca3f01000000) = 0x8e8adf2bf64393466a9317f7
version num + first 4 bytes overwritten w/counter (0xa0030700):
01000000a0030700f64393466a9317f7 = trailer signature

second message is same:

RC4(0x0102030405060708) = sealed message = 0x1caf3c9a114ca2f4

trailer buffer signature with sequence 2:

RC4(0x00000000c588ca3f02000000) = 0xf23fc1e495c1958123ecafce
version num + first 4 bytes overwritten w/counter (0x88030700)
01000000803070095c1958123ecafce = trailer signature
```

NTLMv1 Authentication; NTLM1 Signing and Sealing Using the LM User Session Key

Demonstration of NTLMv1 authentication with LM User Session Key NTLM1 signing and sealing (without key exchange). NTLM response is manually removed from the Type 3 message to force the server-side context to use the LM User Session Key.

```
LMCompatibilityLevel set to 0 (LM/NTLM).

AcquireCredentialsHandle called with domain "TESTNT", user "test", password
"test1234".

--------------------------------------------------------------------------------
InitializeSecurityContext called with ISC_REQ_INTEGRITY and
ISC_REQ_CONFIDENTIALITY.  Following flags were masked off of resulting Type 1
message:

    Negotiate Lan Manager Key      (0x00000080)
    Negotiate NTLM2 Key            (0x00080000)
    Negotiate 128                  (0x20000000)
```

```
    Negotiate Key Exchange         (0x40000000)
    Negotiate 56                   (0x80000000)

4e544c4d53535000010000003782000000000000000000000000000000000000

4e544c4d53535000    "NTLMSSP"
01000000            Type 1 message
37820000            Flags
    Negotiate Unicode              (0x00000001)
    Negotiate OEM                  (0x00000002)
    Request Target                 (0x00000004)
    Negotiate Sign                 (0x00000010)
    Negotiate Seal                 (0x00000020)
    Negotiate NTLM                 (0x00000200)
    Negotiate Always Sign          (0x00008000)
0000000000000000    Supplied Domain header (empty, supplied credentials)
0000000000000000    Supplied Workstation header (empty, supplied credentials)

--------------------------------------------------------------------------------
AcceptSecurityContext called with ASC_REQ_INTEGRITY and ASC_REQ_CONFIDENTIALITY.
Produces Type 2 message:

4e544c4d53535000020000000c000c0030000000358281006da297169f7aa9c2
0000000000000000460046003c0000005400450053005400e00540002000c00
5400450053005400e00540001000c004d0045004d004200450052000300 1e00
6d0065006d006200650072002e00740065007300 74002e0063006f006d000000
0000

4e544c4d53535000    "NTLMSSP"
02000000            Type 2 message
0c000c0030000000    Target Name header (length 12, offset 48)
35828100            Flags
    Negotiate Unicode              (0x00000001)
    Request Target                 (0x00000004)
    Negotiate Sign                 (0x00000010)
    Negotiate Seal                 (0x00000020)
    Negotiate NTLM                 (0x00000200)
    Negotiate Always Sign          (0x00008000)
    Target Type Domain             (0x00010000)
    Negotiate Target Info          (0x00800000)
6da297169f7aa9c2    Challenge
0000000000000000    Context
460046003c000000    Target Information header (length 70, offset 60)
5400450053005400 4e005400    Target Name ("TESTNT")
Target Information block:
    02000c00    NetBIOS Domain Name (length 12)
    5400450053005400 4e005400    "TESTNT"
    01000c00    NetBIOS Server Name (length 12)
    4d0045004d00420045005200    "MEMBER"
    03001e00    DNS Server Name (length 30)
    6d0065006d006200650072002e00740065007300 74002e0063006f006d00
        "member.test.com"
    00000000    Target Information Terminator

--------------------------------------------------------------------------------
InitializeSecurityContext called.  Produces a Type 3 message; NTLM response
is removed to force usage of the LM User Session Key:

4e544c4d5353500003000000180018004000000000000000000000000c000c00
5800000008000800640000000c000c006c0000000000000000000035828000
2e17884ea16177e2b751d53b5cc756c3cd57cdfd6e3bf8b95400450053005400
4e00540074006500730074004d0045004d00420045005200

4e544c4d53535000    "NTLMSSP"
03000000            Type 3 message
1800180040000000    LM/LMv2 Response (length 24, offset 64)
```

```
0000000000000000    NTLM/NTLMv2 Response (empty, removed)
0c000c0058000000    Domain Name header (length 12, offset 88)
0800080064000000    User Name header (length 8, offset 100)
0c000c006c000000    Workstation Name header (length 12, offset 108)
0000000000000000    Session Key header (empty)
35828000            Flags
    Negotiate Unicode              (0x00000001)
    Request Target                 (0x00000004)
    Negotiate Sign                 (0x00000010)
    Negotiate Seal                 (0x00000020)
    Negotiate NTLM                 (0x00000200)
    Negotiate Always Sign          (0x00008000)
    Negotiate Target Info          (0x00800000)
2e17884ea16177e2b751d53b5cc756c3cd57cdfd6e3bf8b9    LM/LMv2 Response
540045005300540054004e005400                        Domain Name ("TESTNT")
7400650073007400                                    User Name ("test")
4d0045004d00420045005200                            Workstation Name ("MEMBER")

------------------------------------------------------------------------------
AcceptSecurityContext called to complete the server-side context.

Server Context Export:

40a42e7840a42e78000000000000000000000000000000000
08ee0b00    server context handle dwUpper
35828101    flags
000000000000000f8290900f8290900782a0900782a0900000000000000000000
d00200000000000000000000000000000000000000000000000200000000000000
ffffffffffffff7f28010000
624aac413795cdc10000000000000000    outbound signing key
624aac413795cdc10000000000000000    inbound verifying key
624aac413795cdc10000000000000000    outbound encrypting key
624aac413795cdc10000000000000000    inbound decrypting key
0000000000000000000000000
59ad15f0da768c3d17b6c57d41885c21d530a998e4140743f86c0a45067994a3
4a91002db96aae962b0b7abe77feaf1651ccaa249ae754563ab8abe6c9869ddc
1d1052e3328a6f2564b0f7408eeb6b57e9d268dd73198b5bf3c75a995e1fbffa
bcdf851a958fc2ea49a8d839fc5dd755fb44c4a258600523ed7ea1ee38351bca
29b72f37d075b567a5d4123ee20de5a09b63ffd633183bfd660ef5780ceccb28
6e97c8e0e84e48c16d7b114cbaa622b4c62708655f1e20c32cf2bb7fb136f672
accf84027c3101d1902eb2932ab3e13f4271826209343c1c61269cd3f4509253
80cd47f946f1efc00313cedb4fa49e879fbd04a7748969d970830f4b8dde814d
0000000000000000000000000000000000
59ad15f0da768c3d17b6c57d41885c21d530a998e4140743f86c0a45067994a3
4a91002db96aae962b0b7abe77feaf1651ccaa249ae754563ab8abe6c9869ddc
1d1052e3328a6f2564b0f7408eeb6b57e9d268dd73198b5bf3c75a995e1fbffa
bcdf851a958fc2ea49a8d839fc5dd755fb44c4a258600523ed7ea1ee38351bca
29b72f37d075b567a5d4123ee20de5a09b63ffd633183bfd660ef5780ceccb28
6e97c8e0e84e48c16d7b114cbaa622b4c62708655f1e20c32cf2bb7fb136f672
accf84027c3101d1902eb2932ab3e13f4271826209343c1c61269cd3f4509253
80cd47f946f1efc00313cedb4fa49e879fbd04a7748969d970830f4b8dde814d
0000000000000000
540045005300540054004e0054005c007400650073007400    "TESTNT\test"

lmHash = lmEncrypt(password, "KGS!@#$%") = 0x624aac413795cdc1ff17365faf1ffe89
LMUserSessionKey = trunc(lmHash) = 0x624aac413795cdc10000000000000000

Key is *not* weakened (NTLM1 only weakens Lan Manager Session Keys).

Called MakeSignature on the server-side context for message
"0x0102030405060708".

Yields signature:

010000090010700cacc888006466cb5
```

```
CRC32(0x0102030405060708) = 0xc588ca3f
Sequence number is 0.
0x00000000 + crc32 + seqnum = 0x00000000c588ca3f00000000

RC4(key, 0x00000000c588ca3f00000000) = 0xdd0e70b1cacc888006466cb5

version num + first 4 bytes overwritten with counter value (0x90010700 here):

0100000090010700cacc888006466cb5 = signature

--------------------------------------------------------------------------------
Called EncryptMessage twice on the server-side context for message
"0x0102030405060708".

Yields encrypted messages:

48793abbf0145ddb01000000a0030700e286c6021ffc3742
09613b9790f7d40e0100000088030700fb8e614d1cf2284c

same RC4 cipher is used from previous signing operation (i.e., not reset):

RC4(0x0102030405060708) = sealed message = 0x48793abbf0145ddb

trailer buffer gets signature; again uses same RC4 cipher
(sequence number is now 1 because of previous signing):

RC4(0x00000000c588ca3f01000000) = 0x623bc698e286c6021ffc3742
version num + first 4 bytes overwritten w/counter (0xa0030700):
01000000a0030700e286c6021ffc3742 = trailer signature

second message is same:

RC4(0x0102030405060708) = sealed message = 0x09613b9790f7d40e

trailer buffer signature with sequence 2:

RC4(0x00000000c588ca3f02000000) = 0x85433470fb8e614d1cf2284c
version num + first 4 bytes overwritten w/counter (0x88030700)
0100000088030700fb8e614d1cf2284c = trailer signature
```

NTLMv1 Authentication; NTLM1 Signing and Sealing Using the 56-bit Lan Manager Session Key

Demonstration of NTLMv1 authentication with 56-bit Lan Manager Session Key NTLM1 signing and sealing (without key exchange).

```
LMCompatibilityLevel set to 0 (LM/NTLM).

AcquireCredentialsHandle called with domain "TESTNT", user "test", password
"test1234".

--------------------------------------------------------------------------------
InitializeSecurityContext called with ISC_REQ_INTEGRITY and
ISC_REQ_CONFIDENTIALITY.  Following flags were masked off of resulting Type 1
message:

    Negotiate NTLM2 Key            (0x00080000)
    Negotiate 128                  (0x20000000)
    Negotiate Key Exchange         (0x40000000)

4e544c4d535350001000000b782008000000000000000000000000000000000000

4e544c4d53535000    "NTLMSSP"
01000000            Type 1 message
b7820080            Flags
```

```
    Negotiate Unicode             (0x00000001)
    Negotiate OEM                 (0x00000002)
    Request Target                (0x00000004)
    Negotiate Sign                (0x00000010)
    Negotiate Seal                (0x00000020)
    Negotiate Lan Manager Key     (0x00000080)
    Negotiate NTLM                (0x00000200)
    Negotiate Always Sign         (0x00008000)
    Negotiate 56                  (0x80000000)
0000000000000000    Supplied Domain header (empty, supplied credentials)
0000000000000000    Supplied Workstation header (empty, supplied credentials)

------------------------------------------------------------------------------
AcceptSecurityContext called with ASC_REQ_INTEGRITY and ASC_REQ_CONFIDENTIALITY.
Produces Type 2 message:

4e544c4d53535000020000000c000c0030000000b5828180c77c1fcdb77ad042
0000000000000000460046003c0000005400450053005400 4e00540002000c00
5400450053005400 4e00540001000c004d0045004d00420045005200 03001e00
6d0065006d006200650072002e0074006500730074002e0063006f006d000000
0000

4e544c4d53535000    "NTLMSSP"
02000000            Type 2 message
0c000c0030000000    Target Name header (length 12, offset 48)
b5828180            Flags
    Negotiate Unicode             (0x00000001)
    Request Target                (0x00000004)
    Negotiate Sign                (0x00000010)
    Negotiate Seal                (0x00000020)
    Negotiate Lan Manager Key     (0x00000080)
    Negotiate NTLM                (0x00000200)
    Negotiate Always Sign         (0x00008000)
    Target Type Domain            (0x00010000)
    Negotiate Target Info         (0x00800000)
    Negotiate 56                  (0x80000000)
c77c1fcdb77ad042    Challenge
0000000000000000    Context
460046003c000000    Target Information header (length 70, offset 60)
5400450053005400 4e005400    Target Name ("TESTNT")
Target Information block:
    02000c00    NetBIOS Domain Name (length 12)
    5400450053005400 4e005400    "TESTNT"
    01000c00    NetBIOS Server Name (length 12)
    4d0045004d00420045005200    "MEMBER"
    03001e00    DNS Server Name (length 30)
    6d0065006d006200650072002e0074006500730074002e0063006f006d00
        "member.test.com"
    00000000    Target Information Terminator

------------------------------------------------------------------------------
InitializeSecurityContext called.  Produces a Type 3 message:

4e544c4d535350000300000018001800600000001800180078000000 0c000c00
4000000080008004c0000000c000c0054000000000000000090000000b5828080
5400450053005400 4e00540074006500730074004d0045004d00420045005200
2e1580af209c1579bbd95a0c9568e2a7455764064cd8ff8c75791b1820178018
c9d00365a5dedfaa455ef8c3b3ad1c1c

4e544c4d53535000    "NTLMSSP"
03000000            Type 3 message
1800180060000000    LM/LMv2 Response header (length 24, offset 96)
1800180078000000    NTLM/NTLMv2 Response header (length 24, offset 120)
0c000c0040000000    Domain Name header (length 12, offset 64)
080008004c000000    User Name header (length 8, offset 76)
0c000c0054000000    Workstation Name header (length 12, offset 84)
```

```
0000000090000000    Session Key header (empty)
b5828080            Flags
    Negotiate Unicode              (0x00000001)
    Request Target                 (0x00000004)
    Negotiate Sign                 (0x00000010)
    Negotiate Seal                 (0x00000020)
    Negotiate Lan Manager Key      (0x00000080)
    Negotiate NTLM                 (0x00000200)
    Negotiate Always Sign          (0x00008000)
    Negotiate Target Info          (0x00800000)
    Negotiate 56                   (0x80000000)
54004500530054004e005400                            Domain Name ("TESTNT")
7400650073007400                                    User Name ("test")
4d0045004d00420045005200                            Workstation Name ("MEMBER")
2e1580af209c1579bbd95a0c9568e2a7455764064cd8ff8c    LM/LMv2 Response
75791b1820178018c9d00365a5dedfaa455ef8c3b3ad1c1c    NTLM/NTLMv2 Response


------------------------------------------------------------------------------
AcceptSecurityContext called to complete the server-side context.

Server Context Export:

40a42e7840a42e78000000000000000000000000000000
08ee0b00    server context handle dwUpper
b5828181    flags, also has an unknown flag set
000000000000000e8290900e8290900682a0900682a0900000000000000000000
d00200000000000000000000000000000000000000000000000200000000000000
ffffffffffffff7f20010000
f41c7848bec59daa4cfe52156645f77b
f41c7848bec59daa4cfe52156645f77b
f41c7848bec59daa4cfe52156645f77b
f41c7848bec59daa4cfe52156645f77b
00000000000000000000000000
f41115be030662aca4a022504869d30421ae431964da92b42af589ea8a4e5f5e
4c140ab6b342bacb3405ce25e787e5176f30108229c538b5db44e86b1367ad86
573b5294818d6e5174f683c10ecf2ec7a27831dd1efbeeb2fe7302e48b402061
af07bf45d5bcfacc36b81c3e3d6018128e79a54b93998097ab66b798720bbd47
d96a6890d2c2f0ffcdc985a9955ae09fc36c54f17f71d0287d8fdea34df35688
6dc6bb1dc823241a7e7ac4edefa10d7c26d18c77639675d40ce37bebf75cd62c
ecfd84769a3fb15ba82d2f53d74f32fca7e6aa0f279bb02b41084ae1f8dff2e9
e2399c003aa6b95d5833ca1b37dc65593c914916099e7001551f9dd8c04635f9
00000000000000000000000000000000
f41115be030662aca4a022504869d30421ae431964da92b42af589ea8a4e5f5e
4c140ab6b342bacb3405ce25e787e5176f30108229c538b5db44e86b1367ad86
573b5294818d6e5174f683c10ecf2ec7a27831dd1efbeeb2fe7302e48b402061
af07bf45d5bcfacc36b81c3e3d6018128e79a54b93998097ab66b798720bbd47
d96a6890d2c2f0ffcdc985a9955ae09fc36c54f17f71d0287d8fdea34df35688
6dc6bb1dc823241a7e7ac4edefa10d7c26d18c77639675d40ce37bebf75cd62c
ecfd84769a3fb15ba82d2f53d74f32fca7e6aa0f279bb02b41084ae1f8dff2e9
e2399c003aa6b95d5833ca1b37dc65593c914916099e7001551f9dd8c04635f9
0000000000000000
54004500530054004e0054005c007400650073007400    "TESTNT\test"

lmHash = lmEncrypt(password, "KGS!@#$%") = 0x624aac413795cdc1ff17365faf1ffe89
trunc(lmHash + pad) = 0x624aac413795cdc1bdbdbdbdbdbd
lmResponse[0-7] = 0x2e1580af209c1579
LanManagerSessionKey =
    lmEncrypt(0x624aac413795cdc1bdbdbdbdbdbd, 0x2e1580af209c1579) =
        0xf41c7848bec59daa4cfe52156645f77b

Key is weakened to 56-bit w/0xa0 = 0xf41c7848bec59da0

Called MakeSignature on the server-side context for message
"0x0102030405060708".
```

```
Yields signature:

0100000090010700598d18d8150514cc

CRC32(0x0102030405060708) = 0xc588ca3f
Sequence number is 0.
0x00000000 + crc32 + seqnum = 0x00000000c588ca3f00000000

RC4(key, 0x00000000c588ca3f00000000) = 0x2cfc55af598d18d8150514cc

version num + first 4 bytes overwritten with counter value (0x90010700 here):

0100000090010700598d18d8150514cc = signature

--------------------------------------------------------------------------------
Called EncryptMessage twice on the server-side context for message
"0x0102030405060708".

Yields encrypted messages:

357f77b267a494c101000000a0030700fb2ce7d1bfd23a0a
4db804533e6ffc2301000000880307003736d2b43c149c48

same RC4 cipher is used from previous signing operation (i.e., not reset):

RC4(0x0102030405060708) = sealed message = 0x357f77b267a494c1

trailer buffer gets signature; again uses same RC4 cipher
(sequence number is now 1 because of previous signing):

RC4(0x00000000c588ca3f01000000) = 0x757b9976fb2ce7d1bfd23a0a
version num + first 4 bytes overwritten w/counter (0xa0030700):
01000000a0030700fb2ce7d1bfd23a0a = trailer signature

second message is same:

RC4(0x0102030405060708) = sealed message = 0x4db804533e6ffc23

trailer buffer signature with sequence 2:

RC4(0x00000000c588ca3f02000000) = 0x5eb9ab2f3736d2b43c149c48
version num + first 4 bytes overwritten w/counter (0x88030700)
0100000088030700 3736d2b43c149c48 = trailer signature
```

NTLMv1 Authentication; NTLM1 Signing and Sealing Using the 40-bit Lan Manager Session Key

Demonstration of NTLMv1 authentication with 40-bit Lan Manager Session Key NTLM1 signing and sealing (without key exchange).

```
LMCompatibilityLevel set to 0 (LM/NTLM).

AcquireCredentialsHandle called with domain "TESTNT", user "test", password
"test1234".

--------------------------------------------------------------------------------
InitializeSecurityContext called with ISC_REQ_INTEGRITY and
ISC_REQ_CONFIDENTIALITY.  Following flags were masked off of resulting Type 1
message:

    Negotiate NTLM2 Key            (0x00080000)
    Negotiate 128                  (0x20000000)
    Negotiate Key Exchange         (0x40000000)
    Negotiate 56                   (0x80000000)
```

```
4e544c4d53535000010000000b7820000000000000000000000000000000000000000000

4e544c4d53535000    "NTLMSSP"
01000000            Type 1 message
b7820000            Flags
    Negotiate Unicode              (0x00000001)
    Negotiate OEM                  (0x00000002)
    Request Target                 (0x00000004)
    Negotiate Sign                 (0x00000010)
    Negotiate Seal                 (0x00000020)
    Negotiate Lan Manager Key      (0x00000080)
    Negotiate NTLM                 (0x00000200)
    Negotiate Always Sign          (0x00008000)
0000000000000000    Supplied Domain header (empty, supplied credentials)
0000000000000000    Supplied Workstation header (empty, supplied credentials)

--------------------------------------------------------------------------------
AcceptSecurityContext called with ASC_REQ_INTEGRITY and ASC_REQ_CONFIDENTIALITY.
Produces Type 2 message:

4e544c4d53535000020000000c000c0030000000b58281007116b94341ee4e70
00000000000000004600460003c0000005400450053005400 4e00540002000c00
5400450053005400 4e00540001000c004d0045004d00420045005200 3003001e00
6d0065006d006200650072002e00740065007300740 02e0063006f006d000000
0000

4e544c4d53535000    "NTLMSSP"
02000000            Type 2 message
0c000c0030000000    Target Name header (length 12, offset 48)
b5828100            Flags
    Negotiate Unicode              (0x00000001)
    Request Target                 (0x00000004)
    Negotiate Sign                 (0x00000010)
    Negotiate Seal                 (0x00000020)
    Negotiate Lan Manager Key      (0x00000080)
    Negotiate NTLM                 (0x00000200)
    Negotiate Always Sign          (0x00008000)
    Target Type Domain             (0x00010000)
    Negotiate Target Info          (0x00800000)
7116b94341ee4e70    Challenge
0000000000000000    Context
460046003c000000    Target Information header (length 70, offset 60)
5400450053005400 4e005400    Target Name ("TESTNT")
Target Information block:
    02000c00    NetBIOS Domain Name (length 12)
    5400450053005400 4e005400    "TESTNT"
    01000c00    NetBIOS Server Name (length 12)
    4d0045004d00420045005200    "MEMBER"
    03001e00    DNS Server Name (length 30)
    6d0065006d006200650072002e00740065007300740 02e0063006f006d00
        "member.test.com"
    00000000    Target Information Terminator

--------------------------------------------------------------------------------
InitializeSecurityContext called.  Produces a Type 3 message:

4e544c4d5353500003000000180018006000000018001800780000000c000c00
4000000080008004c0000000c000c0054000000000000009000000 0b5828000
5400450053005400 4e005400740065007300740 04d0045004d00420045005200
66271e46d60b246d25fcc3340235841057c2821f490d073304c6e94c5624abad
6c922d8e64b6c86d43138f8f0d94fc3f

4e544c4d53535000    "NTLMSSP"
03000000            Type 3 message
1800180060000000    LM/LMv2 Response header (length 24, offset 96)
1800180078000000    NTLM/NTLMv2 Response header (length 24, offset 120)
```

```
0c000c0040000000    Domain Name header (length 12, offset 64)
080008004c000000    User Name header (length 8, offset 76)
0c000c0054000000    Workstation Name header (length 12, offset 84)
0000000090000000    Session Key header (empty)
b5828000            Flags
    Negotiate Unicode              (0x00000001)
    Request Target                 (0x00000004)
    Negotiate Sign                 (0x00000010)
    Negotiate Seal                 (0x00000020)
    Negotiate Lan Manager Key      (0x00000080)
    Negotiate NTLM                 (0x00000200)
    Negotiate Always Sign          (0x00008000)
    Negotiate Target Info          (0x00800000)
54004500530054004e005400                            Domain Name ("TESTNT")
7400650073007400                                    User Name ("test")
4d0045004d00420045005200                            Workstation Name ("MEMBER")
66271e46d60b246d25fcc3340235841057c2821f490d0733    LM/LMv2 Response
04c6e94c5624abad6c922d8e64b6c86d43138f8f0d94fc3f    NTLM/NTLMv2 Response

-------------------------------------------------------------------------------
AcceptSecurityContext called to complete the server-side context.

Server Context Export:

40a42e7840a42e78000000000000000000000000000000000
08ee0b00    server context handle dwUpper
b5828101    flags
0000000000000000182e0900182e0900982e0900982e0900000000000000000000
d0020000000000000000000000000000000000000000000000020000000000000
ffffffffffffff7f20010000
b98a3a22c81e31f99e7eca1e123c04d1    outbound signing key
b98a3a22c81e31f99e7eca1e123c04d1    inbound verifying key
b98a3a22c81e31f99e7eca1e123c04d1    outbound encrypting key
b98a3a22c81e31f99e7eca1e123c04d1    inbound decrypting key
00000000000000000000000
b94480a5718a4f50e1a4e815e91021e0a9a387263fbbd69d6e3b03a2027c0e56
1c47582cab375deacb92462f4db09bb32965670de5dc573e7bf20a901fb68ecd
6d697800a1ced9de7a9a79091b5318b104c827bf7791ad3448c606c5e4bca6eb
7da0949e28ec137596fd5cbaeedb08af52898845a75a95e78d63c36bfe054af0
703df1432aef32f3b2d8594e9961b5251d33f9ae8b2b8fc46a2ef520140fc1d1
d3f785428493a8ff3af8cad216da734c556fdfb4384bb8acf4687672367f1a23
835fc20bcfc05bd0fadd31edfb98b78c823cc9cc629faa1924e3d4300c8681bd
7411e6e235be49396601419cd7541251f6605ed51e6cfcc7407e97176407222d
00000000000000000000000000000000
b94480a5718a4f50e1a4e815e91021e0a9a387263fbbd69d6e3b03a2027c0e56
1c47582cab375deacb92462f4db09bb32965670de5dc573e7bf20a901fb68ecd
6d697800a1ced9de7a9a79091b5318b104c827bf7791ad3448c606c5e4bca6eb
7da0949e28ec137596fd5cbaeedb08af52898845a75a95e78d63c36bfe054af0
703df1432aef32f3b2d8594e9961b5251d33f9ae8b2b8fc46a2ef520140fc1d1
d3f785428493a8ff3af8cad216da734c556fdfb4384bb8acf4687672367f1a23
835fc20bcfc05bd0fadd31edfb98b78c823cc9cc629faa1924e3d4300c8681bd
7411e6e235be49396601419cd7541251f6605ed51e6cfcc7407e97176407222d
0000000000000000
54004500530054004e0054005c007400650073007400    "TESTNT\test"

lmHash = lmEncrypt(password, "KGS!@#$%") = 0x624aac413795cdc1ff17365faf1ffe89
trunc(lmHash + pad) = 0x624aac413795cdc1bdbdbdbdbdbd
lmResponse[0-7] = 0x66271e46d60b246d
LanManagerSessionKey =
    lmEncrypt(0x624aac413795cdc1bdbdbdbdbdbd, 0x66271e46d60b246d) =
        0xb98a3a22c81e31f99e7eca1e123c04d1

Key is weakened to 40-bit w/0xe538b0 = 0xb98a3a22c8e538b0

Called MakeSignature on the server-side context for message
"0x0102030405060708".
```

```
Yields signature:

01000000ffffff001a7599e9ad0ad460

CRC32(0x0102030405060708) = 0xc588ca3f
Sequence number is 0.
0x00000000 + crc32 + seqnum = 0x00000000c588ca3f00000000

RC4(key, 0x00000000c588ca3f00000000) = 0xbeb185ce1a7599e9ad0ad460

version num + first 4 bytes overwritten with counter value (0xffffff00 here):

01000000ffffff001a7599e9ad0ad460 = signature

--------------------------------------------------------------------------------
Called EncryptMessage twice on the server-side context for message
"0x0102030405060708".

Yields encrypted messages:

075c81a31875489401000000060030700 33df86be9d65813d
da731ecef152bd7501000000048030700a61d753437944ee5

same RC4 cipher is used from previous signing operation (i.e., not reset):

RC4(0x0102030405060708) = sealed message = 0x075c81a318754894

trailer buffer gets signature; again uses same RC4 cipher
(sequence number is now 1 because of previous signing):

RC4(0x00000000c588ca3f01000000) = 0x0004c5b033df86be9d65813d
version num + first 4 bytes overwritten w/counter (0x60030700):
010000006003070033df86be9d65813d = trailer signature

second message is same:

RC4(0x0102030405060708) = sealed message = 0xda731ecef152bd75

trailer buffer signature with sequence 2:

RC4(0x00000000c588ca3f02000000) = 0xf012a98ca61d753437944ee5
version num + first 4 bytes overwritten w/counter (0x48030700)
0100000048030700a61d753437944ee5 = trailer signature
```

NTLMv1 Datagram-Style Authentication; NTLM1 Signing and Sealing Using the 40-bit Lan Manager Session Key With Key Exchange Negotiated

Demonstration of datagram-style authentication with default options (defaults to 40-bit Lan Manager Session Key NTLM1 with key exchange when used with NTLMv1).

```
LMCompatibilityLevel set to 0 (LM/NTLM).

AcquireCredentialsHandle called with domain "TESTNT", user "test", password
"test1234".

--------------------------------------------------------------------------------
InitializeSecurityContext called with:
    ISC_REQ_INTEGRITY | ISC_REQ_CONFIDENTIALITY | ISC_REG_DATAGRAM

Produces no token (datagram-style).  Client context can be used for signing
and sealing immediately.

--------------------------------------------------------------------------------
AcceptSecurityContext called with:
```

```
    ASC_REQ_INTEGRITY | ASC_REQ_CONFIDENTIALITY | ASC_REG_DATAGRAM

Produces a Type 2 message:

4e544c4d535350000200000000000000300000000f38298e0ada5839570b5cb99
00000000000000000000000030000000

4e544c4d53535000    "NTLMSSP"
02000000            Type 2 message
0000000030000000    Target Name header (empty)
f38298e0            Flags
    Negotiate Unicode              (0x00000001)
    Negotiate OEM                  (0x00000002)
    Negotiate Sign                 (0x00000010)
    Negotiate Seal                 (0x00000020)
    Negotiate Datagram Style       (0x00000040)
    Negotiate Lan Manager Key      (0x00000080)
    Negotiate NTLM                 (0x00000200)
    Negotiate Always Sign          (0x00008000)
    Negotiate NTLM2 Key            (0x00080000)
    Request Init Response          (0x00100000)
    Negotiate Target Info          (0x00800000)
    Negotiate 128                  (0x20000000)
    Negotiate Key Exchange         (0x40000000)
    Negotiate 56                   (0x80000000)
ada5839570b5cb99    Challenge
0000000000000000    Context
0000000030000000    Target Information header (empty)

------------------------------------------------------------------------------
InitializeSecurityContext called.  Produces a Type 3 message:

4e544c4d53535000030000001800180060000000180018007800000000c000c00
40000000800080040000000c000c0054000000100010009000000f5828040
5400450053005400 4e00540074006500730074004d0045004d00420045005200
e8cff653006525da77c6bef2fed79bc6d7d839f598ead91a4e37300050075eeb
aa5915480c3620b8ee6fa869cdf16e7c9227ebee8b19a312664fa4ed44bd3377

4e544c4d53535000    "NTLMSSP"
03000000            Type 3 message
1800180060000000    LM/LMv2 Response header (length 24, offset 96)
1800180078000000    NTLM/NTLMv2 Response header (length 24, offset 120)
0c000c0040000000    Domain Name header (length 12, offset 64)
080008004c000000    User Name header (length 8, offset 76)
0c000c0054000000    Workstation Name header (length 12, offset 84)
1000100090000000    Session Key header (length 16, offset 144)
f5828040            Flags
    Negotiate Unicode              (0x00000001)
    Request Target                 (0x00000004)
    Negotiate Sign                 (0x00000010)
    Negotiate Seal                 (0x00000020)
    Negotiate Datagram Style       (0x00000040)
    Negotiate Lan Manager Key      (0x00000080)
    Negotiate NTLM                 (0x00000200)
    Negotiate Always Sign          (0x00008000)
    Negotiate Target Info          (0x00800000)
    Negotiate Key Exchange         (0x40000000)
5400450053005400 4e00540074006500 Domain Name ("TESTNT")
7300740074006500 User Name ("test")
4d0045004d00420045005200          Workstation Name ("MEMBER")
e8cff653006525da77c6bef2fed79bc6d7d839f598ead91a    LM/LMv2 Response
4e37300050075eebaa5915480c3620b8ee6fa869cdf16e7c    NTLM/NTLMv2 Response
9227ebee8b19a312664fa4ed44bd3377                    Session Key

------------------------------------------------------------------------------
AcceptSecurityContext called to complete the server-side context.
```

```
Server Context Export:

40a42e7840a42e78000000000000000000000000000000
08ee0b00    server context handle dwUpper
f5828041    flags
00000000000000080740a0080740a0000750a0000750a00000000000000000
d00200000000000000000000000000000000000000000000200000000000000
ffffffffffffff7f20010000
d56070a4c355c2d91693d8f3406d4d82    outbound signing key
d56070a4c355c2d91693d8f3406d4d82    inbound verifying key
d56070a4c355c2d91693d8f3406d4d82    outbound encrypting key
d56070a4c355c2d91693d8f3406d4d82    inbound decrypting key
0000000000000000000000
05f9b27f16db3ea4563b296433256b1076cdd372908ac60367f37d5e495c11ab
378f70d165ea3d633ca5f819e2f45a3906328b8993718c7726d78145042099a7
b0dde1c831986f661ef6bc9dc7de558d80ed447e4cdf749b02b346c4227915cc
09cb75c9430fa3ba53c018fa78b7daff9f62b9efd2a92a60845fe5e0278692dc
e983be951f24b6a2ec592194bd352c0cb596fd7ae747fba0a1c5340861b85dc1
91f1aad9fc14e801858251d687a61dbbb157303fd0bf73e4ce526a9c0d0ec250
d454cf2e38ad4dac6c42f0b41223079ef21b7ba83a0a7c5b2debca404f1300e6
d58e582b692f68eee31a88c3ae4a17fe4b484e419a1c280b6e36afd8f56d97f7
00000000000000000000000000000000
05f9b27f16db3ea4563b296433256b1076cdd372908ac60367f37d5e495c11ab
378f70d165ea3d633ca5f819e2f45a3906328b8993718c7726d78145042099a7
b0dde1c831986f661ef6bc9dc7de558d80ed447e4cdf749b02b346c4227915cc
09cb75c9430fa3ba53c018fa78b7daff9f62b9efd2a92a60845fe5e0278692dc
e983be951f24b6a2ec592194bd352c0cb596fd7ae747fba0a1c5340861b85dc1
91f1aad9fc14e801858251d687a61dbbb157303fd0bf73e4ce526a9c0d0ec250
d454cf2e38ad4dac6c42f0b41223079ef21b7ba83a0a7c5b2debca404f1300e6
d58e582b692f68eee31a88c3ae4a17fe4b484e419a1c280b6e36afd8f56d97f7
0000000000000000
5400450053005400 4e0054005c007400650073007400    "TESTNT\test"

lmHash = lmEncrypt(password, "KGS!@#$%") = 0x624aac413795cdc1ff17365faf1ffe89
trunc(lmHash + pad) = 0x624aac413795cdc1bdbdbdbdbdbd
lmResponse[0-7] = 0xe8cff653006525da
LanManagerSessionKey =
    lmEncrypt(0x624aac413795cdc1bdbdbdbdbdbd, 0xe8cff653006525da) =
        0x97dba8c110cd6b7976c02c203c6be07a

Key exchange performed:
type3Key (from Type 3) = 0x9227ebee8b19a312664fa4ed44bd3377
key = RC4(LanManagerSessionKey, type3Key) =
    0xd56070a4c355c2d91693d8f3406d4d82

Key is weakened to 40-bit w/0xe538b0 = 0xd56070a4c3e538b0

Called MakeSignature on the server-side context for message
"0x0102030405060708".

Yields signature:

010000009801070012c00705ba25a7ec

CRC32(0x0102030405060708) = 0xc588ca3f
Sequence number is 0.
0x00000000 + crc32 + seqnum = 0x00000000c588ca3f00000000

RC4(key, 0x00000000c588ca3f00000000) = 0x39ec604d12c00705ba25a7ec

version num + first 4 bytes overwritten with counter value (0x98010700 here):

010000009801070012c00705ba25a7ec = signature

--------------------------------------------------------------------------------
```

```
Called EncryptMessage twice on the server-side context for message
"0x0102030405060708".

Yields encrypted messages:

38ee6349d24eca32010000008803070012c00705ba25a7ec
38ee6349d24eca32010000007003070012c00705ba25a7ec

RC4 cipher is reset after each signing/sealing operation in datagram-style:

RC4(key, 0x0102030405060708) = sealed message = 0x38ee6349d24eca32

trailer buffer gets signature; RC4 cipher is reset and sequence number is
*not* incremented in datagram-style:

RC4(key, 0x00000000c588ca3f00000000) = 0x39ec604d12c00705ba25a7ec
version num + first 4 bytes overwritten w/counter (0x88030700):
010000008803070012c00705ba25a7ec = trailer signature

second message is same:

RC4(key, 0x00000000c588ca3f00000000) = 0x39ec604d12c00705ba25a7ec
version num + first 4 bytes overwritten w/counter (0x70030700):
010000007003070012c00705ba25a7ec = trailer signature
```

NTLMv1 Authentication; NTLM1 "Dummy" Signing and Sealing Using the NTLM User Session Key

Demonstration of "dummy" signing and sealing. Context is *not* initialized with support for integrity or confidentiality. "Negotiate Always Sign" is negotiated, which allows the context to be used in dummy signing and sealing.

```
LMCompatibilityLevel set to 0 (LM/NTLM).

AcquireCredentialsHandle called with domain "TESTNT", user "test", password
"test1234".

------------------------------------------------------------------------------
InitializeSecurityContext called with default options.  Produces Type 1 message:

4e544c4d53535000010000007820 0a0000000000000000000000000000000000

4e544c4d53535000    "NTLMSSP"
01000000            Type 1 message
078200a0            Flags
    Negotiate Unicode             (0x00000001)
    Negotiate OEM                 (0x00000002)
    Request Target                (0x00000004)
    Negotiate NTLM                (0x00000200)
    Negotiate Always Sign         (0x00008000)
    Negotiate 128                 (0x20000000)
    Negotiate 56                  (0x80000000)
0000000000000000    Supplied Domain header (empty, supplied credentials)
0000000000000000    Supplied Workstation header (empty, supplied credentials)

------------------------------------------------------------------------------
AcceptSecurityContext called with default options. Produces Type 2 message:

4e544c4d53535000020000000c000c003000000058281a0eacf7d5a2a6fa7d4
0000000000000000460046003c0000005400450053005400 4e00540002000c00
5400450053005400 4e00540001000c004d0045004d004200 4500520003001e00
6d0065006d006200650072002e00740065007300740 02e0063006f006d000000
0000
```

```
4e544c4d53535000    "NTLMSSP"
02000000            Type 2 message
0c000c0030000000    Target Name header (length 12, offset 48)
058281a0            Flags
    Negotiate Unicode              (0x00000001)
    Request Target                 (0x00000004)
    Negotiate NTLM                 (0x00000200)
    Negotiate Always Sign          (0x00008000)
    Target Type Domain             (0x00010000)
    Negotiate Target Info          (0x00800000)
    Negotiate 128                  (0x20000000)
    Negotiate 56                   (0x80000000)
eacf7d5a2a6fa7d4    Challenge
0000000000000000    Context
460046003c000000    Target Information header (length 70, offset 60)
540045005300540045004e005400    Target Name ("TESTNT")
Target Information block:
    02000c00    NetBIOS Domain Name (length 12)
    540045005300540045004e005400    "TESTNT"
    01000c00    NetBIOS Server Name (length 12)
    4d0045004d00420045005200    "MEMBER"
    03001e00    DNS Server Name (length 30)
    6d0065006d006200650072002e0074006500730074002e0063006f006d00
        "member.test.com"
    00000000    Target Information Terminator

-------------------------------------------------------------------------------
InitializeSecurityContext called.  Produces a Type 3 message:

4e544c4d53535000    "NTLMSSP"
03000000            Type 3 message
1800180060000000    LM/LMv2 Response header (length 24, offset 96)
1800180078000000    NTLM/NTLMv2 Response header (length 24, offset 120)
0c000c0040000000    Domain Name header (length 12, offset 64)
080008004c000000    User Name header (length 8, offset 76)
0c000c0054000000    Workstation Name header (length 12, offset 84)
0000000090000000    Session Key header (empty)
058280a0            Flags
    Negotiate Unicode              (0x00000001)
    Request Target                 (0x00000004)
    Negotiate NTLM                 (0x00000200)
    Negotiate Always Sign          (0x00008000)
    Negotiate Target Info          (0x00800000)
    Negotiate 128                  (0x20000000)
    Negotiate 56                   (0x80000000)
540045005300540045004e005400                            Domain Name ("TESTNT")
7400650073007400                                        User Name ("test")
4d0045004d00420045005200                                Workstation Name ("MEMBER")
6c454794b50321a067fdf78e92ee5085a5b0a23057e9125b    LM/LMv2 Response
d2025bc5d6c201af7472550a677ca9904245a16ebb542a8e    NTLM/NTLMv2 Response

-------------------------------------------------------------------------------
AcceptSecurityContext called to complete the server-side context.

Server Context Export:

40a42e7840a42e7800000000000000000000000000000000
08ee0b00    server context handle dwUpper
058281a1    flags
0000000000000028f5080028f50800a8f50800a8f5080000000000000000000
d00200000000000000000000000000000000000000000000000200000000000000
ffffffffffffff7f20010000
ae33a32dca8c9821844f740d5b3f4d6c    outbound signing key
ae33a32dca8c9821844f740d5b3f4d6c    inbound verifying key
ae33a32dca8c9821844f740d5b3f4d6c    outbound encrypting key
ae33a32dca8c9821844f740d5b3f4d6c    inbound decrypting key
```

```
00000000000000000000000
ae3787b72ff94884680d3e5658c064df9d9e5d8f0655f22a002ebc7e9f944e29
08cf90e0057f622d31a92b7d0ca6f586d5d36e24af70cdbe52ea49d067aa4ffa
e85a5cb1a41e3241a288f8de8a4c8909593cc698b657750af014b26f13778eee
855310d716f61ce5c795a0a3bbac358bb02611c954a54b2791d6e297f1fd8c6d
18fffc190b9c69b80122f7c8fb036146c2b581443960fe239b967b17e33de15b
73e445401f1db3a8c5ad51f48d665e38796b8263b9ca363f763aef71d1ec12d8
a76cb46a333b0fddab7850ba3493dcd4077299d22515eb5f300221e9e79a80d9
1a1b4d2865e6ccc3470eda7c4ace04bfbddbedc1cb207483f343c4a1422c7a92
0000000000000000000000000000000000
ae3787b72ff94884680d3e5658c064df9d9e5d8f0655f22a002ebc7e9f944e29
08cf90e0057f622d31a92b7d0ca6f586d5d36e24af70cdbe52ea49d067aa4ffa
e85a5cb1a41e3241a288f8de8a4c8909593cc698b657750af014b26f13778eee
855310d716f61ce5c795a0a3bbac358bb02611c954a54b2791d6e297f1fd8c6d
18fffc190b9c69b80122f7c8fb036146c2b581443960fe239b967b17e33de15b
73e445401f1db3a8c5ad51f48d665e38796b8263b9ca363f763aef71d1ec12d8
a76cb46a333b0fddab7850ba3493dcd4077299d22515eb5f300221e9e79a80d9
1a1b4d2865e6ccc3470eda7c4ace04bfbddbedc1cb207483f343c4a1422c7a92
0000000000000000
5400450053005400 4e0054005c007400 650073007400    "TESTNT\test"

ntlmHash = md4(password) = 0x3b1b47e42e0463276e3ded6cef349f93
NTLMUserSessionKey = md4(ntlmHash) = 0xae33a32dca8c9821844f740d5b3f4d6c

Key is *not* weakened (NTLM1 only weakens Lan Manager Session Keys).

Called MakeSignature on the server-side context for message
"0x0102030405060708".

Yields signature:

01000000000000000000000000000000 ("dummy" signature)

--------------------------------------------------------------------------------
Called EncryptMessage twice on the server-side context for message
"0x0102030405060708".

Yields encrypted messages:

2194108dc8f3292901000000 48020700fa4f9c95a098b258
8f88dc2f36cd5e7101000000 30020700d825f5a1154aa5fc

RC4(key, 0x0102030405060708) = sealed message = 0x2194108dc8f32929

trailer buffer gets signature; uses same RC4 cipher as sealing
(sequence number is 0, previous dummy signing doesn't count):

RC4(0x00000000c588ca3f00000000) = 0x039ae5c5fa4f9c95a098b258
version num + first 4 bytes overwritten w/counter (0x48020700):
0100000048020700fa4f9c95a098b258 = trailer signature

second message is same:

RC4(0x0102030405060708) = sealed message = 0x8f88dc2f36cd5e71

trailer buffer signature with sequence 1 (previous trailer signature
incremented it):

RC4(0x00000000c588ca3f01000000) = 0x6b9317f7d825f5a1154aa5fc
version num + first 4 bytes overwritten w/counter (0x30020700)
0100000030020700d825f5a1154aa5fc = trailer signature
```

NTLM2 Session Response Authentication; NTLM2 Signing and Sealing Using the 128-bit NTLM2 Session Response User Session Key With Key Exchange Negotiated

Demonstration of NTLMv1 authentication with 128-bit NTLM2 Session Response User Session Key NTLM2 signing and sealing, with key exchange enabled.

```
LMCompatibilityLevel set to 0 (LM/NTLM).

AcquireCredentialsHandle called with domain "TESTNT", user "test", password
"test1234".

------------------------------------------------------------------------------
InitializeSecurityContext called with ISC_REQ_INTEGRITY and
ISC_REQ_CONFIDENTIALITY.

4e544c4d5353500001000000b78208e000000000000000000000000000000000

4e544c4d53535000    "NTLMSSP"
01000000            Type 1 message
b78208e0            Flags
    Negotiate Unicode              (0x00000001)
    Negotiate OEM                  (0x00000002)
    Request Target                 (0x00000004)
    Negotiate Sign                 (0x00000010)
    Negotiate Seal                 (0x00000020)
    Negotiate Lan Manager Key      (0x00000080)
    Negotiate NTLM                 (0x00000200)
    Negotiate Always Sign          (0x00008000)
    Negotiate NTLM2 Key            (0x00080000)
    Negotiate 128                  (0x20000000)
    Negotiate Key Exchange         (0x40000000)
    Negotiate 56                   (0x80000000)
0000000000000000    Supplied Domain header (empty, supplied credentials)
0000000000000000    Supplied Workstation header (empty, supplied credentials)

------------------------------------------------------------------------------
AcceptSecurityContext called with ASC_REQ_INTEGRITY and ASC_REQ_CONFIDENTIALITY.
Produces Type 2 message:

4e544c4d5353500002000000c000c003000000358289e0677f1c557a5ee96c
0000000000000000460046003c0000005400450053005400 4e00540002000c00
5400450053005400 4e00540001000c004d0045004d004200450052 0003001e00
6d0065006d006200650072002e00740065007300740 02e0063006f006d000000
0000

4e544c4d53535000    "NTLMSSP"
02000000            Type 2 message
0c000c0030000000    Target Name header (length 12, offset 48)
358289e0            Flags
    Negotiate Unicode              (0x00000001)
    Request Target                 (0x00000004)
    Negotiate Sign                 (0x00000010)
    Negotiate Seal                 (0x00000020)
    Negotiate NTLM                 (0x00000200)
    Negotiate Always Sign          (0x00008000)
    Target Type Domain             (0x00010000)
    Negotiate NTLM2 Key            (0x00080000)
    Negotiate Target Info          (0x00800000)
    Negotiate 128                  (0x20000000)
    Negotiate Key Exchange         (0x40000000)
    Negotiate 56                   (0x80000000)
677f1c557a5ee96c    Challenge
0000000000000000    Context
460046003c000000    Target Information header (length 70, offset 60)
5400450053005400 4e005400    Target Name ("TESTNT")
Target Information block:
    02000c00    NetBIOS Domain Name (length 12)
    5400450053005400 4e005400    "TESTNT"
    01000c00    NetBIOS Server Name (length 12)
```

```
    4d0045004d00420045005200    "MEMBER"
    03001e00    DNS Server Name (length 30)
    6d0065006d006200650072002e0074006500730074002e0063006f006d00
        "member.test.com"
    00000000    Target Information Terminator

--------------------------------------------------------------------------------
InitializeSecurityContext called.  Produces a Type 3 message:

4e544c4d535350000300000018001800600000001800180078000000c000c00
4000000080008004c0000000c000c0054000000100010009000000358288e0
5400450053005400 4e00540074006500730074004d0045004d00420045005200
404d1b6f691525800000000000000000000000000000ea8cc49f24da157f
13436637f77693d8b992d619e584c7ee727a5240822ec7af4e9100c43e6fee7f

4e544c4d53535000    "NTLMSSP"
03000000            Type 3 message
1800180060000000    LM/LMv2 Response header (length 24, offset 96)
1800180078000000    NTLM/NTLMv2 Response header (length 24, offset 120)
0c000c0040000000    Domain Name header (length 12, offset 64)
080008004c000000    User Name header (length 8, offset 76)
0c000c0054000000    Workstation Name header (length 12, offset 84)
1000100090000000    Session Key header (length 16, offset 144)
358288e0            Flags
    Negotiate Unicode             (0x00000001)
    Request Target                (0x00000004)
    Negotiate Sign                (0x00000010)
    Negotiate Seal                (0x00000020)
    Negotiate NTLM                (0x00000200)
    Negotiate Always Sign         (0x00008000)
    Negotiate NTLM2 Key           (0x00080000)
    Negotiate Target Info         (0x00800000)
    Negotiate 128                 (0x20000000)
    Negotiate Key Exchange        (0x40000000)
    Negotiate 56                  (0x80000000)
5400450053005400 4e005400                           Domain Name ("TESTNT")
7400650073007400                                    User Name ("test")
4d0045004d00420045005200                            Workstation Name ("MEMBER")
404d1b6f691525800000000000000000000000000000    LM/LMv2 Response
ea8cc49f24da157f13436637f77693d8b992d619e584c7ee    NTLM/NTLMv2 Response
727a5240822ec7af4e9100c43e6fee7f                    Session Key

--------------------------------------------------------------------------------
AcceptSecurityContext called to complete the server-side context.

Server Context Export:

48a42e7848a42e7800000000000000000000000000000000
70f10b00    server context handle dwUpper
358289e1    flags
0000000000000000382e09003c2e0900b82e0900c82f0900000000000000000000
d00200000000000000000000000000000000000000000000000200000000000000
ffffffffffffff7f20010000
6c713b60e6571035c9396ece1e456395    outbound signing key
e775c02a63d159ec64185f6d7d993344    inbound verifying key
e9b0f8e2cbf7b453b8389e8d2d7bb4ba    outbound encrypting key
cc0fc51f360b7da837cde6cb417fd735    inbound decrypting key
00000000000000000000000000
e932727927bbfe589b599a99d212bf23479fa99e7d8953bd6764862e690a6fac
b5f935ba02df8a75e415ddf7d30fc9a89778a2b7d4f0cc0c032b8757b4312663
8c0b39136536b8af1c3a774fc890cf30fbd16b18059c5eec7b3b3782efe06628
216aeda7e83fcb1446344bdeeb1ef5914ad894333d96ea51fdfc8311ad175b52
254ed93ce58bff50d0c78561aeb36ef8952c205acef3cad506085d5c700080ab
c6d77c22b1c22fb9c32a73a0481d100e298e929dc0e1a671ee1f8d4438a5c476
f22d847ec1196ce6094d5f3edc4201b28f6d43aaf4cdf1e362d6fa98a19307b0
55f640e7544c457424bc04c5dadb41160d7f88beb656a41be21a607a688149a3
```

```
00000000000000000000000000000000
cc094487604571968040c0ac1fe2da492938751932bc4139040def8c27635807
ad4e68b5ab6c6a65897f8f2df24f02164cc35466d0111d81f026a23d6d7221fd
34c9cb885e52b10ec88e532ec57a8ba5a6eb3f769446b62ab8a3840548220cd8
ff3b9bb2b731e1e64a50ae08917dc28d869215cf5a677920a801a493c143ec97
1efb576bd717b0303ea9be9fdb9a5dd55b1b8a77aaf42f9ccea183ea6eb36428
f7984bf9d39e9524a7eedd00cae5425533d9bded0b51ded2e43769c7c41a7bc6
af595ff1e33612e8614ddfe9f523785673a085fcd1d67ed4356f5ccd9ddc9003
0a4799bb2b70e71418823cfa06e0f62cbffe133a7cb4baf8101cf325620fb974
0000000000000000
540045005300540045004e0054005c007400650073007400    "TESTNT\test"

ntlmHash = md4(password) = 0x3b1b47e42e0463276e3ded6cef349f93
NTLMUserSessionKey = md4(ntlmHash) = 0xae33a32dca8c9821844f740d5b3f4d6c
challenge (from Type 2) = 0x677f1c557a5ee96c
nonce (from Type 3) = 0x404d1b6f69152580
NTLM2SessionResponseUserSessionKey =
    HMAC(NTLMUserSessionKey, challenge + nonce) =
        0x0d4b30a8750b73ab2dab39e889455fcd

Key exchange performed:
type3Key (from Type 3) = 0x727a5240822ec7af4e9100c43e6fee7f
key = RC4(NTLM2SessionResponseUserSessionKey, type3Key) =
    0x5764dc0a93b1292fa898c29524c30a54

We are using server context:

serverSigningConstant =
    "session key to server-to-client signing key magic constant" =
        0x73657373696f6e206b657920746f207365727665722d746f2d636c69656e74
        207369676e696e67206b6579206d6167696320636f6e7374616e7400

serverSealingConstant =
    "session key to server-to-client sealing key magic constant" =
        0x73657373696f6e206b657920746f207365727665722d746f2d636c69656e74
        207365616c696e67206b6579206d6167696320636f6e7374616e7400

signingKey = MD5(key + serverSigningConstant) =
    0x6c713b60e6571035c9396ece1e456395

User Session Key is not weakened (128-bit negotiated).

sealingKey = MD5(key + serverSealingConstant) =
    0xe9b0f8e2cbf7b453b8389e8d2d7bb4ba

Called MakeSignature on the server-side context for message
"0x0102030405060708".

Yields signature:

0100000069de1aff9cbee43100000000

Sequence number is 0.
seqnum + message = 0x0000000001020304050607 08

HMAC(signingKey, 0x0000000001020304050607 08) =
    0x9d642651faec52f2164297dcaaa1ff6e

sig = RC4(sealingKey, first 8 bytes) = 0x69de1aff9cbee431
version num + sig + seqnum:

0100000069de1aff9cbee43100000000 = signature

--------------------------------------------------------------------------------
Called EncryptMessage twice on the server-side context for message
"0x0102030405060708".
```

```
Yields encrypted messages:

5b4cbbd3b2d8e8a401000000272c6dee5b236fe201000000
29535954c1e00fb9010000002922b8fcada4cda202000000

Uses RC4 cipher from previous signing operation (key exchange encrypts
signature with sealing key as well).

RC4(0x0102030405060708) = sealed message = 0x5b4cbbd3b2d8e8a4

trailer buffer gets signature (sequence number is now 1 because of previous
signing, same RC4 cipher used for this signature too):

HMAC(signingKey, 0x01000000010203040506070 8) =
    0x1b7fc502e4cc73b6d85360d7ee052800
sig = RC4(first 8 bytes) = 0x272c6dee5b236fe2
version num + sig + seqnum:
01000000272c6dee5b236fe201000000 = trailer signature

second message is same, using RC4 cipher from previous operations:

RC4(0x0102030405060708) = sealed message = 0x29535954c1e00fb9

trailer buffer signature with sequence 2:

HMAC(signingKey, 0x02000000010203040506070 8) =
   0x3e46a45716671a6e0d056636f77db4bb
sig = RC4(first 8 bytes) = 0x2922b8fcada4cda2
version num + sig + seqnum:
010000002922b8fcada4cda202000000 = trailer signature
```

NTLM2 Session Response Authentication; NTLM2 Signing and Sealing Using the 40-bit NTLM2 Session Response User Session Key

Demonstration of NTLMv1 authentication with 40-bit NTLM2 Session Response User Session Key NTLM2 signing and sealing (without key exchange).

```
LMCompatibilityLevel set to 0 (LM/NTLM).

AcquireCredentialsHandle called with domain "TESTNT", user "test", password
"test1234".

------------------------------------------------------------------------------
InitializeSecurityContext called with ISC_REQ_INTEGRITY and
ISC_REQ_CONFIDENTIALITY.  Following flags were masked off of resulting Type 1
message:

    Negotiate 128                 (0x20000000)
    Negotiate Key Exchange        (0x40000000)
    Negotiate 56                  (0x80000000)

4e544c4d53535000010000 00b7820800000000000000000000000000000000

4e544c4d53535000    "NTLMSSP"
01000000            Type 1 message
b7820800            Flags
    Negotiate Unicode             (0x00000001)
    Negotiate OEM                 (0x00000002)
    Request Target                (0x00000004)
    Negotiate Sign                (0x00000010)
    Negotiate Seal                (0x00000020)
    Negotiate Lan Manager Key     (0x00000080)
    Negotiate NTLM                (0x00000200)
    Negotiate Always Sign         (0x00008000)
```

```
    Negotiate NTLM2 Key            (0x00080000)
0000000000000000    Supplied Domain (empty, supplied credentials)
0000000000000000    Supplied Workstation (empty, supplied credentials)

----------------------------------------------------------------------------
AcceptSecurityContext called with ASC_REQ_INTEGRITY and ASC_REQ_CONFIDENTIALITY.
Produces Type 2 message:

4e544c4d5353500002000000c000c0030000000358289009190 13ccde5c4d16
0000000000000000460046003c0000005400450053005400 4e00540002000c00
5400450053005400 4e00540001000c004d0045004d004200 4500520003001e00
6d0065006d006200650072002e0074006500730074002e0063006f006d000000
0000

4e544c4d53535000    "NTLMSSP"
02000000            Type 2 message
0c000c0030000000    Target Name header (length 12, offset 48)
35828900            Flags
    Negotiate Unicode              (0x00000001)
    Request Target                 (0x00000004)
    Negotiate Sign                 (0x00000010)
    Negotiate Seal                 (0x00000020)
    Negotiate NTLM                 (0x00000200)
    Negotiate Always Sign          (0x00008000)
    Target Type Domain             (0x00010000)
    Negotiate NTLM2 Key            (0x00080000)
    Negotiate Target Info          (0x00800000)
919013ccde5c4d16    Challenge
0000000000000000    Context
460046003c000000    Target Information header (length 70, offset 60)
5400450053005400 4e005400    Target Name ("TESTNT")
Target Information block:
    02000c00    NetBIOS Domain Name (length 12)
    5400450053005400 4e005400    "TESTNT"
    01000c00    NetBIOS Server Name (length 12)
    4d0045004d00420045005200    "MEMBER"
    03001e00    DNS Server Name (length 30)
    6d0065006d006200650072002e0074006500730074002e0063006f006d00
        "member.test.com"
    00000000    Target Information Terminator

----------------------------------------------------------------------------
InitializeSecurityContext called.  Produces a Type 3 message:

4e544c4d535350000300000018001800600000001800180078000000 0c000c00
4000000080008004c0000000c000c0054000000000000009000000035828800
5400450053005400 4e00540074006500730074004d0045004d00420045005200
02a668799b43b026000000000000000000000000000000191c91d68a26a933
82ec89178c1e496d8f2f63a1c7dc0b54

4e544c4d53535000    "NTLMSSP"
03000000            Type 3 message
1800180060000000    LM/LMv2 Response header (length 24, offset 96)
1800180078000000    NTLM/NTLMv2 Response header (length 24, offset 120)
0c000c0040000000    Domain Name header (length 12, offset 64)
080008004c000000    User Name header (length 8, offset 76)
0c000c0054000000    Workstation Name header (length 12, offset 84)
0000000090000000    Session Key header (empty)
35828800            Flags
    Negotiate Unicode              (0x00000001)
    Request Target                 (0x00000004)
    Negotiate Sign                 (0x00000010)
    Negotiate Seal                 (0x00000020)
    Negotiate NTLM                 (0x00000200)
    Negotiate Always Sign          (0x00008000)
    Negotiate NTLM2 Key            (0x00080000)
```

```
    Negotiate Target Info          (0x00800000)
54004500530054004e005400                              Domain Name ("TESTNT")
7400650073007400                                      User Name ("test")
4d0045004d00420045005200                              Workstation Name ("MEMBER")
02a668799b43b026000000000000000000000000000000    LM/LMv2 Response
191c91d68a26a93382ec89178c1e496d8f2f63a1c7dc0b54    NTLM/NTLMv2 Response

---------------------------------------------------------------------------
AcceptSecurityContext called to complete the server-side context.

Server Context Export:

40a42e7840a42e78000000000000000000000000000000
08ee0b00    server context handle dwUpper
35828901    flags
000000000000000f8290900fc290900782a0900882b0900000000000000000
d00200000000000000000000000000000000000000000000200000000000000
ffffffffffffff7f20010000
605b738984f36aea7d2ccc5678670f2c    outbound signing key
94d75dd6591eb8569d8480b5c9c25136    inbound verifying key
e4c55ca209611e9e007009731b7103d5    outbound encrypting key
738e75e9b0df0ac9139839abf5cc8354    inbound decrypting key
00000000000000000000000000
37aa90adbac5442beb5c77f553b782d01859687c2e421e524e2fbc4ad5ff0555
43657a283af2366b3f3cafa80372a3a781b123dd6f2c22e45e0c57d23bd78399
f64935c94bbf9112156a67a2dc843e1a7069f8ec30f99388fa61c4cf4c3d3824
b046df9c31fc0a0150d6638ca17fb651c79db25d07efbea5dea02580fe406211
ea9b1341b41f76e366857d5afbe92adb6ee2940bfd6c4fe12995b5acc05fbb73
a98bbd33b906ae14a4da00cb0dcc1b75f0edc2792d9e60cef1c192741d8a1039
8d7b479a97b30e20f4217ef39fe6c30948199604c8f79856d1ee4dd9e589648e
26ca86c67802e78f5b34e8a617d3ab5408582745cdd416711c32e06d87b80fd8
00000000000000000000000000000000
eb024b64187446dcf798db69a299356807ba78fec2b642e6cd77a097a80bb9a5
38e7548a935d923dbd22e1b7d80127fc1cc9fad61e404eac033f6bab10e57923
878f729e84500ce97139d5cb8c11c7b21741e2325f56a34aad757a804f34ed19
a747bf05af9b912f94ffea700dd43a8806d73eb4458bd97e1a375c6d9633f3c6
49092dc8635a9a589ce35221c5dda1f926ef1f6a5bc03bb1ce65e089149d2baa
28b3cab59f448655306c25133629bc812c57be592a1b0e1204537bcf00d176b0
3c0faecce8f6c4dfd0674cfbd27d95fdf1da318e8d8582bbec7f16f0c162eea6
0a8324f26660151d2eb851736edea9e4f86f08a45e4dc3d37c48f54390f46120
0000000000000000
54004500530054004e0054005c007400650073007400    "TESTNT\test"

ntlmHash = md4(password) = 0x3b1b47e42e0463276e3ded6cef349f93
NTLMUserSessionKey = md4(ntlmHash) = 0xae33a32dca8c9821844f740d5b3f4d6c
challenge (from Type 2) = 0x919013ccde5c4d16
nonce (from Type 3) = 0x02a668799b43b026
NTLM2SessionResponseUserSessionKey =
    HMAC(NTLMUserSessionKey, challenge + nonce) =
        0x6b60097a8f9dbbff2d23f5b15377ca28

We are using server context:

serverSigningConstant =
    "session key to server-to-client signing key magic constant" =
        0x73657373696f6e206b657920746f207365727665722d746f2d636c69656e74
        207369676e696e67206b6579206d6167696320636f6e7374616e7400

serverSealingConstant =
    "session key to server-to-client sealing key magic constant" =
        0x73657373696f6e206b657920746f207365727665722d746f2d636c69656e74
        207365616c696e67206b6579206d6167696320636f6e7374616e7400

signingKey = MD5(key + serverSigningConstant) =
    0x605b738984f36aea7d2ccc5678670f2c
```

```
User Session Key is weakened to 40-bit for sealing by truncating =
    0x6b60097a8f

sealingKey = MD5(weakenedKey + serverSealingConstant) =
    0xe4c55ca209611e9e007009731b7103d5

Called MakeSignature on the server-side context for message
"0x0102030405060708".

Yields signature:

01000000d1e2d811145d81ec00000000

Sequence number is 0.
seqnum + message = 0x0000000001020304050607080

HMAC(signingKey, 0x0000000001020304050607080) =
    0xd1e2d811145d81ecf5f8723312c75e3c

version num + first 8 bytes + seqnum:

01000000d1e2d811145d81ec00000000 = signature

--------------------------------------------------------------------------------
Called EncryptMessage twice on the server-side context for message
"0x0102030405060708".

Yields encrypted messages:

ab8d38bb0cad7dd601000000eed64de8afb80c8001000000
b011cc07a7f6127b01000000644a8509d73ac48c02000000

RC4(sealingKey, 0x0102030405060708) = sealed message = 0xab8d38bb0cad7dd6

trailer buffer gets signature (sequence number is now 1 because of previous
signing):

HMAC(signingKey, 0x0100000001020304050607080) =
    0xeed64de8afb80c801ea5ddd1d98cbdc8
version num + first 8 bytes + seqnum:
01000000eed64de8afb80c8001000000 = trailer signature

second message is same, using RC4 cipher from previous sealing operation:

RC4(0x0102030405060708) = sealed message = 0xb011cc07a7f6127b

trailer buffer signature with sequence 2:

HMAC(signingKey, 0x0200000001020304050607080) =
    0x644a8509d73ac48c8f5a7f7f65bc8381
version num + first 8 bytes + seqnum:
01000000644a8509d73ac48c02000000 = trailer signature
```

NTLMv2 Authentication; NTLM1 Signing and Sealing Using the 40-bit NTLMv2 User Session Key

Demonstration of NTLMv2 authentication with NTLMv2 User Session Key NTLM1 signing and sealing (without key exchange).

```
LMCompatibilityLevel set to 3 (LMv2/NTLMv2).

AcquireCredentialsHandle called with domain "TESTNT", user "test", password
"test1234".

--------------------------------------------------------------------------------
```

```
InitializeSecurityContext called with ISC_REQ_INTEGRITY and
ISC_REQ_CONFIDENTIALITY.  Following flags were masked off of resulting Type 1
message:

    Negotiate Lan Manager Key      (0x00000080)
    Negotiate NTLM2 Key            (0x00080000)
    Negotiate 128                  (0x20000000)
    Negotiate Key Exchange         (0x40000000)
    Negotiate 56                   (0x80000000)

4e544c4d53535000010000003782000000000000000000000000000000000000

4e544c4d53535000    "NTLMSSP"
01000000            Type 1 message
37820000            Flags
    Negotiate Unicode              (0x00000001)
    Negotiate OEM                  (0x00000002)
    Request Target                 (0x00000004)
    Negotiate Sign                 (0x00000010)
    Negotiate Seal                 (0x00000020)
    Negotiate NTLM                 (0x00000200)
    Negotiate Always Sign          (0x00008000)
0000000000000000    Supplied Domain header (empty, supplied credentials)
0000000000000000    Supplied Workstation header (empty, supplied credentials)

--------------------------------------------------------------------------------
AcceptSecurityContext called with ASC_REQ_INTEGRITY and ASC_REQ_CONFIDENTIALITY.
Produces Type 2 message:

4e544c4d53535000020000000c000c0030000000358281000033b02d17275b77
0000000000000000460046003c0000005400450053005400 4e00540002000c00
5400450053005400 4e00540001000c004d0045004d00420045005200 3003001e00
6d0065006d006200650072002e0074006500730074002e0063006f006d000000
0000

4e544c4d53535000    "NTLMSSP"
02000000            Type 2 message
0c000c0030000000    Target Name header (length 12, offset 48)
35828100            Flags
    Negotiate Unicode              (0x00000001)
    Request Target                 (0x00000004)
    Negotiate Sign                 (0x00000010)
    Negotiate Seal                 (0x00000020)
    Negotiate NTLM                 (0x00000200)
    Negotiate Always Sign          (0x00008000)
    Target Type Domain             (0x00010000)
    Negotiate Target Info          (0x00800000)
0033b02d17275b77    Challenge
0000000000000000    Context
460046003c000000    Target Information header (length 70, offset 60)
5400450053005400 4e005400    Target Name ("TESTNT")
Target Information block:
    02000c00    NetBIOS Domain Name (length 12)
    5400450053005400 4e005400    "TESTNT"
    01000c00    NetBIOS Server Name (length 12)
    4d0045004d00420045005200    "MEMBER"
    03001e00    DNS Server Name (length 30)
    6d0065006d006200650072002e0074006500730074002e0063006f006d00
        "member.test.com"
    00000000    Target Information Terminator

--------------------------------------------------------------------------------
InitializeSecurityContext called.  Produces a Type 3 message:

4e544c4d53535000030000001800180060000000760076007800000000c000c00
4000000080008004c0000000c000c005400000000000000ee00000035828000
```

```
5400450053005400 4e00540074006500730074004d0045004d00420045005200
5d55a02b60a40526ac9a1e4d15fa45a0f2e6329726c598e8f77c67dad00b9321
6242b197fe6addfa0101000000000000502db638677bc301f2e6329726c598e8
0000000002000c0054004500530054004e00540001000c004d0045004d004200
4500520003001e006d0065006d006200650072002e00740065007300740002e00
63006f006d00000000000000000

4e544c4d53535000    "NTLMSSP"
03000000            Type 3 message
1800180060000000    LM/LMv2 Response header (length 24, offset 96)
7600760078000000    NTLM/NTLMv2 Response header (length 118, offset 120)
0c000c0040000000    Domain Name header (length 12, offset 64)
080008004c000000    User Name header (length 8, offset 76)
0c000c0054000000    Workstation Name header (length 12, offset 84)
00000000ee000000    Session Key header (empty)
35828000            Flags
    Negotiate Unicode             (0x00000001)
    Request Target                (0x00000004)
    Negotiate Sign                (0x00000010)
    Negotiate Seal                (0x00000020)
    Negotiate NTLM                (0x00000200)
    Negotiate Always Sign         (0x00008000)
    Negotiate Target Info         (0x00800000)
5400450053005400 4e005400                           Domain Name ("TESTNT")
7400650073007400                                    User Name ("test")
4d0045004d00420045005200                            Workstation Name ("MEMBER")
5d55a02b60a40526ac9a1e4d15fa45a0f2e6329726c598e8    LM/LMv2 Response
NTLM/NTLMv2 Response:
f77c67dad00b93216242b197fe6addfa0101000000000000502db638677bc301
f2e6329726c598e80000000002000c0054004500530054004e00540001000c00
4d0045004d0042004500520003001e006d0065006d006200650072002e007400
6500730074002e0063006f006d00000000000000000

--------------------------------------------------------------------------------
AcceptSecurityContext called to complete the server-side context.

Server Context Export:

40a42e7840a42e78000000000000000000000000000000000
08ee0b00    server context handle dwUpper
35828101    flags, also has an unknown flag set
0000000000000000782e0900782e0900f82e0900f82e09000000000000000000000
d00200000000000000000000000000000000000000000000000200000000000000
ffffffffffffff7f20010000
1c4c7aaa7403acf01b1fa565bc950810    outbound signing key
1c4c7aaa7403acf01b1fa565bc950810    inbound verifying key
1c4c7aaa7403acf01b1fa565bc950810    outbound encrypting key
1c4c7aaa7403acf01b1fa565bc950810    inbound decrypting key
00000000000000000000000000
783ee50c1d1209b1cb2ea115e7899fbe9547b2b03c1ef14c20d102e0a7282a00
d4919756821af4ff388a5333c51835fc482c71a5f6bdea3745e2dc6124dd3241
c9a9849d267ebc3d7608554962235fb568d6874699cd141fb7395b801b796650
ca0edaee65d2304afe707504067aefb38669eda3547f7c5d946ecf3aadbf437d
6f4288e1d9a6935e63ae7ba467afaac1ccc3529a1c36d7b49cf2e407779e2985
3173db8d814e9b405cc00d01d5d016a022b6c62be96b90c2f792ecce4dc4f36c
f58b2d74980a6dde17d3d8f8e63b0b7264342f600fab05580344bae88ffa5ab9
27df19f96a51bba8f083eb3f10ac218cfdc8fb8ea2255711c74b4f5913e3b896
00000000000000000000000000000000
783ee50c1d1209b1cb2ea115e7899fbe9547b2b03c1ef14c20d102e0a7282a00
d4919756821af4ff388a5333c51835fc482c71a5f6bdea3745e2dc6124dd3241
c9a9849d267ebc3d7608554962235fb568d6874699cd141fb7395b801b796650
ca0edaee65d2304afe707504067aefb38669eda3547f7c5d946ecf3aadbf437d
6f4288e1d9a6935e63ae7ba467afaac1ccc3529a1c36d7b49cf2e407779e2985
3173db8d814e9b405cc00d01d5d016a022b6c62be96b90c2f792ecce4dc4f36c
f58b2d74980a6dde17d3d8f8e63b0b7264342f600fab05580344bae88ffa5ab9
27df19f96a51bba8f083eb3f10ac218cfdc8fb8ea2255711c74b4f5913e3b896
```

```
0000000000000000
54004500530054004e0054005c007400650073007400    "TESTNT\test"

ntlmHash = md4(password) = 0x3b1b47e42e0463276e3ded6cef349f93
ntlmv2Hash = hmac(ntlmHash, "TESTTESTNT") = 0xc4ea95cb148df11bf9d7c3611ad6d722
challenge (from Type 2) = 0x0033b02d17275b77
blob (from Type 3) = 0x0101000000000000502db638677bc301f2e632972
6c598e80000000002000c00540045005300540004e00540001000c004d0045004
d004200450052000300 1e006d0065006d006200650072002e007400650073007
4002e0063006f006d000000000000000000
```

NTLMv2UserSessionKey = HMAC(ntlmv2Hash(HMAC(ntlmv2Hash, challenge + blob))) =
0x1c4c7aaa7403acf01b1fa565bc950810

Key is *not* weakened (NTLM1 only weakens Lan Manager Session Keys).

Called MakeSignature on the server-side context for message
"0x0102030405060708".

Yields signature:

01000000ffffff0051cefea77f098ee3

CRC32(0x0102030405060708) = 0xc588ca3f
Sequence number is 0.
0x00000000 + crc32 + seqnum = 0x00000000c588ca3f00000000

RC4(key, 0x00000000c588ca3f00000000) = 0x8661cb7d51cefea77f098ee3

version num + first 4 bytes overwritten with counter value (0xffffff00 here):

01000000ffffff0051cefea77f098ee3 = signature

Called EncryptMessage twice on the server-side context for message
"0x0102030405060708".

Yields encrypted messages:

```
f483b904264d8306010000098010700bd9719c0b34f5362
022cc2127f9e206e0100000800307001855ec8494231273
```

same RC4 cipher is used from previous signing operation (i.e., not reset):

RC4(0x0102030405060708) = sealed message = 0xf483b904264d8306

trailer buffer gets signature; again uses same RC4 cipher
(sequence number is now 1 because of previous signing):

RC4(0x00000000c588ca3f01000000) = 0x2246e9cbbd9719c0b34f5362
version num + first 4 bytes overwritten w/counter (0x98010700):
0100000098010700bd9719c0b34f5362 = trailer signature

second message is same:

RC4(0x0102030405060708) = sealed message = 0x022cc2127f9e206e

trailer buffer signature with sequence 2:

RC4(0x00000000c588ca3f02000000) = 0x6fde4d031855ec8494231273
version num + first 4 bytes overwritten w/counter (0x80030700)
0100000800307001855ec8494231273 = trailer signature

NTLMv2 Authentication; NTLM2 Signing and Sealing Using the 56-bit NTLMv2 User Session Key

Demonstration of NTLMv2 authentication with 56-bit NTLMv2 User Session Key NTLM2 signing and sealing (without key exchange).

```
LMCompatibilityLevel set to 3 (LMv2/NTLMv2).

AcquireCredentialsHandle called with domain "TESTNT", user "test", password
"test1234".

------------------------------------------------------------------------------
InitializeSecurityContext called with ISC_REQ_INTEGRITY and
ISC_REQ_CONFIDENTIALITY.  Following flags were masked off of resulting Type 1
message:

    Negotiate 128                 (0x20000000)
    Negotiate Key Exchange        (0x40000000)

4e544c4d53535000010000000b782088000000000000000000000000000000000000

4e544c4d53535000    "NTLMSSP"
01000000            Type 1 message
b7820880            Flags
    Negotiate Unicode             (0x00000001)
    Negotiate OEM                 (0x00000002)
    Request Target                (0x00000004)
    Negotiate Sign                (0x00000010)
    Negotiate Seal                (0x00000020)
    Negotiate Lan Manager Key     (0x00000080)
    Negotiate NTLM                (0x00000200)
    Negotiate Always Sign         (0x00008000)
    Negotiate NTLM2 Key           (0x00080000)
    Negotiate 56                  (0x80000000)
0000000000000000    Supplied Domain header (empty, supplied credentials)
0000000000000000    Supplied Workstation header (empty, supplied credentials)

------------------------------------------------------------------------------
AcceptSecurityContext called with ASC_REQ_INTEGRITY and ASC_REQ_CONFIDENTIALITY.
Produces Type 2 message:

4e544c4d53535000020000000c000c0030000000358289805142469 73ea892c1
000000000000000046004600 3c0000005400450053005400 4e00540002000c00
5400450053005400 4e00540001000c004d0045004d004200 4500520003001e00
6d0065006d006200650072002e00740065007300740 02e0063006f006d000000
0000

4e544c4d53535000    "NTLMSSP"
02000000            Type 2 message
0c000c0030000000    Target Name header (length 12, offset 48)
35828980            Flags
    Negotiate Unicode             (0x00000001)
    Request Target                (0x00000004)
    Negotiate Sign                (0x00000010)
    Negotiate Seal                (0x00000020)
    Negotiate NTLM                (0x00000200)
    Negotiate Always Sign         (0x00008000)
    Target Type Domain            (0x00010000)
    Negotiate NTLM2 Key           (0x00080000)
    Negotiate Target Info         (0x00800000)
    Negotiate 56                  (0x80000000)
514246973ea892c1    Challenge
0000000000000000    Context
460046003c000000    Target Information header (length 70, offset 60)
5400450053005400 4e005400    Target Name ("TESTNT")
Target Information block:
    02000c00    NetBIOS Domain Name (length 12)
    5400450053005400 4e005400    "TESTNT"
    01000c00    NetBIOS Server Name (length 12)
```

```
    4d0045004d00420045005200    "MEMBER"
    03001e00    DNS Server Name (length 30)
    6d0065006d006200650072002e0074006500730074002e0063006f006d00
        "member.test.com"
    00000000    Target Information Terminator

--------------------------------------------------------------------------------
InitializeSecurityContext called.  Produces a Type 3 message:

4e544c4d535350000300000018001800600000007600760078000000c000c00
4000000080008004c0000000c000c0054000000000000ee00000035828880
5400450053005400 4e00540074006500730074004d0045004d00420045005200
bf2e015119f6bdb3f6fdb768aa12d478f5ce3d2401c8f6e9caa4da8f25d5e840
974ed8976d3ada4601010000000000000030fa7e3c677bc301f5ce3d2401c8f6e9
0000000002000c0054004500530054004e00540001000c004d0045004d004200
4500520003001e006d0065006d006200650072002e0074006500730074002e00
63006f006d000000000000000000

4e544c4d53535000    "NTLMSSP"
03000000            Type 3 message
1800180060000000    LM/LMv2 Response header (length 24, offset 96)
7600760078000000    NTLM/NTLMv2 Response header (length 118, offset 120)
0c000c0040000000    Domain Name header (length 12, offset 64)
080008004c000000    User Name header (length 8, offset 76)
0c000c0054000000    Workstation Name header (length 12, offset 84)
00000000ee000000    Session Key header (empty)
35828880            Flags
    Negotiate Unicode               (0x00000001)
    Request Target                  (0x00000004)
    Negotiate Sign                  (0x00000010)
    Negotiate Seal                  (0x00000020)
    Negotiate NTLM                  (0x00000200)
    Negotiate Always Sign           (0x00008000)
    Negotiate NTLM2 Key             (0x00080000)
    Negotiate Target Info           (0x00800000)
    Negotiate 56                    (0x80000000)
5400450053005400 4e005400                           Domain Name ("TESTNT")
7400650073007400                                    User Name ("test")
4d0045004d00420045005200                            Workstation Name ("MEMBER")
bf2e015119f6bdb3f6fdb768aa12d478f5ce3d2401c8f6e9    LM/LMv2 Response
NTLM/NTLMv2 Response:
caa4da8f25d5e840974ed8976d3ada4601010000000000000030fa7e3c677bc301
f5ce3d2401c8f6e90000000002000c0054004500530054004e00540001000c00
4d0045004d0042004500520003001e006d0065006d006200650072002e007400
6500730074002e0063006f006d000000000000000000

--------------------------------------------------------------------------------
AcceptSecurityContext called to complete the server-side context.

Server Context Export:

40a42e7840a42e78000000000000000000000000000000000
08ee0b00    server context handle dwUpper
35828981    flags
0000000000000000282e09002c2e0900a82e0900b82f0900000000000000000000
d00200000000000000000000000000000000000000000000000200000000000000
ffffffffffffff7f20010000
f7301e5d23f1d578c51ec0728b67453e    outbound signing key
06403212f9e8c05ce1739938c200eca5    inbound verifying key
3d6483dce52cd6c4d7553545e607d92d    outbound encrypting key
ccc6efbcea980c0ac685753a4c9bbe0c    inbound decrypting key
00000000000000000000000000
3a6d27764b2010cdeda24d421b961cb92835eac0ecd6bf50375d308219362975
443ce13be9d2621d1283dc484f3e6ae66c4e8851f12632c3c1d9d5808763770b
9b61ebf7b7152252da1ff484b611732e3dcca15e9567939ce48bac450c7d46df
14ff026557a49db4dbe54cde538c21c6d0a59a24d87a06498dd74340bc4a13ab
```

```
6f851aa070a7587f9fa86b560d16d4e3b22d97b33147eefccbc87c6e3479f53f
bab898af09865aa3692f5b330a99f600c2b118cec766fec903a6551ecfe0e225
e77bae642ad3c5fa0fbb0e5caac4bd9e07ef2b602c1754f2908eb501be5991fd
41ad688add8f392308f3f974f881783872b07ee8fb7104055fa99289d1f0ca94
00000000000000000000000000000000
21a58f9635cee06b4f94512adc8125717aa7a877750d646503e17073b1e5c1a6
d874c3a2b0499f905bebef1a3f5f8bee8e8ce69c4b87c900bf2f41ea72167f91
1dfc82a959862708575caf3a0a7bbbd7f337ba5ab6fae7c69734e9a1e367a44e
cfc5292ed9da1f5e0f2d01a0ae0214b58a1eb8619b524c7dbe47d41bc83bb346
0983d533dd40ab76c742ca45c0e4c2cbf5e8f6f7f222b76658a31c795005193e
f17812d0c4247e806e4afddb10cc68d6b90c2807b20e9860549318621130ac06
bc15e2f8232b9aec20fb95decd89d2f43d0b69bdd3136f382644888d533284b4
92feedaa7c9df9393c55316cffd1489ead172cdf43995636636a85f05d6d044d
0000000000000000
5400450053005400 4e0054005c007400650073007400    "TESTNT\test"

ntlmHash = md4(password) = 0x3b1b47e42e0463276e3ded6cef349f93
ntlmv2Hash = hmac(ntlmHash, "TESTTESTNT") = 0xc4ea95cb148df11bf9d7c3611ad6d722
challenge (from Type 2) = 0x514246973ea892c1
blob (from Type 3) = 0x0101000000000000 30fa7e3c677bc301f5ce3d240
1c8f6e90000000002000c00540045005300 54004e00540001000c004d0045004
d00420045005200 03001e006d0065006d006200650072002e007400650073007
4002e0063006f006d00000000000000000

NTLMv2UserSessionKey = HMAC(ntlmv2Hash(HMAC(ntlmv2Hash, challenge + blob))) =
    0x62ff13231f566f5dadf7391e183b5f39

We are using server context:

serverSigningConstant =
    "session key to server-to-client signing key magic constant" =
        0x73657373696f6e206b657920746f207365727665722d746f2d636c69656e74
        207369676e696e67206b6579206d6167696320636f6e7374616e7400

serverSealingConstant =
    "session key to server-to-client sealing key magic constant" =
        0x73657373696f6e206b657920746f207365727665722d746f2d636c69656e74
        207365616c696e67206b6579206d6167696320636f6e7374616e7400

signingKey = MD5(key + serverSigningConstant) =
    0xf7301e5d23f1d578c51ec0728b67453e

User Session Key is weakened to 56-bit for sealing by truncating =
    0x62ff13231f566f

sealingKey = MD5(weakenedKey + serverSealingConstant) =
    0x3d6483dce52cd6c4d7553545e607d92d

Called MakeSignature on the server-side context for message
"0x0102030405060708".

Yields signature:

01000000fa317a333d8f510c00000000

Sequence number is 0.
seqnum + message = 0x0000000001020304050607 08

HMAC(signingKey, 0x0000000001020304050607 08) =
    0xfa317a333d8f510cccab257d9f2193c4

version num + first 8 bytes + seqnum:

01000000fa317a333d8f510c00000000 = signature

--------------------------------------------------------------------------------
```

```
Called EncryptMessage twice on the server-side context for message
"0x0102030405060708".

Yields encrypted messages:

a8e6671c79cf26570100000673773407fb60b4201000000
2fe89f6c6ea06d4b01000000244e0bcbce6ec16c02000000

RC4(sealingKey, 0x0102030405060708) = sealed message = 0xa8e6671c79cf2657

trailer buffer gets signature (sequence number is now 1 because of previous
signing):

HMAC(signingKey, 0x01000000010203040506070 8) =
    0x673773407fb60b428f70358ca67720ba
version num + first 8 bytes + seqnum:
01000000673773407fb60b4201000000 = trailer signature

second message is same, using RC4 cipher from previous sealing operation:

RC4(0x0102030405060708) = sealed message = 0x2fe89f6c6ea06d4b

trailer buffer signature with sequence 2:

HMAC(signingKey, 0x02000000010203040506070 8) =
    0x244e0bcbce6ec16ce3493f55854bcc5c
version num + first 8 bytes + seqnum:
01000000244e0bcbce6ec16c02000000 = trailer signature
```

Anonymous NTLMv1 Authentication; NTLM2 Signing and Sealing Using the 128-bit Null User Session Key With Key Exchange Negotiated

Demonstration of anonymous authentication with signing and sealing.

```
LMCompatibilityLevel set to 0 (LM/NTLM).

AcquireCredentialsHandle called with empty string ("") for domain, user, and
password.

------------------------------------------------------------------------------
InitializeSecurityContext called with ISC_REQ_INTEGRITY and
ISC_REQ_CONFIDENTIALITY.  Produces Type 1 message:

4e544c4d53535000 01000000b78208e0 0000000000000000 0000000000000000

4e544c4d53535000    "NTLMSSP"
01000000            Type 1 message
b78208e0            Flags
    Negotiate Unicode              (0x00000001)
    Negotiate OEM                  (0x00000002)
    Request Target                 (0x00000004)
    Negotiate Sign                 (0x00000010)
    Negotiate Seal                 (0x00000020)
    Negotiate Lan Manager Key      (0x00000080)
    Negotiate NTLM                 (0x00000200)
    Negotiate Always Sign          (0x00008000)
    Negotiate NTLM2 Key            (0x00080000)
    Negotiate 128                  (0x20000000)
    Negotiate Key Exchange         (0x40000000)
    Negotiate 56                   (0x80000000)
0000000000000000    Supplied Domain header (empty, supplied credentials)
0000000000000000    Supplied Workstation header (empty, supplied credentials)

------------------------------------------------------------------------------
AcceptSecurityContext called with ASC_REQ_INTEGRITY and ASC_REQ_CONFIDENTIALITY.
```

```
Produces Type 2 message:

4e544c4d53535000020000000c000c0030000000358289e05bce6f12f47ddbdf
0000000000000000460046003c0000005400450053005400 4e00540002000c00
5400450053005400 4e00540001000c004d0045004d0042004500520003001e00
6d0065006d006200650072002e0074006500730074002e0063006f006d000000
0000

4e544c4d53535000    "NTLMSSP"
02000000            Type 2 message
0c000c0030000000    Target Name header (length 12, offset 48)
358289e0            Flags
    Negotiate Unicode              (0x00000001)
    Request Target                 (0x00000004)
    Negotiate Sign                 (0x00000010)
    Negotiate Seal                 (0x00000020)
    Negotiate NTLM                 (0x00000200)
    Negotiate Always Sign          (0x00008000)
    Target Type Domain             (0x00010000)
    Negotiate NTLM2 Key            (0x00080000)
    Negotiate Target Info          (0x00800000)
    Negotiate 128                  (0x20000000)
    Negotiate Key Exchange         (0x40000000)
    Negotiate 56                   (0x80000000)
5bce6f12f47ddbdf    Challenge
0000000000000000    Context
460046003c000000    Target Information header (length 70, offset 60)
5400450053005400 4e005400    Target Name ("TESTNT")
Target Information block:
    02000c00    NetBIOS Domain Name (length 12)
    5400450053005400 4e005400    "TESTNT"
    01000c00    NetBIOS Server Name (length 12)
    4d0045004d00420045005200    "MEMBER"
    03001e00    DNS Server Name (length 30)
    6d0065006d006200650072002e0074006500730074002e0063006f006d00
        "member.test.com"
    00000000    Target Information Terminator

--------------------------------------------------------------------------------
InitializeSecurityContext called.  Produces a Type 3 message:

4e544c4d535350000300000001000 1004c000000000000004d0000000000000
4000000000000000400000000c000c0040000000100010004d000000358a88e0
4d0045004d0042004500520000c1442e6cca8c010e77138430aa35738e

4e544c4d53535000    "NTLMSSP"
03000000            Type 3 message
010001004c000000    LM/LMv2 Response header (length 1, offset 76)
000000004d000000    NTLM/NTLMv2 Response header (empty)
0000000040000000    Domain Name header (empty)
0000000040000000    User Name header (empty)
0c000c0040000000    Workstation Name header (length 12, offset 64)
100010004d000000    Session Key header (length 16, offset 77)
358a88e0            Flags
    Negotiate Unicode              (0x00000001)
    Request Target                 (0x00000004)
    Negotiate Sign                 (0x00000010)
    Negotiate Seal                 (0x00000020)
    Negotiate NTLM                 (0x00000200)
    Negotiate Anonymous            (0x00000800)
    Negotiate Always Sign          (0x00008000)
    Negotiate NTLM2 Key            (0x00080000)
    Negotiate Target Info          (0x00800000)
    Negotiate 128                  (0x20000000)
    Negotiate Key Exchange         (0x40000000)
    Negotiate 56                   (0x80000000)
```

```
4d0045004d00420045005200            Workstation Name ("MEMBER")
00                                  LM/LMv2 Response
c1442e6cca8c010e77138430aa35738e    Session Key

------------------------------------------------------------------------------
AcceptSecurityContext called to complete the server-side context.

Server Context Export (on Win2k, appears to vary by OS):

40a42e7840a42e78000000000000000000000000000000
08ee0b00    server context handle dwUpper
358a89e1    flags
0000000000000000e8290900ec290900682a0900782b0900000000000000000000
0000000000000000000000000000000000000000000000000200000000000000
ffffffffffffff7f00000000
9128c3e5df618a48a83b44cfd92d58fe    outbound signing key
594757aaa803afd943de25e087e3f9f1    inbound verifying key
fc52e8bf1605ab57e89c6d6b4ffa92f6    outbound encrypting key
96465577ba181d141711572e5e15fe5d    inbound decrypting key
000000000000000000000000
53a979fb1564d02ea0c33a32b0a1b2015f261b6ee6207804aa4f88fc57826b61
5a9fe8764931f9903efea3e1d258cddda292d6da10b536287a9859ff212d6781
f51a0eb439afc1f30cf0a75d4b54145c5bf4e7036a95c573dc0a3b479dcbbb09
6c4c698b1fc22acc8eeb8f7183ea99182274d1e4dfe5bf8d13382941f6ca25b7
7084a56216c906ad7da891fa42082b9b170733b65ef23437a4fd667e5686c6de
87240080ec89d3d4bc194e759aaebde302d55550f83f1d0b46303545c71eb3b1
3c2f6f7b9eb8ab6d1c4011f12c51baf77fcec00fd7e07248e2604485be238aef
77b91265ac5227ee944ded8cd868634a973dc4db439c0dc8a696937ccfd9e905
00000000000000000000000000000000
09dd12ae4213ac358896e19a9d26cfc8957750b5f557bae5f3ef98d891a4059f
f25ed0195d141aea3d62e31fd7a9aa1e07fd83cb4d3f7401825b7e66a14b28cd
52d6e8a2049036756dfcb929d220d5463499ca3368a071634c0c9cd10dcc4ef9
11d30f108a45648979477f0641c2b485f6adc0a8922be91dde847c93278040a6
b886c6605ac132fe731c589467a76131c9498db13beb2f9baf2c59e06f6c6af0
bbc4b00e6bce78da2de602e797f103b2b6bf4f1b388c0816dfbde2258bd9a3b7
157b5600c55ffaf79e3ac78e48ee55308737ff43817d515cf83e6e70c317bcec
2aed21442e24d465a56923be0b53f4fb8fb3e454ab18760a7adc2239723c4adb
0000000000000000

Anonymous User Session Key = 0x00000000000000000000000000000000

Key exchange performed:
type3Key (from Type 3) = 0xc1442e6cca8c010e77138430aa35738e
key = RC4(AnonymousUserSessionKey, type3Key) =
    0x1f5ca72d69bb5c34fd159a57fd5be1e3

We are using server context:

serverSigningConstant =
    "session key to server-to-client signing key magic constant" =
        0x73657373696f6e206b657920746f207365727665722d746f2d636c69656e74
        207369676e696e67206b6579206d6167696320636f6e7374616e7400

serverSealingConstant =
    "session key to server-to-client sealing key magic constant" =
        0x73657373696f6e206b657920746f207365727665722d746f2d636c69656e74
        207365616c696e67206b6579206d6167696320636f6e7374616e7400

signingKey = MD5(key + serverSigningConstant) =
    0x9128c3e5df618a48a83b44cfd92d58fe

sealingKey = MD5(key + serverSealingConstant) =
    0xfc52e8bf1605ab57e89c6d6b4ffa92f6

Called MakeSignature on the server-side context for message
"0x0102030405060708".
```

```
Yields signature:

01000000ae0cbe0dd0b2110300000000

Sequence number is 0.
seqnum + message = 0x0000000001020304050607080

HMAC(signingKey, 0x0000000001020304050607080) =
    0xc14074817421530195b4532d61a4e625

sig = RC4(sealingKey, first 8 bytes) = 0xae0cbe0dd0b21103
version num + sig + seqnum:

01000000ae0cbe0dd0b2110300000000 = signature

--------------------------------------------------------------------------------
Called EncryptMessage twice on the server-side context for message
"0x0102030405060708".

Yields encrypted messages:

fb2e1d6ff8a3569a01000000cc2c5bf59319e7ca01000000
1e2216588e5a7d9801000000e9a3066b8fab0bf102000000

Uses RC4 cipher from previous signing operation (key exchange encrypts
signature with sealing key as well).

RC4(0x0102030405060708) = sealed message = 0xfb2e1d6ff8a3569a

trailer buffer gets signature (sequence number is now 1 because of previous
signing, same RC4 cipher used for this signature too):

HMAC(signingKey, 0x010000000102030405060708) =
    0x25141e76f441b10622cfaf0e61450663
sig = RC4(first 8 bytes) = 0xcc2c5bf59319e7ca
version num + sig + seqnum:
01000000cc2c5bf59319e7ca01000000 = trailer signature

second message is same, using RC4 cipher from previous operations:

RC4(0x0102030405060708) = sealed message = 0x1e2216588e5a7d98

trailer buffer signature with sequence 2:

HMAC(signingKey, 0x020000000102030405060708) =
   0x7efa2ba18ed911696a0f4d571b05a244
sig = RC4(first 8 bytes) = 0xe9a3066b8fab0bf1
version num + sig + seqnum:
01000000e9a3066b8fab0bf102000000 = trailer signature
```

Local NTLMv1 Authentication; NTLM2 Signing and Sealing Using an Unknown Session Key With Key Exchange Negotiated (Analysis Incomplete)

Demonstration of local authentication with signing and sealing. It is assumed that the User Session Key derivation and Key Exchange processes are performed normally offline within the established local context (in the absence of the required information being provided over-the-wire).

```
LMCompatibilityLevel set to 0 (LM/NTLM).

AcquireCredentialsHandle called with NULL identity (using default credentials
handle for authenticated local user).

--------------------------------------------------------------------------------
```

```
InitializeSecurityContext called with ISC_REQ_INTEGRITY and
ISC_REQ_CONFIDENTIALITY.  Produces Type 1 message:

4e544c4d5353500001000000b7b208e006000600260000000600060020000000
4d454d424552544553544e54

4e544c4d53535000    "NTLMSSP"
01000000            Type 1 message
b7b208e0            Flags
    Negotiate Unicode              (0x00000001)
    Negotiate OEM                  (0x00000002)
    Request Target                 (0x00000004)
    Negotiate Sign                 (0x00000010)
    Negotiate Seal                 (0x00000020)
    Negotiate Lan Manager Key      (0x00000080)
    Negotiate NTLM                 (0x00000200)
    Negotiate Domain Supplied      (0x00001000)
    Negotiate Workstation Supplied (0x00002000)
    Negotiate Always Sign          (0x00008000)
    Negotiate NTLM2 Key            (0x00080000)
    Negotiate 128                  (0x20000000)
    Negotiate Key Exchange         (0x40000000)
    Negotiate 56                   (0x80000000)
0600060026000000    Supplied Domain header (length 6, offset 38)
0600060020000000    Supplied Workstation header (length 6, offset 32)
4d454d424552    Supplied Workstation ("MEMBER")
544553544e54    Supplied Domain ("TESTNT")

--------------------------------------------------------------------------------
AcceptSecurityContext called with ASC_REQ_INTEGRITY and ASC_REQ_CONFIDENTIALITY.
Produces Type 2 message:

4e544c4d5353500002000000c000c003000000035c289e0d7ef496afa055352
08ee0b0000000000460046003c0000005400450053005400 4e00540002000c00
5400450053005400 4e00540001000c004d0045004d00420045005200 03001e00
6d0065006d006200650072002e00740065007300740 02e0063006f006d000000
0000

4e544c4d53535000    "NTLMSSP"
02000000            Type 2 message
0c000c0030000000    Target Name header (length 12, offset 48)
35c289e0            Flags
    Negotiate Unicode              (0x00000001)
    Request Target                 (0x00000004)
    Negotiate Sign                 (0x00000010)
    Negotiate Seal                 (0x00000020)
    Negotiate NTLM                 (0x00000200)
    Negotiate Local Call           (0x00004000)
    Negotiate Always Sign          (0x00008000)
    Target Type Domain             (0x00010000)
    Negotiate NTLM2 Key            (0x00080000)
    Negotiate Target Info          (0x00800000)
    Negotiate 128                  (0x20000000)
    Negotiate Key Exchange         (0x40000000)
    Negotiate 56                   (0x80000000)
d7ef496afa055352    Challenge
08ee0b0000000000    Context (equal to server context handle's dwUpper field)
460046003c000000    Target Information header (length 70, offset 60)
5400450053005400 4e005400    Target Name ("TESTNT")
Target Information block:
    02000c00    NetBIOS Domain Name (length 12)
    5400450053005400 4e005400    "TESTNT"
    01000c00    NetBIOS Server Name (length 12)
    4d0045004d00420045005200    "MEMBER"
    03001e00    DNS Server Name (length 30)
    6d0065006d006200650072002e00740065007300740 02e0063006f006d00
```

```
        "member.test.com"
    00000000    Target Information Terminator

-------------------------------------------------------------------------------
InitializeSecurityContext called.  Produces a Type 3 message:

4e544c4d535350000300000000000000400000000000000040000000000000000
400000000000000040000000000000004000000000000000400000035c288e0

4e544c4d53535000    "NTLMSSP"
03000000            Type 3 message
0000000040000000    LM/LMv2 Response header (empty, local authentication)
0000000040000000    NTLM/NTLMv2 Response header (empty, local authentication)
0000000040000000    Domain Name header (empty, local authentication)
0000000040000000    User Name header (empty, local authentication)
0000000040000000    Workstation Name header (empty, local authentication)
0000000040000000    Session Key header (empty)
35c288e0            Flags
    Negotiate Unicode              (0x00000001)
    Request Target                 (0x00000004)
    Negotiate Sign                 (0x00000010)
    Negotiate Seal                 (0x00000020)
    Negotiate NTLM                 (0x00000200)
    Negotiate Local Call           (0x00004000)
    Negotiate Always Sign          (0x00008000)
    Negotiate NTLM2 Key            (0x00080000)
    Negotiate Target Info          (0x00800000)
    Negotiate 128                  (0x20000000)
    Negotiate Key Exchange         (0x40000000)
    Negotiate 56                   (0x80000000)

-------------------------------------------------------------------------------
AcceptSecurityContext called to complete the server-side context.

Server Context Export:

40a42e7840a42e78000000000000000000000000000000
08ee0b00    server context handle dwUpper
35c289e1    flags
0000000000000000182e09001c2e0900982e0900a82f0900000000000000000
d00200000000000000000000000000000000000000000000200000000000000
ffffffffffffff7f00000000
65836771d7bf52df502634e521b3154b    outbound signing key
9ed7ce6558f0f26db1fbf5163d7b24f9    inbound verifying key
f56d6b81749b7adaf3467f37bc36224e    outbound encrypting key
65adf5ede40e84e7338d681e6cbc38ff    inbound decrypting key
00000000000000000000000
7263d05c01f686ab2717e06f35c91b7a10fd0f5289ced6b6c423a213e6256a4d
3d128d30290899cbb502412f2d77ae1d4e55cf3fa6df5a785ded4aa3504c5484
155396d88345ff065beab3b738956e07f2b0ad36b8a80ba9f49305753e14e540
3428b4ccd3641ad18b037e9b67be3c1e70a17d816520565e744bd587bfc5b27f
8558099048ec800af1a5bbef92cdd2afe8c3042e510026f3bac7579ec8db5fbd
eb24bc43a742fc6111def922666dfe9d4791e9c60ef0d431d98cc259e2162b7c
49c0da1c2af5ee8e4f440c9a60ca4668391f3b213288f7a46be1c19f198a9c79
699737aadcfb829476f8d7b17be42c73338f71a0fa986c0dac18b9e3e73add62
00000000000000000000000000000000
651771fae2f5fc6da8973bd149126e92212b2f896101d23c568e40b325a383bd
42ec2752b295e5d09e36c899a94fffb8e053ac1c5c72d4de7c66b14b8d862977
bf5f47cfb76a81201f829098a067df7b78dc51630d8f790e0985316cb459e7d5
7f466b6f5a13948b38434ac54c7ee9ab340a6926d75d2aa523d81d0bc23ae4cc
aa1a419fc303cd88f6a4be04e100bb32e6cbf4feae9ab04808d9a111f937fb76
50104dc1c696b6c0740528f193f87d14cedd3f442e15d324454e64c7733e0f9c
ea0c163de8f3c92d1eebed33ba30f7f2350757d6b98abccafd7568229d1bef7a
5519da065e709be3ad2c8418dbb5f002a2eea68c395891c454af62806087a75b
0000000000000000
5400450053005400 4e0054005c007400650073007400    "TESTNT\test"
```

```
User Session Key = ????

Key exchange performed somehow?  Shows up in signatures.

We are using server context:

serverSigningConstant =
    "session key to server-to-client signing key magic constant" =
        0x73657373696f6e206b657920746f207365727665722d746f2d636c69656e74
        207369676e696e67206b6579206d6167696320636f6e7374616e7400

serverSealingConstant =
    "session key to server-to-client sealing key magic constant" =
        0x73657373696f6e206b657920746f207365727665722d746f2d636c69656e74
        207365616c696e67206b6579206d6167696320636f6e7374616e7400

signingKey = MD5(key + serverSigningConstant) =
    0x65836771d7bf52df502634e521b3154b (from context export)

sealingKey = MD5(key + serverSealingConstant) =
    0xf56d6b81749b7adaf3467f37bc36224e (from context export)

Called MakeSignature on the server-side context for message
"0x0102030405060708".

Yields signature:

01000000adc70c8bd4834ae800000000

Sequence number is 0.
seqnum + message = 0x0000000001020304050607080

HMAC(signingKey, 0x0000000001020304050607080) =
    0xb0bd637382da53f4ad3edf2c5a2c9592

sig = RC4(sealingKey, first 8 bytes) = 0xadc70c8bd4834ae8
version num + sig + seqnum:

01000000adc70c8bd4834ae800000000 = signature

--------------------------------------------------------------------------------
Called EncryptMessage twice on the server-side context for message
"0x0102030405060708".

Yields encrypted messages:

9c89720008251ab601000000d4af70567f5ab1df01000000
a8950e40b10e9af501000000c5447ccf7102d98c02000000

Uses RC4 cipher from previous signing operation (key exchange encrypts
signature with sealing key as well).

RC4(0x0102030405060708) = sealed message = 0x9c89720008251ab6

trailer buffer gets signature (sequence number is now 1 because of previous
signing, same RC4 cipher used for this signature too):

HMAC(signingKey, 0x010000000102030405060708) =
    0xcd1d04a1962b7d82e6ffd7c37871bf15
sig = RC4(first 8 bytes) = 0xd4af70567f5ab1df
version num + sig + seqnum:
01000000d4af70567f5ab1df01000000 = trailer signature

second message is same, using RC4 cipher from previous operations:
```

```
RC4(0x0102030405060708) = sealed message = 0xa8950e40b10e9af5

trailer buffer signature with sequence 2:

HMAC(signingKey, 0x020000000102030405060708) =
   0x3011549106db10de5694f567749cb9eb
sig = RC4(first 8 bytes) = 0xc5447ccf7102d98c
version num + sig + seqnum:
01000000c5447ccf7102d98c02000000 = trailer signature
```

Appendix D: Java Implementation of the Type 3 Response Calculations

Listed below is an annotated sample implementation of the various Type 3 response calculations in Java. This example requires a JCE provider implementing the MD4 message-digest algorithm; the author recommends GNU Crypto, available at http://www.gnu.org/software/gnu-crypto/.

```
import java.security.Key;
import java.security.MessageDigest;

import javax.crypto.Cipher;

import javax.crypto.spec.SecretKeySpec;

/**
 * Calculates the various Type 3 responses.
 */
public class Responses {

    /**
     * Calculates the LM Response for the given challenge, using the specified
     * password.
     *
     * @param password The user's password.
     * @param challenge The Type 2 challenge from the server.
     *
     * @return The LM Response.
     */
    public static byte[] getLMResponse(String password, byte[] challenge)
            throws Exception {
        byte[] lmHash = lmHash(password);
        return lmResponse(lmHash, challenge);
    }

    /**
     * Calculates the NTLM Response for the given challenge, using the
     * specified password.
     *
     * @param password The user's password.
     * @param challenge The Type 2 challenge from the server.
     *
     * @return The NTLM Response.
     */
    public static byte[] getNTLMResponse(String password, byte[] challenge)
            throws Exception {
        byte[] ntlmHash = ntlmHash(password);
        return lmResponse(ntlmHash, challenge);
    }

    /**
     * Calculates the NTLMv2 Response for the given challenge, using the
     * specified authentication target, username, password, target information
```

```
     * block, and client nonce.
     *
     * @param target The authentication target (i.e., domain).
     * @param user The username.
     * @param password The user's password.
     * @param targetInformation The target information block from the Type 2
     * message.
     * @param challenge The Type 2 challenge from the server.
     * @param clientNonce The random 8-byte client nonce.
     *
     * @return The NTLMv2 Response.
     */
    public static byte[] getNTLMv2Response(String target, String user,
            String password, byte[] targetInformation, byte[] challenge,
                    byte[] clientNonce) throws Exception {
        byte[] ntlmv2Hash = ntlmv2Hash(target, user, password);
        byte[] blob = createBlob(targetInformation, clientNonce);
        return lmv2Response(ntlmv2Hash, blob, challenge);
    }

    /**
     * Calculates the LMv2 Response for the given challenge, using the
     * specified authentication target, username, password, and client
     * challenge.
     *
     * @param target The authentication target (i.e., domain).
     * @param user The username.
     * @param password The user's password.
     * @param challenge The Type 2 challenge from the server.
     * @param clientNonce The random 8-byte client nonce.
     *
     * @return The LMv2 Response.
     */
    public static byte[] getLMv2Response(String target, String user,
            String password, byte[] challenge, byte[] clientNonce)
                    throws Exception {
        byte[] ntlmv2Hash = ntlmv2Hash(target, user, password);
        return lmv2Response(ntlmv2Hash, clientNonce, challenge);
    }

    /**
     * Calculates the NTLM2 Session Response for the given challenge, using the
     * specified password and client nonce.
     *
     * @param password The user's password.
     * @param challenge The Type 2 challenge from the server.
     * @param clientNonce The random 8-byte client nonce.
     *
     * @return The NTLM2 Session Response.  This is placed in the NTLM
     * response field of the Type 3 message; the LM response field contains
     * the client nonce, null-padded to 24 bytes.
     */
    public static byte[] getNTLM2SessionResponse(String password,
            byte[] challenge, byte[] clientNonce) throws Exception {
        byte[] ntlmHash = ntlmHash(password);
        MessageDigest md5 = MessageDigest.getInstance("MD5");
        md5.update(challenge);
        md5.update(clientNonce);
        byte[] sessionHash = new byte[8];
        System.arraycopy(md5.digest(), 0, sessionHash, 0, 8);
        return lmResponse(ntlmHash, sessionHash);
    }

    /**
     * Creates the LM Hash of the user's password.
     *
```

```
     * @param password The password.
     *
     * @return The LM Hash of the given password, used in the calculation
     * of the LM Response.
     */
    private static byte[] lmHash(String password) throws Exception {
        byte[] oemPassword = password.toUpperCase().getBytes("US-ASCII");
        int length = Math.min(oemPassword.length, 14);
        byte[] keyBytes = new byte[14];
        System.arraycopy(oemPassword, 0, keyBytes, 0, length);
        Key lowKey = createDESKey(keyBytes, 0);
        Key highKey = createDESKey(keyBytes, 7);
        byte[] magicConstant = "KGS!@#$%".getBytes("US-ASCII");
        Cipher des = Cipher.getInstance("DES/ECB/NoPadding");
        des.init(Cipher.ENCRYPT_MODE, lowKey);
        byte[] lowHash = des.doFinal(magicConstant);
        des.init(Cipher.ENCRYPT_MODE, highKey);
        byte[] highHash = des.doFinal(magicConstant);
        byte[] lmHash = new byte[16];
        System.arraycopy(lowHash, 0, lmHash, 0, 8);
        System.arraycopy(highHash, 0, lmHash, 8, 8);
        return lmHash;
    }

    /**
     * Creates the NTLM Hash of the user's password.
     *
     * @param password The password.
     *
     * @return The NTLM Hash of the given password, used in the calculation
     * of the NTLM Response and the NTLMv2 and LMv2 Hashes.
     */
    private static byte[] ntlmHash(String password) throws Exception {
        byte[] unicodePassword = password.getBytes("UnicodeLittleUnmarked");
        MessageDigest md4 = MessageDigest.getInstance("MD4");
        return md4.digest(unicodePassword);
    }

    /**
     * Creates the NTLMv2 Hash of the user's password.
     *
     * @param target The authentication target (i.e., domain).
     * @param user The username.
     * @param password The password.
     *
     * @return The NTLMv2 Hash, used in the calculation of the NTLMv2
     * and LMv2 Responses.
     */
    private static byte[] ntlmv2Hash(String target, String user,
            String password) throws Exception {
        byte[] ntlmHash = ntlmHash(password);
        String identity = user.toUpperCase() + target;
        return hmacMD5(identity.getBytes("UnicodeLittleUnmarked"), ntlmHash);
    }

    /**
     * Creates the LM Response from the given hash and Type 2 challenge.
     *
     * @param hash The LM or NTLM Hash.
     * @param challenge The server challenge from the Type 2 message.
     *
     * @return The response (either LM or NTLM, depending on the provided
     * hash).
     */
    private static byte[] lmResponse(byte[] hash, byte[] challenge)
            throws Exception {
```

```
        byte[] keyBytes = new byte[21];
        System.arraycopy(hash, 0, keyBytes, 0, 16);
        Key lowKey = createDESKey(keyBytes, 0);
        Key middleKey = createDESKey(keyBytes, 7);
        Key highKey = createDESKey(keyBytes, 14);
        Cipher des = Cipher.getInstance("DES/ECB/NoPadding");
        des.init(Cipher.ENCRYPT_MODE, lowKey);
        byte[] lowResponse = des.doFinal(challenge);
        des.init(Cipher.ENCRYPT_MODE, middleKey);
        byte[] middleResponse = des.doFinal(challenge);
        des.init(Cipher.ENCRYPT_MODE, highKey);
        byte[] highResponse = des.doFinal(challenge);
        byte[] lmResponse = new byte[24];
        System.arraycopy(lowResponse, 0, lmResponse, 0, 8);
        System.arraycopy(middleResponse, 0, lmResponse, 8, 8);
        System.arraycopy(highResponse, 0, lmResponse, 16, 8);
        return lmResponse;
    }

    /**
     * Creates the LMv2 Response from the given hash, client data, and
     * Type 2 challenge.
     *
     * @param hash The NTLMv2 Hash.
     * @param clientData The client data (blob or client nonce).
     * @param challenge The server challenge from the Type 2 message.
     *
     * @return The response (either NTLMv2 or LMv2, depending on the
     * client data).
     */
    private static byte[] lmv2Response(byte[] hash, byte[] clientData,
            byte[] challenge) throws Exception {
        byte[] data = new byte[challenge.length + clientData.length];
        System.arraycopy(challenge, 0, data, 0, challenge.length);
        System.arraycopy(clientData, 0, data, challenge.length,
                         clientData.length);
        byte[] mac = hmacMD5(data, hash);
        byte[] lmv2Response = new byte[mac.length + clientData.length];
        System.arraycopy(mac, 0, lmv2Response, 0, mac.length);
        System.arraycopy(clientData, 0, lmv2Response, mac.length,
                         clientData.length);
        return lmv2Response;
    }

    /**
     * Creates the NTLMv2 blob from the given target information block and
     * client nonce.
     *
     * @param targetInformation The target information block from the Type 2
     * message.
     * @param clientNonce The random 8-byte client nonce.
     *
     * @return The blob, used in the calculation of the NTLMv2 Response.
     */
    private static byte[] createBlob(byte[] targetInformation,
            byte[] clientNonce) {
        byte[] blobSignature = new byte[] {
            (byte) 0x01, (byte) 0x01, (byte) 0x00, (byte) 0x00
        };
        byte[] reserved = new byte[] {
            (byte) 0x00, (byte) 0x00, (byte) 0x00, (byte) 0x00
        };
        byte[] unknown1 = new byte[] {
            (byte) 0x00, (byte) 0x00, (byte) 0x00, (byte) 0x00
        };
        byte[] unknown2 = new byte[] {
```

```
            (byte) 0x00, (byte) 0x00, (byte) 0x00, (byte) 0x00
        };
        long time = System.currentTimeMillis();
        time += 11644473600000l; // milliseconds from January 1, 1601 -> epoch.
        time *= 10000; // tenths of a microsecond.
        // convert to little-endian byte array.
        byte[] timestamp = new byte[8];
        for (int i = 0; i < 8; i++) {
            timestamp[i] = (byte) time;
            time >>>= 8;
        }
        byte[] blob = new byte[blobSignature.length + reserved.length +
                               timestamp.length + clientNonce.length +
                               unknown1.length + targetInformation.length +
                               unknown2.length];
        int offset = 0;
        System.arraycopy(blobSignature, 0, blob, offset, blobSignature.length);
        offset += blobSignature.length;
        System.arraycopy(reserved, 0, blob, offset, reserved.length);
        offset += reserved.length;
        System.arraycopy(timestamp, 0, blob, offset, timestamp.length);
        offset += timestamp.length;
        System.arraycopy(clientNonce, 0, blob, offset,
                         clientNonce.length);
        offset += clientNonce.length;
        System.arraycopy(unknown1, 0, blob, offset, unknown1.length);
        offset += unknown1.length;
        System.arraycopy(targetInformation, 0, blob, offset,
                         targetInformation.length);
        offset += targetInformation.length;
        System.arraycopy(unknown2, 0, blob, offset, unknown2.length);
        return blob;
    }

    /**
     * Calculates the HMAC-MD5 hash of the given data using the specified
     * hashing key.
     *
     * @param data The data for which the hash will be calculated.
     * @param key The hashing key.
     *
     * @return The HMAC-MD5 hash of the given data.
     */
    private static byte[] hmacMD5(byte[] data, byte[] key) throws Exception {
        byte[] ipad = new byte[64];
        byte[] opad = new byte[64];
        for (int i = 0; i < 64; i++) {
            ipad[i] = (byte) 0x36;
            opad[i] = (byte) 0x5c;
        }
        for (int i = key.length - 1; i >= 0; i--) {
            ipad[i] ^= key[i];
            opad[i] ^= key[i];
        }
        byte[] content = new byte[data.length + 64];
        System.arraycopy(ipad, 0, content, 0, 64);
        System.arraycopy(data, 0, content, 64, data.length);
        MessageDigest md5 = MessageDigest.getInstance("MD5");
        data = md5.digest(content);
        content = new byte[data.length + 64];
        System.arraycopy(opad, 0, content, 0, 64);
        System.arraycopy(data, 0, content, 64, data.length);
        return md5.digest(content);
    }

    /**
```

```
     * Creates a DES encryption key from the given key material.
     *
     * @param bytes A byte array containing the DES key material.
     * @param offset The offset in the given byte array at which
     * the 7-byte key material starts.
     *
     * @return A DES encryption key created from the key material
     * starting at the specified offset in the given byte array.
     */
    private static Key createDESKey(byte[] bytes, int offset) {
        byte[] keyBytes = new byte[7];
        System.arraycopy(bytes, offset, keyBytes, 0, 7);
        byte[] material = new byte[8];
        material[0] = keyBytes[0];
        material[1] = (byte) (keyBytes[0] << 7 | (keyBytes[1] & 0xff) >>> 1);
        material[2] = (byte) (keyBytes[1] << 6 | (keyBytes[2] & 0xff) >>> 2);
        material[3] = (byte) (keyBytes[2] << 5 | (keyBytes[3] & 0xff) >>> 3);
        material[4] = (byte) (keyBytes[3] << 4 | (keyBytes[4] & 0xff) >>> 4);
        material[5] = (byte) (keyBytes[4] << 3 | (keyBytes[5] & 0xff) >>> 5);
        material[6] = (byte) (keyBytes[5] << 2 | (keyBytes[6] & 0xff) >>> 6);
        material[7] = (byte) (keyBytes[6] << 1);
        oddParity(material);
        return new SecretKeySpec(material, "DES");
    }

    /**
     * Applies odd parity to the given byte array.
     *
     * @param bytes The data whose parity bits are to be adjusted for
     * odd parity.
     */
    private static void oddParity(byte[] bytes) {
        for (int i = 0; i < bytes.length; i++) {
            byte b = bytes[i];
            boolean needsParity = (((b >>> 7) ^ (b >>> 6) ^ (b >>> 5) ^
                                    (b >>> 4) ^ (b >>> 3) ^ (b >>> 2) ^
                                    (b >>> 1)) & 0x01) == 0;
            if (needsParity) {
                bytes[i] |= (byte) 0x01;
            } else {
                bytes[i] &= (byte) 0xfe;
            }
        }
    }

}
```


The most current version of this document may be obtained from http://davenport.sourceforge.net/ntlm.html. The author may be contacted vie e-mail at *eric.glass at gmail.com*.