

The NTLM Authentication Protocol

Abstract

This article seeks to describe NTLM 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 in the open-source jCIFS library, available at <http://jcifs.samba.org>.

Contents

- [What is NTLM?](#)
- [NTLM Data Types](#)
- [The NTLM Message Header Layout](#)
 - [The NTLM Flags](#)
- [The Type 1 Message](#)
 - [Type 1 Message Example](#)
- [The Type 2 Message](#)
 - [Type 2 Message Example](#)
- [The Type 3 Message](#)
 - [Responding to the Challenge](#)
 - [The LM Response](#)
 - [The NTLM Response](#)
 - [The NTLMv2 Response](#)
 - [The LMv2 Response](#)
 - [The NTLM2 Session Response](#)
 - [Type 3 Message Example](#)
- [NTLM Version 2](#)
- [NTLMSSP and SSPI](#)
 - [Local Authentication](#)
 - [Datagram Authentication](#)
- [NTLM HTTP Authentication](#)
- [NTLM POP3 Authentication](#)
- [NTLM IMAP Authentication](#)
- [NTLM SMTP Authentication](#)
- [Links and References](#)
- [Appendix A: Java Implementation of the Type 3 Response Calculations](#)

What is NTLM?

NTLM is an authentication protocol 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.

NTLM employs a challenge-response mechanism for authentication, in which clients are able to prove their identities without sending a password to the server. It consists 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.

NTLM Data Types

Before we start digging in any further, we will need to define a few data types as used in the messages.

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 hexadecimal 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 "0x48454c4c4c4f".

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 in bytes.

2. A `short` containing the allocated space for the buffer in bytes (typically, though not necessarily, 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 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. Many of these flags are not used, or are used infrequently; commonly occurring flags are displayed in bold in the table below. Flags marked as "unidentified" or "unknown" 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	<i>unknown</i>	<i>This flag's usage has not been identified.</i>
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	<i>This flag's usage has not been identified.</i>
0x00000200	Negotiate NTLM	Indicates that NTLM authentication is being used.
0x00000400	<i>unknown</i>	<i>This flag's usage has not been identified.</i>
0x00000800	<i>unknown</i>	<i>This flag's usage has not been identified.</i>
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	<i>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.</i>
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	<i>This flag's usage has not been identified.</i>
0x00200000	Request Accept Response	<i>This flag's usage has not been identified.</i>
0x00400000	Request Non-NT Session Key	<i>This flag's usage has not been identified.</i>
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	<i>unknown</i>	<i>This flag's usage has not been identified.</i>
0x02000000	<i>unknown</i>	<i>This flag's usage has not been identified.</i>
0x04000000	<i>unknown</i>	<i>This flag's usage has not been identified.</i>
0x08000000	<i>unknown</i>	<i>This flag's usage has not been identified.</i>
0x10000000	<i>unknown</i>	<i>This flag's usage has not been identified.</i>
0x20000000	Negotiate 128	Indicates that 128-bit encryption is supported.
0x40000000	Negotiate Key Exchange	Indicates that the client will provide an encrypted master session key in the "Session Key" field of the Type 3 message. This is used in signing and sealing, and is RC4-encrypted using the previous session key as the encryption key.
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 (<i>Optional</i>)	security buffer
(24) Supplied Workstation (<i>Optional</i>)	security buffer
(32) <i>start of data block (if required)</i>	

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.

Note that the supplied domain and workstation are optional fields; they may be empty (security buffer indicating a length of zero), or may not be sent at all (security buffer omitted altogether). If the supplied domain and workstation are omitted, the Type 1 message carries no data block (the message ends after the flags field, and is a fixed-length 16-byte structure). The "most-minimal" well-formed Type 1 message, therefore, would be:

```
4e544c4d535350000100000002020000
```

This message contains 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:

```
4e544c4d535350000100000007320000060006002b0000000b000b0020000000
574f524b53544154494f4e444f4d41494e
```

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	0x060006002b000000	Supplied Domain Security Buffer: Length: 6 bytes (0x0600) Allocated Space: 6 bytes (0x0600) Offset: 43 bytes (0x2b000000)
24	0x0b000b0020000000	Supplied Workstation Security Buffer: Length: 11 bytes (0x0b00) Allocated Space: 11 bytes (0x0b00) Offset: 32 bytes (0x20000000)
32	0x574f524b53544154494f4e	Supplied Workstation Data ("WORKSTATION")
43	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).
- 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 (<i>optional</i>)	8 bytes (two consecutive longs)
(40)	Target Information (<i>optional</i>)	security buffer
32 (48)	<i>start of data block</i>	

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 context and target information may be omitted, in which case the data block begins at offset 32 (immediately following the challenge). A minimal Type 2 message would look something like this:

```
4e544c4d53535000020000000000000000000000000020200000123456789abcdef
```

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
49004e0001000c0053004500520056004500520004001400
64006f006d00610069006e002e0063006f006d0003002200
7300650072007600650072002e0064006f006d0061006900
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	<p>Flags:</p> <p>Negotiate Unicode (0x00000001) Negotiate NTLM (0x00000200) Target Type Domain (0x00010000) Negotiate Target Info (0x00800000)</p>						
24	0x0123456789abcdef	Challenge						
32	0x0000000000000000	Context						
40	0x620062003c000000	<p>Target Information Security Buffer:</p> <p>Length: 98 bytes (0x6200) Allocated Space: 98 bytes (0x6200) Offset: 60 bytes (0x3c000000)</p>						
48	0x44004f004d004100 49004e00	Target Name Data ("DOMAIN")						
60	0x02000c0044004f00 4d00410049004e00 01000c0053004500 5200560045005200 0400140064006f00 6d00610069006e00 2e0063006f006d00 0300220073006500 7200760065007200 2e0064006f006d00 610069006e002e00 63006f006d000000 0000	<p>Target Information Data:</p> <table border="1"> <tr> <td>0x02000c0044004f00 4d00410049004e00</td> <td> <p>Domain name subblock:</p> <p>Type: 2 (Domain name, 0x0200) Length: 12 bytes (0x0c00) Data: "DOMAIN"</p> </td> </tr> <tr> <td>0x01000c0053004500 5200560045005200</td> <td> <p>Server name subblock:</p> <p>Type: 1 (Server name, 0x0100) Length: 12 bytes (0x0c00) Data: "SERVER"</p> </td> </tr> <tr> <td>0x0400140064006f00 6d00610069006e00 2e0063006f006d00</td> <td> <p>DNS domain name subblock:</p> <p>Type: 4 (DNS domain name, 0x0400) Length: 20 bytes (0x1400) Data: "domain.com"</p> </td> </tr> </table>	0x02000c0044004f00 4d00410049004e00	<p>Domain name subblock:</p> <p>Type: 2 (Domain name, 0x0200) Length: 12 bytes (0x0c00) Data: "DOMAIN"</p>	0x01000c0053004500 5200560045005200	<p>Server name subblock:</p> <p>Type: 1 (Server name, 0x0100) Length: 12 bytes (0x0c00) Data: "SERVER"</p>	0x0400140064006f00 6d00610069006e00 2e0063006f006d00	<p>DNS domain name subblock:</p> <p>Type: 4 (DNS domain name, 0x0400) Length: 20 bytes (0x1400) Data: "domain.com"</p>
0x02000c0044004f00 4d00410049004e00	<p>Domain name subblock:</p> <p>Type: 2 (Domain name, 0x0200) Length: 12 bytes (0x0c00) Data: "DOMAIN"</p>							
0x01000c0053004500 5200560045005200	<p>Server name subblock:</p> <p>Type: 1 (Server name, 0x0100) Length: 12 bytes (0x0c00) Data: "SERVER"</p>							
0x0400140064006f00 6d00610069006e00 2e0063006f006d00	<p>DNS domain name subblock:</p> <p>Type: 4 (DNS domain name, 0x0400) Length: 20 bytes (0x1400) Data: "domain.com"</p>							

<pre>0x0300220073006500 7200760065007200 2e0064006f006d00 610069006e002e00 63006f006d00</pre>	<p>DNS server name subblock:</p> <p>Type: 3 (DNS server name, 0x0300) Length: 34 bytes (0x2200) Data: "server.domain.com"</p>
<pre>0x00000000</pre>	<p>Terminator subblock:</p> <p>Type: 0 (terminator, 0x0000) Length: 0 bytes (0x0000)</p>

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 Domain Name	security buffer

36	User Name	security buffer
44	Workstation Name	security buffer
(52)	Session Key (<i>optional</i>)	security buffer
(60)	Flags (<i>optional</i>)	long
52 (64)	<i>start of data block</i>	

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 domain 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. In this case, the data block begins at offset 52, immediately following the workstation name security buffer. 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 domain name is a security buffer containing the authentication realm in which the authenticating account has membership. 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 largely unknown, and is often empty when included; it is apparently relevant in newer signing and sealing mechanisms. The Open Group documentation states that it additionally plays a role in datagram-style authentication.

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.

Responding to the Challenge

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

- LM (LAN Manager) Response - Sent by most 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.

For more detailed information on these schemes, it is highly recommended that you read Christopher Hertel's [Implementing CIFS](#), especially [the section on authentication](#).

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 A](#) for a sample implementation in Java):

1. The user's password (as an OEM string) is converted to uppercase.
2. This password is either null-padded or truncated 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.

This 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 "0x5345435245543031000000000000".
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:

```
01010010 10100010 01010000 01101010 00100100 00101010 01010000 01100000
```

(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:

```
01010010 10100010 01010001 01101011 00100101 00101010 01010001 01100001
```

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:

```
00110000 10000000 00000000 00000000 00000000 00000000 00000000 00000000
```

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

```
00110001 10000000 00000001 00000001 00000001 00000001 00000001 00000001
```

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 A](#) 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 "0x53006500630052004500740030003100" 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).

- 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).
- 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. While its effectiveness in this regard is questionable, it does at least provide a more secure replacement for the LM response. 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 A](#) for a sample implementation in Java):

- The NTLM password hash is obtained (as discussed previously, this is the MD4 digest of the Unicode mixed-case password).
- The Unicode uppercase username is concatenated with the Unicode uppercase authentication target (domain or server name). 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.
- 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 Challenge	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:

Domain: 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 "0x53006500630052004500740030003100" 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 uppercase 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 challenge"; we will use the not-so-random "0xffffffff0011223344". Constructing the rest of the blob is easy; we just concatenate:

```

0x01010000          (the blob signature)
0x00000000          (reserved value)
0x0090d336b734c301 (our timestamp)
0xffffffff0011223344 (a random client challenge)
0x00000000          (unknown, but zero will work)
0x02000c0044004f00 (our target information block)
 4d00410049004e00
 01000c0053004500
 5200560045005200
 0400140064006f00
 6d00610069006e00
 2e0063006f006d00
 0300220073006500
 7200760065007200
 2e0064006f006d00
 610069006e002e00
 63006f006d000000
 0000
0x00000000          (unknown, but zero will work)

```

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

```

0x0123456789abcdef0101000000000000
 0090d336b734c301ffffffff0011223344
 0000000002000c0044004f004d004100
 49004e0001000c005300450052005600
 450052000400140064006f006d006100
 69006e002e0063006f006d0003002200
 7300650072007600650072002e006400
 6f006d00610069006e002e0063006f00
 6d0000000000000000000000000000

```

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
0101000000000000000090d336b734c301
ffffffff00112233440000000002000c00

```

```

44004f004d00410049004e0001000c00
53004500520056004500520004001400
64006f006d00610069006e002e006300
6f006d00030022007300650072007600
650072002e0064006f006d0061006900
6e002e0063006f006d00000000000000
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 A](#) 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 uppercase authentication target (domain or server name). 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 challenge is created (this is the same client challenge used in the NTLMv2 blob).
4. The challenge from the Type 2 message is concatenated with the client challenge. 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 challenge to form the 24-byte LMv2 response.

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

Domain: DOMAIN

Username: user

Password: SecREt01

Challenge: 0x0123456789abcdef

1. The Unicode mixed-case password is "0x53006500630052004500740030003100" 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 uppercase 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. A random 8-byte client challenge is created. From our NTLMv2 example, we will use "0xffffffff0011223344".

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

```
0x0123456789abcdef ffffffff0011223344
```

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 challenge to obtain the 24-byte LMv2 response:

```
0xd6e6152ea25d03b7c6ba6629c2d6aaf0 ffffffff0011223344
```

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 in environments which do not support NTLMv2 authentication.

The calculation of the NTLM2 session response is similar to the NTLM response; it effectively replaces both the LM and NTLM response fields as follows (see [Appendix A](#) for a sample implementation in Java):

1. A random 8-byte client challenge is created.
2. The client challenge 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 challenge 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 challenge is created; we will use "0xffffffff0011223344", as in the previous examples.
2. The challenge is null-padded to 24 bytes:

12	0x180018006a000000	<p>LM Response Security Buffer:</p> <p>Length: 24 bytes (0x1800) Allocated Space: 24 bytes (0x1800) Offset: 106 bytes (0x6a000000)</p>
20	0x1800180082000000	<p>NTLM Response Security Buffer:</p> <p>Length: 24 bytes (0x1800) Allocated Space: 24 bytes (0x1800) Offset: 130 bytes (0x82000000)</p>
28	0x0c000c0040000000	<p>Domain Name Security Buffer:</p> <p>Length: 12 bytes (0x0c00) Allocated Space: 12 bytes (0x0c00) Offset: 64 bytes (0x40000000)</p>
36	0x080008004c000000	<p>User Name Security Buffer:</p> <p>Length: 8 bytes (0x0800) Allocated Space: 8 bytes (0x0800) Offset: 76 bytes (0x4c000000)</p>
44	0x1600160054000000	<p>Workstation Name Security Buffer:</p> <p>Length: 22 bytes (0x1600) Allocated Space: 22 bytes (0x1600) Offset: 84 bytes (0x54000000)</p>
52	0x000000009a000000	<p>Session Key Security Buffer:</p> <p>Length: 0 bytes (0x0000) Allocated Space: 0 bytes (0x0000) Offset: 154 bytes (0x9a000000)</p>
60	0x01020000	<p>Flags:</p> <p>Negotiate Unicode (0x00000001) Negotiate NTLM (0x00000200)</p>
64	0x44004f004d004100 49004e00	Domain 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 server must be compatible in order for authentication to be successful. The result is that the overwhelming majority of hosts just use the default setting, and NTLMv2 authentication is rarely seen in deployed systems.

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 Levels 1 and higher, NTLM2 session security is supported. Only Levels 0 and 3 are available on Win9x-based systems (Windows 95, Windows 98, and Windows ME); these platforms do not support the NTLM response, and instead send only the LM response in the Type 3 message. 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 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.

NTLMSSP and SSPI

Now that we have a working knowledge of NTLM authentication, it is appropriate to look at how NTLM fits into the "big picture".

Windows provides an authentication 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 authentication. 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 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 go into the details of SSPI, but we will briefly outline 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.

At this point, we have established a fairly decent fundamental understanding of NTLM. We will now examine its use 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 TlRMTVNTUAABAAAABzIAAAAYABgArAAAACwALACAAAABXT1
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 TlRMTVNTUAACAAAADAAMADAAAABAoEAASNFZ4mrze8
AAAAAAAAAAGIAYgA8AAAARABPAE0AQQBjAE4AAgAMAEQATwBNAEEASQBOAAEADABTA
EUAUgBWAEUAUgAEABQAZABvAG0AYQBpAG4ALgBjAG8AbQADACIAcwBlAHlAdgBlAHl
ALgBkAG8AbQBhAGkAbgAuAGMAbwBtAAAAAAA=
```

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
AACAAIAEwAAAAWABYAVAAAAAACAaAAAAAQIAAEQATwBNAEEASQBOAHUAcwBlAHIA
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:

```
TlRMTVNTUAABAAAABzIAAAYABgArAAAACwALACAAAABXT1JLU1RBVElPTkrRPTUFJTg==
```

6. The server replies with the Type 2 challenge message (again, Base-64 encoded). This is sent 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
BvAG0AYQBpAG4ALgBjAG8AbQADACIACwBlAHlAdgBlAHlALgBkAG8AbQBhAGkAbgAu
AGMAbwBtAAAAAAA=
```

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

```
TlRMTVNTUAADAAAAGAAAYAGoAAAAAYABgAggAAAAwADABAAAAACAAIAEwAAAAWABYAVA
AAAAAAAAACaAAAAAQIAAEQATwBNAEEASQBOAHUAcwBlAHlAVwBPFIASwBTAFQAQQBU
AEkATwBOAMM3zVy9RPyXgqZnr2lCfG3mfCDC0+d8ViWpjBwx6BhHRmspst9GgPOZWP
```

```
uMITqcxg==
```

- 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:

- 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
```

- 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.
```

- The client initiates NTLM authentication by sending an AUTHENTICATE command specifying NTLM as the authentication mechanism:

```
0001 AUTHENTICATE NTLM
```

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

```
+
```

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

```
TlRMTVNTUAABAAAABzIAAAYABgArAAAACwALACAAAABXT1JLU1RBVElPTkrRPTUFJTg==
```

- The server replies with the Type 2 challenge message (again, Base-64 encoded). This is sent 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
BvAG0AYQBpAG4ALgBjAG8AbQADACIAcwBlAHlAdgBlAHlAlgBkAG8AbQBhAGkAbgAu
AGMAbwBtAAAAAAA=
```

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

```
TlRMTVNTUAADAAAAGAYAGoAAAAAYABgAggAAAAwADABAAAAACAAIAEwAAAAWABYAVA
AAAAAAAACaAAAAAQIAAEQATwBNAEEASQBOAHUAcwBlAHlAVwBPFIASwBTAFQAQQBU
AEkATwBOAMM3zVy9RPyXgqZnr2lCfG3mfCDC0+d8ViWpjBwx6BhHRmspst9GgPOZWP
uMITqcxg==
```

- 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:

- 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
```

- 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
```

- 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.

- 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
BvAG0AYQBpAG4ALgBjAG8AbQADACIACwBlAHlAdgBlAHlALgBkAG8AbQBhAGkAbgAu
AGMAbwBtAAAAAAA=
```

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

```
TlRMTVNTUAADAAAAGAAYAGoAAAAAYABgAggAAAAwADABAAAAACAAIAEwAAAAWABYAVA
AAAAAAAAACaAAAAAQIAAEQATwBNAEEASQBOAHUACwBlAHlAVwBPAFIASwBTAFQAQQBU
AEkATwBOAMM3zVy9RPyXgqZnr2lCfG3mfCDC0+d8ViWpjBwx6BhHRmspst9GgPOZWP
uMITqcxcg==
```

- 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.

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/)

<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 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.

[Implementing CIFS: The Common Internet FileSystem](http://ubiqx.org/cifs/)

<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. Inaccurate in some regards, but still quite useful.

[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.

Appendix A: 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 challenge.
 *
 * @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 clientChallenge The random 8-byte client challenge.
 *
 * @return The NTLMv2 Response.
 */
public static byte[] getNTLMv2Response(String target, String user,
    String password, byte[] targetInformation, byte[] challenge,
    byte[] clientChallenge) throws Exception {
    byte[] ntlmv2Hash = ntlmv2Hash(target, user, password);
    byte[] blob = createBlob(targetInformation, clientChallenge);
    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 clientChallenge The random 8-byte client challenge.
 *
 * @return The LMv2 Response.

```

```

*/
public static byte[] getLMv2Response(String target, String user,
    String password, byte[] challenge, byte[] clientChallenge)
    throws Exception {
    byte[] ntlmv2Hash = ntlmv2Hash(target, user, password);
    return lmv2Response(ntlmv2Hash, clientChallenge, challenge);
}

/**
 * Calculates the NTLM2 Session Response for the given challenge,
using the
 * specified password and client challenge.
 *
 * @param password The user's password.
 * @param challenge The Type 2 challenge from the server.
 * @param clientChallenge The random 8-byte client challenge.
 *
 * @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 challenge, null-padded to 24 bytes.
 */
public static byte[] getNTLM2SessionResponse(String password,
    byte[] challenge, byte[] clientChallenge) throws Exception {
    byte[] ntlmHash = ntlmHash(password);
    MessageDigest md5 = MessageDigest.getInstance("MD5");
    md5.update(challenge);
    md5.update(clientChallenge);
    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.toUpperCase();
    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 challenge).
 * @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);

```

2

epoch.

```

    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 challenge.
 *
 * @param targetInformation The target information block from the Type
 *
 * message.
 * @param clientChallenge The random 8-byte client challenge.
 *
 * @return The blob, used in the calculation of the NTLMv2 Response.
 */
private static byte[] createBlob(byte[] targetInformation,
    byte[] clientChallenge) {
    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 += 11644473600001; // milliseconds from January 1, 1601 ->
    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 + clientChallenge.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(clientChallenge, 0, blob, offset,
                clientChallenge.length);
        offset += clientChallenge.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;
    }
}
}
```

All trademarks mentioned in this document are the property of their respective owners.

Copyright © 2003 Eric Glass

Permission to use, copy, modify, and distribute this document for any purpose and without any fee is hereby granted, provided that the above copyright notice and this list of conditions appear in all copies.